Bang for the Buck Test Automation

Elisabeth Hendrickson

January 2001

You've heard the horror stories.

Hundreds of thousands of dollars spent on test automation that found 2 bugs. Thousands of person-hours spent writing automated tests that became useless as soon as the interface changed. Test automation tools that became shelf-ware within a few weeks of purchase because the organization wasn't really ready to automate.

In 1994, I worked on a test automation project that failed spectacularly. The test manager asked us to "automate everything." A few weeks into the project, the manager asked me, "so how long before we see some benefits?"

"It will take us about 2 years to automate everything," I replied, giving him the full estimate that my team and I had worked out.

"2 years!?!"

"Yup."

The manager's face clouded as he waved his arms. "That's not acceptable. I want you to think about Bang for the Buck!"

I did. Since then, I've used test automation as a part of an overall test strategy. As a result, I have examples of good test automation stories. A tool that took a day to write found serious bugs that would not have been found otherwise. A script used information about the software under test to generate tests as it executed. Small disposable scripts set up the test environment and saved testers hours.

These stories are not about comprehensive press-a-button-and-the-tests-run-overnight-while-you-sleep test suites. They aren't about attempting to reach a goal of 100% automated. They're about writing simple tools to make the process of testing easier. These stories inspired this paper.

The Problem with Return on Investment (ROI) Numbers

Before we can talk about having a maximizing return on investment, we need to define the term.

There are a number of ways to calculate your return on investment for automated tests. Most techniques boil down to:

1. Determine a dollar figure to represent the benefit of the automated tests.
2. Determine a dollar figure to represent the cost of the automated tests.
3. Divide the $benefit by $cost.

This is your ROI on test automation. The argument is that if you spent $50,000 automating tests that saved you $100,000, your ROI is 200%: a good investment.

Unfortunately, this method of calculating a precise return on investment ignores some factors:

It is difficult to put an accurate dollar amount on the benefit of test automation. I have always had difficulty putting a price tag on the capabilities provided by automating tests. One common method, calculating a dollar figure based on the number of times the automated tests were run, overstates the actual value. Did the automated tests suddenly become twice as valuable because I ran them 10 times instead of 5? Whether the benefits include speed, saved tester time, enhanced bug finding capability, or more consistent test effort, I find it almost impossible to quantify the benefits in terms of dollars.

It is difficult to put an accurate price on an automation effort. Although I can sum up the out-of-pocket expenses associated with using an automation tool, I find it difficult to quantify the fuzzier costs. For example, I can create a budget that includes the cost of the tool, training, extra machines on which to run the tests, and the salaries for people dedicated to test automation. However, I have a much harder time predicting ongoing maintenance costs or accounting for the time spent by other programmers and manual testers helping the test automation effort. I also have a difficult time deciding how to account for costs shared between several test automation initiatives.

ROI figures impress executives. However, since it's too easy to overstate value and understate cost, I don't find them useful for determining what to automate when. Fortunately, precise ROI calculations are not necessary for making good decisions about test automation strategy.

If you already know that you want to implement some form of test automation but are unsure where to start, a simple 2x2 matrix can help you determine your automation strategy. You don't need precise dollar figures—you need to assess the relative costs and benefits of various approaches in your environment.

Redefining Return: High and Low

Rather than trying to put a value on test automation, think about it in terms of high benefit and low benefit. High benefit test automation finds significant bugs or saves testers a significant amount of time.

"Significant" depends on your environment. I consider finding critical severity bugs or saving testers more than 10 hours per build cycle to be a significant return.

By contrast, low return test automation is likely to find a few bugs or save a small amount of time. Low benefit test automation can be worthwhile. Even small time savings can be a boon to a test group under a lot of pressure.

Redefining Cost: High and Low

There are a number of factors influencing the cost of test automation:

- How much effort is involved?
- Is there someone on staff who knows how to do the work?
- Do we have an appropriate tool?

- Does it build on what we've already done?
- How much is the software under test changing?

Even if the automation effort is minimal, the automation is high cost if it requires buying additional tools or relies on a test automation infrastructure that doesn't exist yet. Further, otherwise low cost automation can become high cost as maintenance costs mount due to changes in the software under test.

I consider a test automation proposal to be low cost if I don't have any significant ($1000 or less) additional out-of-pocket expenses, if it takes a week or less to create, and if maintenance is not an issue or takes less than an hour a week on average. If I need to buy software licenses or machines or if it takes more than a week to create the test automation, I consider it to be high cost.

