EuroSTAR eBook
2017 Series

The Pathologies of Failed Test Automation Projects

Discover five failure patterns that contribute to test automation project failures

Michael Stahl

Intel Corporation, Israel

EuroSTAR Software Testing

biography

Michael Stahl is a SW Validation Architect at Intel. In the last 17 years Michael tested code for cable-modems, Smart TVs, graphics cards and is lately testing the code behind Intel's RealSense 3D camera technology.
In this role, he defines testing strategies and work methodologies for test teams, and sometimes even gets to test something himself - which he enjoys most.

Michael presented papers in SIGiST Israel, STARWest, STAREast, EuroSTAR and other international conferences. Michael is teaching SW Testing in the Hebrew University in Jerusalem.

EuroSTAR Software Testing

abstract

Many test automation projects start with a lot pf promise but fast become a mess. Initial solutions that start well and are full of promise often end up as brittle and unmaintainable monsters consuming more effort than they save. Political feuds flourish as different automation solutions compete for attention and dominance. Tests become inefficient in both execution time and resource usage. Disillusionment ensues, projects are redefined, and the cycle begins again.

This eBook analyzes five failure patterns that contribute to test automation project failures. Recognizing a failure patterns in your project is an alert signal that the project is heading in the wrong direction. After describing the patterns, actions are proposed to avoid or mitigate the impact of the failure patterns.

EuroSTAR
Software Testing

abstract

Even successful automation projects suffer from some aspects of these patterns. You may be able to apply some of the remedies to improve existing, successful projects.

Key takeaways

- Test automation projects are prone to suffer from known failure patterns
- Become aware of five common failure patterns
- Learn how to mitigate or completely avoid the failure patterns

EuroSTAR Software Testing

Pattern #1: Mushrooming

"Mushrooming" describes the evolution path of a small and simple automation utility as it develops into a full-fledged Test Automation Framework. The evolution is very natural and is a consequence of how organizations behave and make decisions. The end result is a large system that is critical to the organization but is also a consistent source of trouble and usually considered one of the main issues the test organization has.

Below is a description of this phenomena, and how you can identify the pattern as it happens. Later in this book I suggest how to avoid or mitigate this natural progression.

Stage 1: Small and Localized

Many test automation projects start in the following way:

A single tester, who has some programming or scripting skills, gets tired of re-running the same manual regression tests week after week. In her spare time, the tester writes some code to automate parts of her work load. Magically, a set of manual commands that took an hour to execute are done in less than a minute.

It is done by a simple, small automation tool, targeted to automate a very specific task.

The tester is naturally (and justifiably) proud of this achievement, and shows the results to the other team members.

> **Are we there yet?** (what signals to look for to know that an automation effort has reached this stage)

- The tool creator is the tool's user (single user)
- The tool is usually made unofficially; it's a personal initiative; many times it is "skunk work" – no one discussed or approved its development
- Key words (look for these in status reports or water-cooler conversations):
    - "Utility"
    - "Tool"

Stage 2: Generalization

Team members quickly realize how this solution applies to their own daily work and ask the initiator to help them achieve that: “if you add this simple capability, I will be able to use your tool as well!”.

It’s flattering to get these requests and they are usually easy to implement. Our tester implements some additional capabilities and more team members can now use the tool.

By now the tool is a bit more complicated but is still small enough to be supported by the original writer without a noticeable impact on the daily deliveries: instead of running manual tests, the time is put into the automation tool. The tests themselves run automatically, so less time is needed to get the work done.

Management is quite happy with this development: Test Automation was something that was always on the To-do list and the grassroots emergence is delightful. It looks good in the lab and it definitely looks good on the monthly reports.

Are we there yet?

- The tool serves more than one feature
- There are multiple users for the tool, but still a single owner/developer
- Maintenance and development of the tool takes >25% of the owner’s time
- There is an Automation Web Site[1]
- Key words
    - “Used by other testers”
    - “Common Libraries”[2]

[1] A web site for the tool is a good signal that more than one person is using it; it shows there is a need for disseminating information or for an update-delivery mechanisms. If the tool is still used by one person - why would you need a web site?

