BeaufortFairmont

5 Secrets to Automation Success

A White Paper by Paul Merrill, Consultant and Trainer at Beaufort Fairmont, LLC

Secret #1

Practice Exceptional Leadership

If you're leading an automated testing effort, what exactly do you want to achieve? Have you or your manager defined your goals? If not, you will not and cannot succeed. Goals must predate means.

In any field, good leaders must understand their objectives, communicate them clearly, and work with others to achieve the objective. This preparation is particularly crucial for successful automated testing. Vaguely defined goals such as "automating as much as possible" are not attainable. Instead, an effective leader might create a goal of "95% code coverage by December 31st." Define success by setting a clear target for your automation effort.

You will encounter obstacles along the way to your objective – that's expected. A good leader prepares her team to meet and conquer them. Your automation tool may fail to accomplish a critical goal. There may be some misunderstanding between what the salesperson told you and what you find the tool can actually do. How are you going to handle that? Are you prepared? Do you have the expertise and skill-set to overcome the challenges that will arise in the course of your project?

A good leader staffs her team to overcome obstacles.

Expect a High Return

For every dollar invested in labor, the client got back **$8.32**.

What is a good return? I had this question too. In order to find an answer, I analyzed a number of projects that I had been involved with over the last decade that used test automation. I wanted to gain insight into what the key metrics in an automation effort were and how they related to project success or failure.

The primary justification most managers use for an automation effort is saving labor. I determined the labor costs (Cost of Investment) for these projects by talking with recruiting agencies that specialize in recruiting these skill-sets locally. Additionally, I needed to understand the Gain from the Investment. There are several ways a person could determine this, but in this case, I was looking for the amount of testing time the company gained by pursuing automation.

Each test case does work that a human would otherwise do. I took the amount of time a human would take to do the work and multiplied it by the number of tests cases. Each type of test case had a different average amount of work it would take a human to do, so that was considered.

So the total amount of labor hours generated each time the automated test suite ran times multiplied by the number of times the suites ran over the course of the project was the gain.

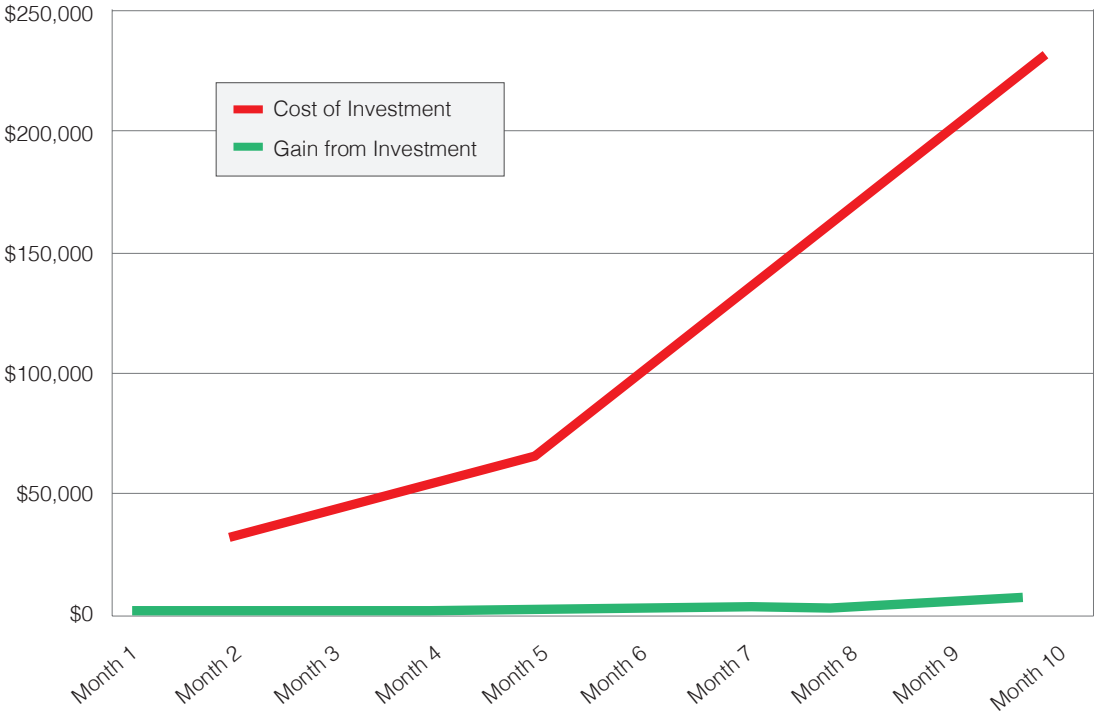
Knowing these two factors, I was able to calculate the Return on Investment (ROI).