The 2x2 Cost/Benefit Matrix

Defining costs and benefits in high-low terms allows you to divide the world neatly into four categories of test automation. The following matrix provides examples of test automation in each of these categories.

	Low Benefit	High Benefit
Low Cost	Setup scripts Throwaway scripts Partial verification	"Poka-Yoke" scripts "Hammer" scripts
High Cost	Many full regression test suites	Data driven tests Automatic on-the-fly test generation

This matrix provides a framework for thinking about test automation ROI without calculating numbers.

Caveat: *In reading through the examples in each category, note that "low" and "high" vary from environment to environment. Don't take my word for the relative cost or benefit of a given type of test automation. The cost and benefit may be different for your environment.*

This matrix also provides a way to think about the risk of choosing a given test automation strategy. If the software under test is still changing dramatically with every new build, investing in high-cost test automation is risky. Every change to the software under test incurs more maintenance costs. In this situation, use low-cost techniques to assist testing. Invest in the testability of the software. But don't attempt to create large data-driven test suites if the data elements are changing on a daily basis.

Low Investment, Low Return Examples

Setup Scripts

In one informal survey of a group of testers testing enterprise client/server software, I learned that they spent 30 - 50% of their time performing setup tasks. By writing automation scripts to perform a handful of the setup tasks, the testers saved between 5 – 10 hours a week. The scripts didn't try to accomplish all the setup tasks—just those that were easy to automate.

This is one area where record and playback can pay off. The script doesn't need to verify anything. If it breaks because of a change in the software under test, you can just re-record it. Low investment scripts can contribute to the test effort and pay for themselves quickly.

Throwaway Scripts

In one case I created throwaway tests because I needed to be able to exercise an interface was changing quickly. I didn't try to make the scripts maintainable. I'm sure they no longer exist today.

In another case, I needed to extract information about errors from logs. I quickly created a Perl script to parse the logs and pull out just the information I needed. This quick and dirty script later became the basis for a more permanent, maintainable utility. However, the first iteration was ugly—code only its creator could love. It also took me less than an hour to create and pinpointed several problem areas needing further investigation.

Throwaway scripts don't become part of a permanent toolbox. They're temporary. You might not even check them into source control. These kinds of scripts must pay for themselves after one or two test runs because you might not run them any more than that. Quick and dirty doesn't work as a long-term strategy, but it does work as a solution to a temporary problem.

Partial Verification

Sometimes it's difficult to write automated tests to verify the results of a test completely. For example, consider testing a client/server system where many testers share the server environment. You cannot always know exactly what data is in the server. Other testers will have added, deleted, and modified records. Verifying that a search returned the right results set becomes a tricky business.

When testing in this kind of environment, I used a combination of automated and manual verification. The automated scripts verified the behavior of the client—did the correct window appear and did any results appear? I manually verified that the results set was correct given the current contents of the database.

This is imperfect verification [KANER00]. The result is usable but incomplete test automation. This is one way to realize rapid payback.

Low Investment, High Return Examples

Poka-Yoke Tests

"Poka-Yoke" [ROBINSON98] means "mistake-proofing." An example is available on most computers: the 3.5" floppy drive. If you attempt to insert the floppy disk in any orientation other than the correct one, it will not fit. The disks are a little longer than they are wide, so you cannot put them in sideways. There is a mechanism in the floppy drive to keep you from putting it in backwards or upside down.

Test automation tools that help "mistake proof" the software development process have enormous value: they can save the testers hundreds of person hours if they find a subtle problem that would cause the test team to start over again.

For example, consider a utility to ensure that the executable files in a given build have the correct version numbers, that all the needed files have been delivered, and that the configuration files contain the correct settings. Humans might not catch the difference between 2.0.3.1534 and 2.0.3.1584 after looking at 20 file versions. An automated program will.

Hammer Tests

Hammer tests truly hammer the software under test. They don't necessarily verify any results; the goal is to make the software fail. These are "fall over dead" tests—how much punishment can the software takes before it fails catastrophically?

Two relatively simple tools provide examples. Each found critical bugs in a software project. One tool generated an enormous amount of random test data and imported it into the system under test. The other tool disabled the software under test at random times by causing the machine to shut down or telling the software to disable itself directly. Both tools found serious problems.

Each tool took a couple of days to write in Visual Basic. A test automation engineer with extensive programming experience wrote these tools—this particular example is only "low investment" if you already have such a person on staff who can write the tool.

High Investment, High Return Examples

Data Driven Tests

In data driven testing, the automated script reads a data file to get the values with which to test.