[2] As long as a tool serves a single feature, there are usually no common libraries of code. While good design practices would indicate that even in the case of serving a single feature, some of the code should be developed as “common library”, the realities are that common libraries are just not happening. The lack of proper software design expertise, which plays a key role in the Mushrooming pattern, means usually that common libraries are introduced only when it becomes painfully clear that they are needed.

Stage 3: Staffing

Life goes on: more code is added to the tool; more users (testers) and more features are covered. At this point, completing a test cycle on time is dependent on the automation tool.

And it does happen that sometimes the tool reports incorrect failures; or that a new test tool release is so buggy that it blocks everyone from getting much work done.

The tool's author is increasingly busy with doing automation work and has a hard time meeting her commitments to the test cycle. More than that: a lot of requests for added capabilities are being delayed since there is just so much a single person can do.

Eventually, management realizes that this automation thing is important enough and that it can't be done just as a side job of a single person. Additional heads are added and a new "test automation team" is officially created to continue the development of the automation tool.

Are we there yet?

- There are requests for additional manpower to work on automation
- A test automation team exists or is in the stages of being formed
- Automation Face-to-Face meetings take place[3]
- Tool-related issues delay the test execution cycles (this is an indication the tool is becoming complex and brittle)
- Key words:
  - "Tool Owner"; "Automation team"
  - "Framework"; "Infrastructure"
  - "Roll back"
  - "Bug fix release"

[3] In geo-dispersed organizations, automation developers are spread across geographical locations. It is common that the isolated teams have different opinions how to progress (what programming language to use; design decisions etc.). When the argument heats up, a common solution is to "get them in a room and don't leave until you see white smoke". This means a F2F meeting. Thus, calling a F2F is firstly an indication that many people are now involved with automation and a possible indication of friction between automation engineers across geographical locations.

Stage 4: Development of Non-Core features

As more capabilities are added to the tool and more tests are automated, it becomes clear that some test-management capabilities are needed in order to take full advantage of the automated tests.

Tests must be grouped together into test cycles; pass/fail results need to be collected and reported efficiently; automating the logistics of assigning test cases to test machines emerges as a dire need.

Additionally, testers ask for generic features – things that are not related to a specific technology, but more to test automation in general. For example: “When a test fails, run it again on another system”; “Implement timeout, so when a test is stuck, the system aborts the test and moves on”.

Code is written to automate the installation and configuration of the application under test.

Code is written to allow links between tests (“if this test fails, marks these tests as blocked”).

More and more of the automation team’s time is invested in developing a Test Framework – code to manage the test cases and test execution and not code that automates actual test cases. This is code that addresses something else than your Core Technology. Additionally, the automation team puts a rather large effort keeping the system running, fixing bugs and solving problems that lead to false fails.

Are we there yet?

- Much of the development effort is going into developing generic features (test-case management, test cycle management, data collection features)
- The system creates enough false fails to be a concern
- A lot of time is spent on analysis of test logs
- Key words:
    - “Test suite / Test cycle generation”
    - “Test results database”
    - “Robustness enhancement”
    - “Setup issues”
    - “False positives”

Stage 5: Overload

By now, you have a large testing framework that was developed internally. The framework is central to the daily life of your test and development organizations but suffers from many problems; most of the automation team's time is spent on keeping the test system running, instead of developing new capabilities.

In fact, many people are so unhappy with the system that they start blaming it for all kind of problems – some of which it has nothing to do with. It becomes clear that localized fixes won't do. It needs to be redesigned from the ground up.

Are we there yet?

- The automation team suffers from maintenance & logistics overload
- Users and customers overplay the system's limitations
- The system loses credibility. Test failures are suspected to be a test problem, not a product problem and manual reproduction is required as a standard procedure
- Some engineers start developing their own, stage 1 initiatives to solve their specific problem
- Key words
    - "Did it fail in manual test?"
    - "Architecture limitation"
    - "Refactoring"; "Redesign"
    - "…I can write a small program…"

Some ways to address and mitigate these problems will be given in the last section of this book.