$$ROI = \frac{(\text{Gain from Investment} - \text{Cost of Investment})}{\text{Cost of Investment}}$$

Return on Investment gives us a good understanding of how each invested dollar pays off.

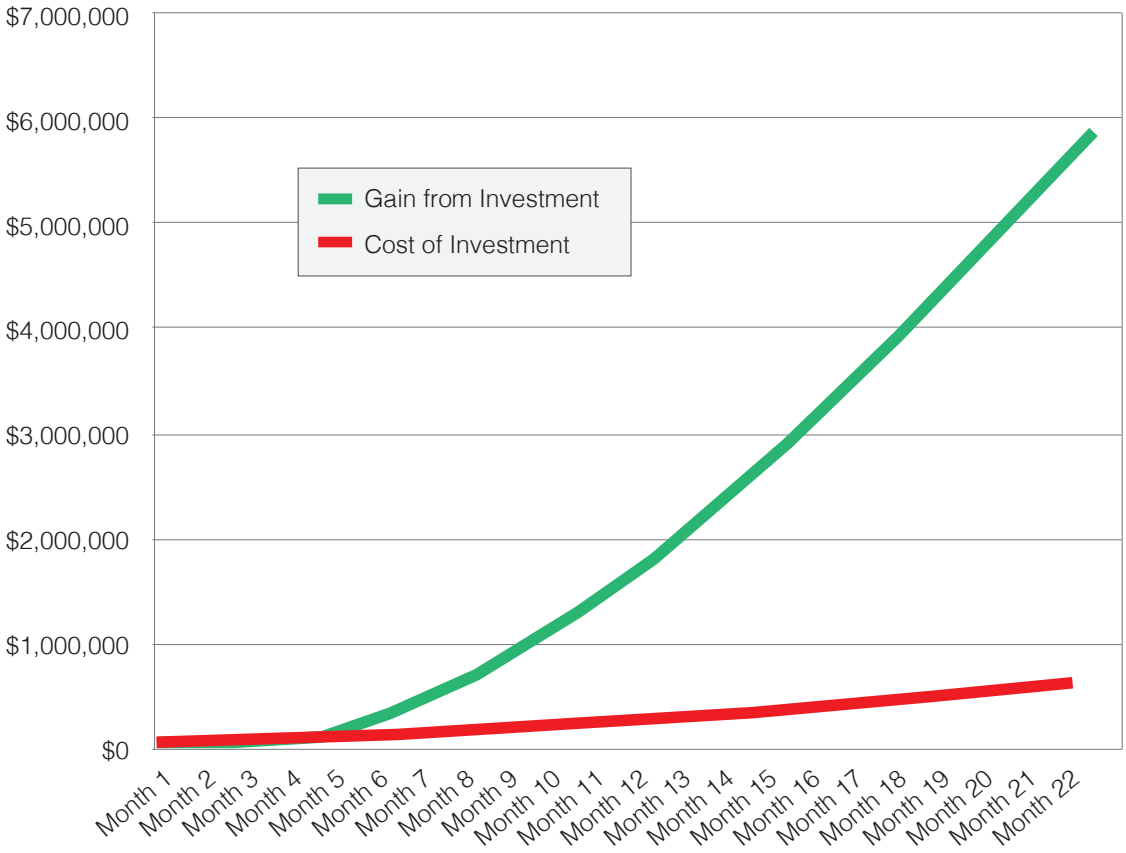
Of the projects I analyzed, I decided to take the most successful and the least successful projects to use as reference points in this discussion. Once again, these are real projects that intended to use automation to improve the efficiency of their testing efforts.