For example, consider an order entry form. Create a script to read a file and iterate through the lines of data. In the data file, include long names, short names, names with international characters, etc. Include domestic and international addresses. In other words, include all the data with which you need to test.

An advantage of this automation architecture is that adding tests becomes trivial. The data files can be handed off to non-technical testers for maintenance.

I have implemented data driven tests several times. Each time, it took a relatively large investment to create the structure to make the data driven tests possible. However, the investment paid off both in saved tester time and in ability to automate new test cases without involving the test automation team.

Further, investing in a data driven test automation architecture lowered the cost of implementing other automated test strategies.

Automatic On-the-Fly Test Generation

4Test, the scripting language for Segue's Silk family of tools, allows you to encode the behavior of the elements in the system under test in the test frame. You can then write scripts to walk through the interface of the software under test and verify the behavior described in the frame file [HENDRICKSON97].

For example, in one UI-intensive configuration application, I included the minimum value, maximum value, and expected error window identifiers in the field declarations. As the script executed, it automatically verified that the correct error window appeared when it entered data outside the boundaries and that it could enter data inside the boundaries successfully. In this case I was able to create and run 256 automated test cases in a two-day period. The script found several bugs and saved the manual testers an enormous amount of time.

In another case, I used a similar technique on a web-based application. Web links, especially automatically generated links, can be error-prone. By associating the destination page with the links in the declarations, I was able to automatically verify that the links went to the correct page as a part of functional testing.

I consider this strategy to be high-cost because it requires a significant investment in test automation infrastructure and testing tools. It also requires that the person implementing the automation have extensive expertise with the test tool. However, this technique might be low-cost in your environment if you already have a test automation infrastructure and expertise on staff.

Conclusion

I've had successes and failures with test automation. My greatest successes came from having an effective strategy for using the tools at my disposal. My worst failure was a result of assuming we needed to "automate everything." [HENDRICKSON98]

"Automate everything" isn't a strategy; it's a trap. What's "everything" and why does it need to be automated? What if automating everything is more expensive than anyone thought possible? What if it we succeed in creating a wonderfully comprehensive automated test suite and it doesn't find any bugs?

The 2x2 cost/benefit matrix helps me keep from falling into the "automate everything" trap. Are there opportunities to create simple scripts to save time and find bugs? That's what "Bang for the Buck" means. Maximizing returns, minimizing costs.

I hope that the examples and framework in this paper can help you define a "Bang for the Buck" test automation strategy for your environment.

Bibliography

[KANER00]
Kaner, Cem, "Architectures of Test Automation: Alternatives to GUI Regression Testing," *Proceedings from STARWest 2000*

[ROBINSON97]
Robinson, Harry, "Using Poka-Yoke Techniques for Early Defect Detection", STAR 1997 proceedings

[HENDRICKSON97]
Hendrickson, Elisabeth, "Get More Out of Your Automation: Put More in Your .inc," Segue Quest User Conference 1997 Proceedings

[HENDRICKSON98]
Hendrickson, Elisabeth, "The Difference Between Test Automation Success and Failure," STARWest 1998 Proceedings

Acknowledgements

The following people helped me refine my thinking about test automation: Bret Pettichord, Noel Nyman, Brian Marick, Cem Kaner, Doug Hoffman, Brian Lawrence, James Bach, and Michael Snyder. I am especially grateful to Noel Nyman for reviewing early drafts of this work. I am grateful to Harry Robinson for introducing me to "Poka-Yoke" techniques.

Some of these ideas were discussed at LAWST 2, 4, and 5. I am grateful to the LAWST participants for their insights and analysis: Chris Agruss, Tom Arnold, James Bach, Richard Bender, Jim Brooks, Jack Falk, Karla Fisher, David Gelperin, Chip Groder, Elisabeth Hendrickson, Doug Hoffman, Keith W. Hooper, III, Bob Johnson, Cem Kaner, Brian Lawrence, Tom Lindemuth, Brian Marick, Noel Nyman, Jeff Payne, Bret Pettichord, Drew Pritsker, Johanna Rothman, Jane Stepak, Loretta Suzuki, Melora Svoboda, Ned Young

Elisabeth Hendrickson

Elisabeth Hendrickson is currently an independent consultant specializing in software quality and testing. With over a dozen years of experience in the software field, Elisabeth has seen several examples of better testing leading to worse quality. Prior to becoming an independent consultant, Elisabeth was the Director of Quality Engineering and Project Management at Aveo Inc. You can read more of Elisabeth's thoughts on software management, quality, and testing at www.qualitytree.com or reach her at esh@qualitytree.com.