Pattern #2: The Competition

Large organizations frequently end up with more than a single test automation system, for the reasons listed below. The competition is usually not a healthy one and leads to wasted effort and on-going conflicts. I have identified three variations of competition; all are related to the previous pattern of Mushrooming.

Competition (1st variety):

Mushrooming, as noted before, is a natural occurrence. It is therefore quite normal that the same pattern will develop in parallel in two (or even more) teams in the same organization; especially when the organization is large and spans across geographical areas.

At the beginning (stage 1, 2), there isn't really a problem. The automation solves localized needs. Many times, the teams developing these automation solutions are not aware that another team is also doing automation work; even if they knew, it would not look like something that calls for attention: the other team is solving *their* localized needs.

But as both teams move towards Stage 3, the fact that there are two solutions being built becomes more apparent – if only for the fact that you now have two Test Automation teams on the org chart. When the teams move to Stage 4, there is no way that this duplicity will go unrecognized: program managers need to collect tests results data from two systems, which makes their life difficult.

So what happens next? It is obvious: Both teams get together and peacefully merge their solutions into one system, implementing the best of both initial solutions…

You think?

All I can say is that I did not see this happen.

What does happen is a long series of fruitless discussions at the engineering level, where each team tries to convince the other – and management – that their code, choice of programming language, design, user Interface, feature set, database etc. is superior to the other solution. Therefore, management should cancel the other solution, and pick theirs.

While these discussions are going on (for months or years), both teams continue to develop their solution, proudly and surely moving towards Stage 5 – Overload. You will then have two systems in deep trouble, not just one.

Competition (2nd variety):

This is the case where teams are more coordinated and are aware that a test automation solution is under way. They look at the budding solution, like it, and start feeding requirements for enhancements, pushing the solution fast from Stage 1 to the following stages.

Since not all the requests can be accommodated by the limited test automation resources, some type of prioritization takes place. Some requests are implemented, some delayed a bit, and some requests somehow never make it to the releases.

At a certain point, a team whose requests are constantly being pushed out, will get annoyed and frustrated enough to start their own simple, localized solution… and we are back on the Mushrooming wagon and competition of the 1st variety.

Competition (3rd variety):

Another reason that will push teams to start their own Stage 1 solutions is when the leading test automation framework is in Stage 5 for too long. After a year or two of constantly fighting the system's inefficiency, some teams will get fed up enough to start their own Stage 1 solutions, which will in time, become a competing, Stage 5, inefficient behemoth.

Pattern #3: The Night Time Fallacy

One of the big selling points of test automation is the idea of "tests running at night". This is a compelling vision: as the workday gets close to its end, the testers fire their automated test system which will run unattended during the off hours, completing the test cycle and having the results ready when the testers come in the next morning. It is such an idyllic vision… you can almost hear the violins play.

But this vision drives some behaviors; behaviors that result in an inefficient test automation system.

If tests run at night, it seems they cost the same whether they run 1 hour or 8 hours. It's still all off-shift. This means that testers can add many more tests and don't have to be that diligent about what tests to add and how much test time these tests take.

As the project progresses, as older versions enter maintenance phase and new versions are added, more and more tests are added to the Nightly Run. Eventually, you run out of night and test runs continue into the day, the next night etc.

One way to deal with this would be to re-assess the test strategy; redesign tests, and reduce test count and test time. The problem is that optimizing the test suites calls for a lot of engineering time. Since the regression test suites seem to be doing the job (albeit with some inefficiency), it appears that the engineers' time it better used doing something else

A different approach is to get more machines and run tests in parallel.

> **A Corporate Truism:** It's easier to get budget for machines than for more testers.

This corporate-world fact-of-life means that in most cases organizations end up buying more machines and splitting the regression tests over a large machine pool. Test time is down again to the one-night length. More tests are added… more machines are added… more code is added to manage these machines…

Eventually, you hit another problem: Since tests were never written with optimization in mind, their logs are also inefficient. They either generate too much or too little information. In both cases, testers spend a good deal of their time going over test logs to figure out why something failed, what is a real issue and what can be ignored.