Project A



ROI is calculated as a ratio. On Project A ROI was -0.97. Yes, that's negative. That means that for every dollar invested in the project, 97 cents were thrown away.

Project B



Project B was quite different. For every dollar invested in labor, the client got back $8.32.

Notice the amount of labor was not all that different in these two projects. After 10 months, Project A was estimated to be at $231k and Project B was at $228k. The gains on the two projects, however, are significantly different. I'll talk more about these projects and what made Project B so successful as I reveal the other four secrets.

Speed of Return

An investment on automated testing should begin to pay off immediately. Waiting six months for someone to "create a framework" is foolish -- time invested in automation should pay off in weeks, not months.

You don't need to spend hundreds of thousands of dollars on software applications for a popular brand or attractive interface. You need immediate results. Good leadership will drive this. Good leadership will understand and create attainable goals, set high standards, and help the team accomplish goals effectively.

Setting a target ROI

> Each tester should create an average of 6-7 test cases per day.

After studying the reference projects above, I found several milestones that appear to be constant.

Milestones:

- Average 6-7 test cases/day per person
- After three months, the value of daily test runs must be equal to or greater than the daily effort to create test cases
- Nightly runs start in week one or two

As I discuss the next four secrets, I'll refer back to these reference projects and illuminate why these milestones are accurate, constant, and will work for your project.

Secret #2

Educate Your Testers

How do you make the gains we've examined? In Project A, automation failed to pay for itself. Instead, it cost hundreds of thousands of dollars in spite of many measures the company took to ensure success, such as:

- Hire only testers with experience in automation
- Purchasing training from the vendor of the tool they were using
- Investing months into building an automation framework

So why did it fail?

Project A failed for many reasons, but one of the biggest was education. In most cases, those who learn automated testing learn it on the job with little or no programming experience. Most people who attempt automation have implemented one or two systems. Few have built systems from the ground up, architected the infrastructure and framework from nothing. Many have witnessed and used systems that work. But as Sophocles said, "One must learn by doing the thing; for though you think you know it, you have no certainty, until you try." That's where Beaufort Fairmont's experience and expertise are second to none.

Often, managers of QA projects have not managed the construction phase of the development effort – which is very different from what is normally, a later/reactive testing cycle. This was the case in Project A. The manager had no hands-on Software Engineering experience and no experience leading a development effort.

Often, managers use automated testing as a place to put the people in their team that want to program but don't have the experience or skill necessary to join the development effort. It's an easy choice because the promise of automation is so heavenly.

If we're serious about automation, however, we should look at it as a product in and of itself – with as much value add for the company as the customer-facing product or service. And when this is the case, that the automation effort is as important a project as development of a production system, why would we put people without development experience and formal education in charge of the technical leadership of these efforts? Why do we plan them or resource them any differently from a normal development effort?

Automation efforts are as important and complicated as producing commercially viable software. Hiring above-average software engineers to develop these projects is also a difficult task. So where does this leave us?

Beaufort Fairmont offers a 3-day class called Automated Testing.

Educating Existing QA Engineers and Automated Testers.

I mentioned that in Project A the company paid for training from the software vendor. The training, however, was specific to the tool. It was training on how to use the tool – not principles of automation. And it cost tens of thousands of dollars!