So eventually you end up with not enough testers, and learn another lesson:

> **Test Automation Truism:** More machines create work for more testers.

You end up with a lot of machines running an inefficient set of automated test cases and a large tests automation team that spends most of its time in maintenance and wading through results.

Three other phenomena exacerbate the above situation.

> **The Tragedy of the Commons (as applied to test systems)**
>
> Let's take an example: A project has five test teams, all using the same machine pool to run their (inefficient) regression tests. The pool is overloaded and it takes time to get the test results back. Each of the teams would do well to reduce its test time, but here is an interesting calculation:
>
> Assume each team consumes 12 hours of test. The regression tests end in 60 hours.
>
> Let's say that team A decided to invest time and make its tests 50% more efficient. They will now run in 6 hours. The overall regression test cycle time will be… 54 hours.
>
> So, for the super-human effort team A invested in achieving 50% test time reduction on their test suite, they effectively got a 10% test time reduction. And this is likely to be quickly swallowed up by another team's inefficient tests.
>
> The result is that teams have little incentive to improve their test time.
>
> It may be interesting to compare this situation to the economic principle named The Tragedy of the Commons.
>
> "The Tragedy of the Commons is the depletion of a shared resource by individuals, acting independently and rationally according to each one's self-interest, despite their understanding that depleting the common resource is contrary to the group's long-term best interests." (http://en.wikipedia.org/wiki/Tragedy_of_the_commons).
>
> Here too, each team, doing the right thing for themselves, ends up using the available testing resources inefficiently and harming the organization.

The Turnpike Effect

The Turnpike Effect (aka The Parking Lot effect) explains what happens when more capacity is introduced to a constrained system. The larger capacity ultimately results in increased volume of usage, up to the full capacity of the larger system [Gauss&Weinberg89] p.272-273.

Applied to a test machine pool, the Turnpike Effect predicts that any number of test machines added to an existing pool will fast become busy running new tests, added as a result of having more capacity.

Test is Automation

In some teams Test Automation becomes the holy grail. It is promoted as the most important effort; engineers working on test automation get exposure and recognition, while engineers whose main skill set is in creating good tests are not recognized enough.

Programming skills are nurtured; Testing skills are neglected. New members are added to the team based on their ability to write code and not their being good testers.

The lack of testing skills means a faster overload of the available test machine pool, as unskilled testers tend to create more tests than are needed, with little attention paid to test time. Making the risk decisions what tests can be removed or how they can be optimized is becoming a task that is beyond the abilities of the test team.

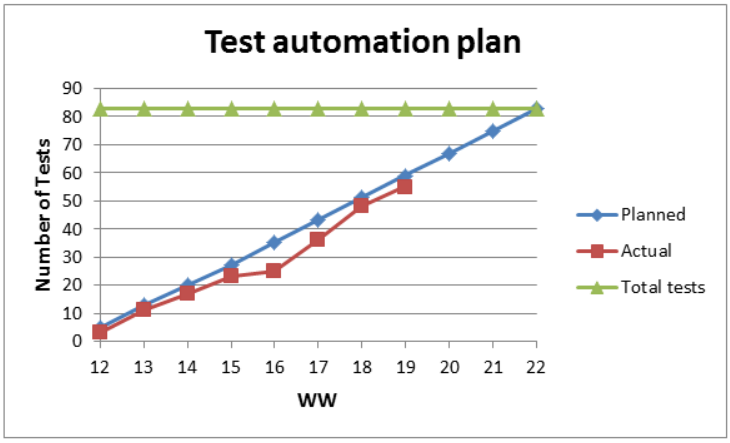
Added to the basic pattern of The Night Time Fallacy, these phenomena almost guarantee that the test machine pools will be overloaded beyond capacity and provide inferior testing services to the organization.

Pattern #4: Going for the Numbers

How do you track progress of an automation project?

A common practice is to define the number of tests that are to be automated, draw a timeline for when it will all be done, and track performance against schedule.