Beaufort Fairmont offers a 3-day class for QA Engineers and Automated testers. The course is called "Automated Testing". This course is for QA Engineers with or without Automation experience. It is not about a specific tool, it is about the principles of automated testing. It covers how to choose a tool for a specific assignment, how to evaluate testing products and an overview of several software automation tools on the market.

We do hands-on work to learn a number of patterns and best practices in implementing testing frameworks, test suites and test cases. We cover continuous integration, TDD and ATDD. We even touch on JUnit and TDD as a mechanism to expand our skill-set.

This is a one-of-a-kind offering. You won't find it anywhere else. Check the Beaufort Fairmont web site (beaufortfairmont.com) for details.

Secret #3

Practice TDD, ATTD, and CI

Test Driven Development was originally created by developers to ensure the code they wrote did what customers requested. It was created in the SmallTalk community and made famous by Kent Beck with his "Test Infected" junit.org site. If you want an introduction to junit and TDD, junit.org is the place to go.

Test Driven Development (or TDD) uses a very simple pattern:

- Write a Failing Test
- Make the Test Pass
- Reduce Duplication

Note that this pattern forces the developer to write a test before writing the code. For many people, this concept can be difficult to understand. How can you write a test case before writing the code?

This strategy is not only possible, but offers many advantages. Coders become more focused, code gets written faster, has fewer defects, actual defects are easier to find, and unit tests are not forgotten, because you can't write any code without a failing test.

The major return from practicing TDD is that developers' feedback loop is tightened. They gain instant feedback from every thing they do in the code, and can know at all times whether they have broken the code or not.

TDD generally doesn't cover all test cases. It covers very specific test cases within specific sub-systems or classes.

The Course

Beaufort Fairmont offers a 3-day course in TDD. This course teaches students how to practice TDD in hands-on examples. It gives students the opportunity to face real-life issues all programmers will face with TDD in a safe and coached environment. The course also encompasses mock objects and patterns for TDD. Furthermore, students will learn about implementing TDD in Legacy codebases and techniques for doing so.

TDD for Testers

I break up testing in to 3 levels:

- Unit Tests
- Integration Tests
- Acceptance Tests

Unit Tests

Normally, TDD is practiced at the unit level by developers. The tests are tightly coupled to the code. This is white-box testing, where the test code has intimate knowledge of the code under test.

Integration Tests

Teams can practice TDD at the Integration Test level as well. When test teams use good tools and have the technical support they need, they can test production code before a user interface is developed.

How is that? Many times, engineers feel they are limited to testing only what they can see and touch – systems with user interfaces. But when teams learn to interface with a subsystem before and during implementation, we add another layer of feedback and early detection of defects. This is another good reason to have your testers attend the Beaufort Fairmont's Automated Testing Training course.

TDD at the unit level will not catch everything. When classes or modules are composed into subsystems, we use integration tests (white box tests) to test the integration of those classes or modules.

Every sub-system has an input and an output, although they are not always clearly defined. Often, testers must work with development to negotiate clear inputs and outputs to the sub-system for testing. Good code is tested code. We'll do whatever we need to ensure a high-quality product. If it means adding a mechanism for tests to interact with the subsystem, that's what we'll do.

Acceptance Tests

Acceptance tests are all about the customer.

Acceptance tests are all about the customer.

When a customer sees the results of unit tests, he should be able to ascertain the purpose of many of the tests by their name and organization. This is because while many of the test cases will be clear user-related functions, many will not. Unit tests will cover boundary conditions that most end-users and customers will never (and should never) have to consider.

When a customer sees the results of integration tests, he should be able to determine most if not all of what the test cases do simply by their names. Acceptance tests, on the other hand, are the domain of the end user. While it is likely that testers will write these tests, the end user is the person who determines what they should do and how the tested system should react.

Like TDD, acceptance testing doesn't have to wait until code is complete to be created. Acceptance Test Driven Development (or ATDD) is the practice of creating acceptance tests or tests which will only pass if acceptance criteria is met, prior to integrating the code into the system. Acceptance tests fail until the code under test meets acceptance criteria.