In reports, this will look something like the graph below:



A goal for the test team is to meet the schedule; maybe even get ahead of schedule. This looks good in progress reports. Slowdowns are very visible and put the automation team and the test manager on the spot for not meeting commitments.

Writing a robust automated test is not trivial. Note the "robust" attribute. Writing an automated test can be rather easy. It's the *robustness* that's hard to achieve. Robustness means the test is not sensitive to small fluctuations in system and environment response time; that it can deal gracefully when expected resources are not available; that it creates good logs, with the level of detail fitting the result (more for Fail, less for Pass); that it can run on multiple platforms and operating systems; that the code is well commented and easy to maintain, and more.

Achieving this is not simple at all. Sometimes, it means writing fail-tolerant library routines; sometimes it means adding features or major re-writes of the automation infrastructure. Even in the simplest cases it means new tests need to be run on many different machines to ensure they work on all configurations.

When progress is measured by "number of completed tests", there is a strong push to declare tests as Done once their coding is completed and push the robustness work till later. In some way it's synonym with developers declaring Done at Code Complete point. We all know they are far from done, since Testing is far from done.

Once the timeline is drawn and committed, it's hard to explain schedule slips by "we are slower than plan since we make sure the tests are robust". It sounds as credible as developers telling you "we are behind schedule since we are making an effort to write high quality code". You are expected to write robust tests, right?

The real issue is that it is hard to "sell" an automation project if the effort estimation includes the huge work needed to make the tests robust. Worse: even when testers do try to add this effort into the timeline, they many times miscalculate how much added work is needed before a test is robust. So when automation plans are presented for planning or for approval, there is a tendency to gloss over the huge effort needed for achieving robustness.

The end result is that many automated test suites are not robust and create a lot of noise: false fails, halt of the testing progress, machine time loss and mountains of useless logs. It can get to the point

that more time is wasted on analyzing and debugging test results than would have been invested in just running the tests manually.

Pattern #5: The Sorcerer's Apprentice Syndrome

Everyone, I assume, is familiar with Disney's Fantasia. The piece about the Sorcerer's Apprentice is probably the best known part.

Let's look at what happens in this scene with professional eyes:

Mickey is assigned with a repetitive, boring task. After doing it for a while, he figures out that this problem can be solved by Automation. He takes a broom, and quickly writes a script in his favorite programming language to make the broom execute the job automatically. The design of the automated system mimics exactly the actions Mickey would use to perform the same task. Two hands, two buckets, walk to the well, fill the buckets, walk to the destination pool, empty buckets.

This is a common occurrence in test automation: manual test cases are taken step by step, and each step is translated to code that performs the exact same action. And here lies the mistake.

Mimicking human actions sound straight forward, but is sometimes hard to accomplish programmatically and is frequently inefficient.

Consider the problem Mickey is faced with: "Transfer water from one location to another; The water well is at higher elevation than the destination pool". Mickey's solution is mechanically unstable and complex: tall, unbalanced structure; mechanical arms that move in a number of directions, and must support the weight of the water; ability to go up and down stairs. Ask a mechanical engineer – building this machine is pretty difficult; some magic would probably help.

Compare it to the trivial solution: a pipe and gravity.

However, arriving at this efficient solution means departing from the written test steps – which calls for an ability to distance oneself from the immediate task.

Many automation solutions – definitely those that start small and mushroom - suffer from this problem. The step-by-step conversion of a manual test to an automated one results in an inefficient, complex and brittle test automation system.

So… Are these patterns inevitable?

Is there anything we can do to avoid them?

Let me make a few suggestions.

While you can't always avoid the failure patterns, you can employ counter measures to mitigate the impact of the patterns or avoid them altogether.

The suggestions are listed for each of the patterns described above.

Mushrooming: Counter Measures

There are different things you should do, depending on what stage your automation is at. The suggestions are therefore presented per stage.

Stage 1 (Small and Local)

Actually… you don't really have a problem at this stage. A small, focused automation utility, that deals with a single feature, is usually very efficient. You **want** to have these utilities.

However, since no tester is guaranteed to stay in the same position for ever, it makes sense to protect yourself against the inevitable point in time when the tool owner will move on.

- Have some type of Code Control. This is important for two reasons:
  - Backup. You don't want the whole code to be lost the next time the hard-drive of your tester crashes.
  - Rollback capability, to avoid cases where the tester, who is probably not a seasoned code writer, will manage to mangle the code for the tool, and needs to start over from a stable code base.
- Have *some* documentation.
  - Good comments in the code. This will allow future maintenance (the first beneficiary from this will be the tool writer herself; people tend to over-estimate how well they remember things…). Also, this will allow other testers to take over the code in case the original creator moves on[4].
  - Some explanation what the tool does and how to use it. It can be as simple as a command line "help" that comes up when the tool is called with no arguments; it can be a README file; or it can be a real manual.

[4] Moshe Cohen, a colleague test manager, had a rule for reviewing test automation code, which he called "Green in the Code". The IDE we used colored comments in green. His first "test" when reviewing new test code, was to scroll fast through the code. If he did not see enough green, he would reject the code and ask for better documentation and comments.

Stage 2 (Generalization)

This is a very critical stage. Failing to react at this stage almost guarantees that your test automation will follow down the path of the Mushrooming pattern. This is the point where additions to the simple tool of stage 1 are usually done without proper consideration. Each additional capability adds just a little code and it seems redundant to stop and do some heavy-weight design. But that's exactly the point where design is most needed.
To quote Fred Brooks: "The hardest part of building a software system is deciding precisely what to build... No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later" [Brooks95].

If you don't architect the system now; if you don't decide, at this stage, what this automation tool will do and what it will NOT do – you will end up with a mess.

So:
- If you did not already do the Stage 1 counter measures, do them now
- Discuss and agree on your test automation strategy – at least as it relates to the tool in question
- Architect the tool. Define the mechanism by which new capabilities will be added. Think about scalability, code reuse etc. In short: Get a software architect to work on this problem and put some type of architecture in place.
- Since the tool is now serving more than one person, you must impose some "program management" rules:
    - Version control
    - Scope control; feature prioritization rules
    - Bugs & Requests database
    - Release management

Stage 3 (Staffing)

If your analysis shows you are at this stage, I advise you to call a total moratorium on writing ANY additional code for automation until you do the steps below. If you can't do that right now, because a new release is just about to go out and was committed already – set a date – as early as possible – and be clear that after that date no coding is done before the following happen:

- The counter measures for stages 1 and 2 are in place
- Management level discussion to define the automation project:
    - Define the scope. What do you want this system to be? Your main automation framework? A focused solution for specific goals?
    - If there are a number of teams involved, and they use different programming languages, make an executive decision on which language to use. If you can't make up your mind (each engineering team will have very good arguments why the language THEY use is best), just toss a coin. While in theory there is no problem with a framework where different parts use different programming languages, the reality is that maintenance will be hard and the learning curve for new engineers will be longer.

    - Take decisions on any other long-standing arguments between the teams.
- Define the release management (e.g. frequency; acceptance tests; logistics)
- Define code management rules (e.g. code reviews; pre-check-in tests; code repository housekeeping rules etc.)
- Ensure the automation developers have the needed skill-set to develop a large, industrial-strength application. If not – define training curriculum or bring in the needed talent.
- Define and start collecting two types of metrics:
    - Quality metrics of the Automation system: Treat the automation system as you would treat any software product and collect the usual metrics: acceptance-tests results; bug count; bug trends; number of times a release had to be pulled back; etc.
    - ROI metrics: Invested effort by type (new features; maintenance; refactoring; bug fix); Metrics that show how well you achieve the goals set for the automation system. For example, if your goal was to run the tests more often, then track the number of times tests were executed by the automation system. If your goal is better coverage, track the percent of the system that is covered by automated tests. Track the number of bugs found by the automation system (during automation development; while running regressions) but keep reasonable expectations[5].