In our AT course, we offer hands-on examples of ATDD and teach the best practices and design principles associated with the practice of ATDD.

Continuous Integration

CI is the practice of developers integrating their code daily (if not more often) into the full code base. All levels of tests are run automatically. At minimum, test suites are run nightly. Many tests are run locally on dev machines before check-in. Many tests are run after every check-in.

Continuous integration depends on being able to build and deploy your product or service with the push of a button.

Applications like Jenkins, Hudson and Continuum allow you to set up automatic builds followed by test cases running automatically. Users are notified by email if a build breaks or test case fails.

This type of feedback ensures that you have a working codebase at all points in time. It is absolutely key to implementing automation successfully.

Secret #4

Select the Proper Tool

We're here to help.

Selecting the proper tool for your team is one of the most important keys to successful test automation. There are many tools out there, and many support and training options available for these tools, but the factors involved in choosing a tool are similar for each team.

Beaufort Fairmont has worked with many tools in the automation testing space, and we understand both their benefits and costs. We have the skills to sit down with your team and help make a non-biased decision with you. Further, if you pick a solution from the open-source space, we offer support contracts for many of these tools.

The choice of an automation tool can be daunting. We're here to help.

The main factors involved are:

- ✓ Maintenance of test cases
- ✓ Integration with Continuous Integration Server
- ✓ Speed of writing tests
- ✓ Ability to refactor test cases into reusable code
- ✓ Integration with source code control
- ✓ Budget
- ✓ Available Training
- ✓ Available Support
- ✓ Persisting results
- ✓ Visibility of results
- ✓ Available Metrics

One thing you don't see listed is integration with an existing defect system. Many managers believe this is the most import feature a system can have. In fact, most of the reasons managers want this are invalid and harm the development model.

For instance, one manager thought it would great if failed test cases created defects in the defect system. Sounds great, right? But what happens when every test fails one night because of a small error a developer introduced? Your defect system is flooded with defects that don't help anyone!

Is it helpful to know that the system broke last night? Absolutely. But the cost of adding hundreds or thousands of defects to a defect system is too high. Unless, of course, you want to make development look bad, and then you have other problems. See Secret #1.

Other teams think they want test cases persisted in a defect system in addition to the test system. This duplication generally leads to confusion. In my experience, there's rarely a need for this.

Most of the time, features like these are requested by managers who need a way to report on their team's metrics. When that is the objective, there are other ways to accomplish it rather than implementing purchasing features to duplicate test cases and automatically flood defect systems with unnecessary noise.

In other words, these are not key features in testing systems. Which is why I don't list them above.

Secret #5

Design Test Cases Effectively

If we're writing automated tests, what do we need to do in order to succeed?

- Test cases that are readable by human beings
- Test case names that define (succinctly) what they do
- Test cases that fail for one and only one reason
- Test cases organized from the perspective of reporting
- Repeated test case code is extracted
- Repeated literals are extracted

Several of these may seem obvious, but you may find others less so.

> Make test cases readable, make them look like a story.

Readability

One cost that is rarely accounted for in automated testing is the learning curve of an individual who wants to read a test case.

Perhaps a Business Analyst wants to see if the dev team has created a test for a certain situation. Perhaps a manager wants to investigate why a build failed. Perhaps you have a new tester writing automated tests for the first time, or a developer wants to see what a test case is doing. How long does it take that individual to read and understand a test case?

Reading a test case should be like reading one's native language. Make test cases readable, and make them look like a story. Test cases don't need lots of comments in the code, but they need keywords and methods that signify their function. This has the added benefit of helping you understand what you did in a year's time.

Naming Test Cases

A test case should fail for one and only one reason. Why is this important?