Stage 4 (Non-Core features)

When you are at this stage, you need to ask yourself if your team's time is spent on the right things. Assuming there is something unique about your product or core technology, it is rather certain you won't find commercial tools, open source scripts or code that will test it. On the other hand, there are many existing solutions for generic test automation needs: test case management; test cycle creation and data collection, etc. (I call these "non-core [technology] features").

Where does it make sense to spend your engineers' time? Re-coding something that already exists, or coding something no one else will do for you?

The Build versus Buy argument is an old and ongoing one. There are many reasons why organizations end up building what could easily be obtained externally. For longer discussion of this topic see my article: The Home-grown Tools Syndrome.

If you want to check whether it makes sense for you to Build what you can Buy, try to answer the list of questions posed by Allen Eskelin in Buy VS Build.

Apart from this major decision, here are other actions to take as counter measures at this stage:

- Implement stage 1, 2 and 3 counter measures

[5] Generally speaking, don't expect to find many bugs while running regressions. It is my experience that test automation finds many bugs while the tests are developed, but not so many bugs when running as part of the regressions tests. See also: [Fewster&Graham] p.206: "The majority of the defects are found when the tests are designed and run for the first time (manually) rather than when they are automated" […] "the true value of automated tests is often in providing confidence rather than finding new defects."

- If you do decide to continue with non-core features development, the least you should do is re-architect your automation system so it has:
    - Low coupling between core and non-core features. This way, if later in time you realize the development of non-core features was a waste of time, you can relatively easily replace this functionality with an external solution.
    - Solid infrastructure: support multiple technologies and platforms; efficient support for off-shore teams (network band-width; efficient database access); database mirroring; efficient build system… all the things that an industry-strength commercial system is designed for, and you won't have if you don't architect for it.

  As a side note: once you realize what it takes to achieve the above minimum goals, you may want to re-think your decision to write the system yourself…

Stage 5 (Overload)

If you reached this stage, you already know that you need to re-design the system from the ground up. The fact you know it, does not mean you can just plunge ahead… the daily realities may mean a redesign is not going to happen soon due to resource issues, project priorities etc. Even if you can start re-doing the system, it is going to be a significant effort and take significant time. You need something to tide you over while this is going on in the background. Here are some counter measures you can take:

- Give up problematic areas. Check If most of your pain (false fails, system halts and broken tests) is related to a relatively small number of features. If it is, consider going back to manual testing of these areas, until the system is re-designed.
  Another approach to deal with problematic areas is to build "Stage 1" tools to test them, and run these simplified tools outside the main automation framework.

- Stop adding features. Invest all your time in refactoring code and redesigning problematic tests. To paraphrase, take the position of "We value Robustness over New Features"

These measures will give you breathing space to start a re-design effort of your system. If you follow some of the advice given here, you know that you need to get a software architect and design the system from the ground up so it is scalable, robust and integrated with an external Test Management tool.

The Competition: Counter Measures

I mentioned three ways in which competition may arise in a large organization and each flavor should be addressed differently.

Competition (1st variety): This is the case where a number of teams developed independently an automation framework and can't agree on how to merge the competing solutions.

I have encountered this situation a number of times and witnessed the futility of trying to arrive at a "peaceful agreement" in which the teams merge their tools to a single system. There is just too much professional pride, inertia and emotions involved.

My conclusion is that when faced with such a situation, the only way to deal with it is to remove the arguments off the table by making a clear and uncompromising decision.

It can be a wholesale adoption of one of the systems. It can be a cancellation of any additional development of any of the existing systems, merging the involved engineers under a single manager, and developing a NEW system.

A single person must be given the authority to make binding decisions on anything that has to do with the system: scope, programming language, code repository system, architecture, etc. Nothing short of such drastic effort will solve the problem. As long as engineers feel that if they only come with the right argument their position will prevail, they will look for such arguments. Once they realize that management made a decision and is not interested anymore in the argument, most engineers will move on and do what now needs to be done.