We create automated tests for a number of reasons. One reason is to replace human labor with computer labor. Too often, we think of that as the only reason. Another reason is to provide a clear assessment of what works in the system and what doesn't work in the system at any given time.

The faster a project team understands what's working and what isn't, the faster we're able to isolate a defect and fix it.

The clearer and more succinct we make the name of a test case, the easier it is to see which part of the system broke in the last test. It's much quicker to look at a report and know by the name of the test case what failed than it is to look at a test case number (for instance) and then have to look through the test to understand what it does.

This can save hours per day in understanding what worked and what didn't in a test run.

Fail for One and Only One Reason

Make the repeated parts of the test case reuseable.

This is one of the biggest reasons for failure I've seen in automated testing. For some reason, we think that we need to verify each step in a test case. Well, that's simply not true.

We don't like to do the same thing over and over. We know that when we go through a set of actions in the system under test, we should go ahead and check to make sure that each action works properly, because it saves time.

Computers, however, don't care about doing the same thing over and over again. (In fact, I believe they like it! It's a task at which they excel.) Instead of adding 20 verification points to one automated test, break that test up into 20 test cases. Make the repeated parts of the test cases reusable, and have one verification point in each test case.

In many cases, what you'll find is that you weren't testing all cases with the original 20 verification points to begin with. Sometimes, you'll need to go back and create multiple cases for each of the 20 new test cases.

When your test cases have one verification point, or one reason to fail, you'll see that it's much easier to name your tests. Additionally, it's much easier to understand them, organize them, report on them, and give and get feedback on them.

Finally, if you're not yet convinced, think of what happens when you've looked at a test case with 20 verification points. You're hard-pressed to know why the test even exists or what purpose it serves!

Organize Tests from a Reporting Perspective

If your team isn't using continuous integration, you're missing the boat on automation. You should be able to sleep at night knowing your test cases are running and your system under test will have a reliable status in the morning.

Part of most continuous integration systems is reporting on tests that fail. Your project team needs the ability to scan reports and know in minutes where in the system a defect resides (or doesn't reside). If your tests are organized in such a way that reporting is easy to look at and understand, your team will save a large amount of time looking into defects that were produced in the last 24 hours.

Reuse Repeated Test Code

As we discussed in Secret #4, you want to select a tool that allows you to reuse your test code. Maybe it allows you to create keywords, maybe its methods or functions or modules…regardless, the ability to reuse test code is a fundamental characteristic of a good testing tool.

Why? Because one of the biggest costs or benefits to your automation effort will be maintenance of test cases. Maintenance will be reduced if you can look at test code and make a change or a fix in one place as opposed to multiple spots in the code.

This is also one of the reasons traditional "record and playback" tools don't work well. In order to reuse repeated code and to reduce maintenance of test cases, you have to stop using record and playback.

So whenever you find code that looks similar, make it the same by introducing variables, and creating a function, method, or keyword with those variables as parameters.

But *bear in mind:* as with test case names, keywords, functions, or methods are more easily used, and more easily read when they are succinct and clearly named. The same goes with introducing variables into test code.

Closing

These are the Five Secrets of Successful Automated Testing. If you implement them, you will be five steps closer to success in your projects!

These are just five of the secrets I've learned over my career in software development. As much as I'd like to, it's impossible to summarize the lessons of an entire career in one whitepaper. That's why I've created multi-day courses for you and your teams. Check back at beaufortfairmont.com for more information on courses in your area. Each of these courses can be customized for your team, your needs, and your location.

The 5 Secrets to Automated Testing Success

Secret #1: Practice Exceptional Leadership
Secret #2: Educate Your Testers
Secret #3: Practice TDD, ATTD, and CI
Secret #4: Select the Proper Tool
Secret #5: Design Test Cases Effectively

Beaufort Fairmont is also available for consulting engagements and player-coach engagements. Contact us today at sales@beaufortfairmont.com or 1.888.385.4851. We look forward to hearing from you and learning about your projects!