Indeed, following this advice does mean that some of your engineers will be too frustrated to continue and will prefer to move out of the team. I believe most of these situations can be avoided if engineers' feel respected and that their anguish is not ignored. If you are about to make a decision they won't like, make it clear – in 1/1 meetings – that you value their inputs and their work, but the situation has to be resolved and this time it will be resolved differently to what they had hoped for. But it's not because their views were ignored or their work unappreciated; it is because there are equally strong, justifiable but opposing opinions.

And be ready to bite the bullet and lose some talent.

Competition (2nd variety): This is the case in which some teams start developing a parallel automation solution because their requirements to the existing system are constantly de-prioritized.

You can't solve this with better prioritization. There will always be some requirements that get pushed out,and the team who needs this functionality will eventually look to solve the problem outside the existing system.

The one solution I am aware of for this problem is to design the automation system with plug-in capabilities. When a system has this capability, all teams can get what they need: they just develop what they need by themselves. The benefit is that all these small efforts are controlled within one large framework; can be re-used by other teams and conform to some development and quality standards.

Looking at examples such as Jenkins and Firefox, each of which have hundreds or even thousands of plugins, you get some confidence that this approach is a workable one. It does mean that when architecting your system (at Stage 2, hopefully), you mandate a plug-in architecture.

Competition (3rd variety): In this version, teams get disillusioned with a "Stage 5" automation system, and start their own automation initiatives.

As long as these are Stage 1 tools – accept this behavior. If the new tools allow you to stop using some of the more problematic parts of the main automation system, it's a good interim solution and will give you some breathing space to fix the main system.

However, be very vigilant. Don't let the tools mushroom. If possible, define some minimal coding standards, so that these tools can later all be used as plugins in a new system.

And yes, you need to fix the main problem… redesign the current system!

The Night Run Fallacy: Counter Measures

Dealing with this pattern calls for a number of actions.

First, you should invest resources and efforts in improving your testers' testing skills. Knowledge of testing principles and techniques will increase the testers' confidence when making risk and tradeoff decisions such as reduction of test time or smarter selection of tests cases. The level of attention to testing skills needs to be at least equal to the attention given to coding skills. Automation needs to take its proper place in the skill hierarchy: automation serves testing; it's not a goal by itself.

A second action that is critical to take, is to have a direct link between efficient test time and teams' performance goals. The easiest (and an action that has immediate results) is to allocate machines to teams. This can be done physically or by test-machine allocation software.

Once done, the three conditions for a Breakthrough System [Daniels95, p.29] are in place:

- The team has a clear understanding of the results expected from their work
- The team has an immediate and valid feedback about current results versus expected results
- The team has control of all the resources needed to meet the expectations

The first two conditions are either already in place as part of existing project-level reports or can be created relatively easy with existing test results data. It's the improved testing skills and the test machine allocation that makes the third condition possible. Once the team's testing skills are improved, tests can be optimized successfully. Once the team owns the test machines, any action that improves test efficiency immediately shows in the reports as the team's success.

My experience shows that once these actions are taken, you will see a fast improvement in test efficiency.

Going for the Numbers: Counter Measures

There isn't really a simple or smart way around this problem. In order to get robust automated tests you must invest in quality assurance of the tests and the test code.

What you should do though, is systemize this work so you can (a) know what you are aiming for and what you get when a test is declared Done and (b) you can start collecting metrics on the time and effort the robustness work takes, which is the first step to optimizing this activity.

For one of the automation projects I was involved with, I developed a detailed "Test Acceptance checklist" which listed everything that needs to be checked on a new test. There were ~30 items to check, starting with documentation, and ending with running the test on all target operating systems. Each line by itself made a lot of sense. The overall result was 2-3 days' work… It was good on paper, but useless in reality.

The checklist needs to be trimmed down, to the most critical elements – and this may change according to the context. But you need to have such a checklist and a test is not Done until the checklist is completed. The time spent on this activity will in most cases be saved later down the road, with tests that always run to completion and whose results are trustworthy.

The Sorcerer's Apprentice Syndrome: Counter Measures

As already eluded to in the pattern description, the key to avoid this problem is to reassess all of the tests before starting the automating work. Automating each test case by itself, step by step, means you give up on more efficient solutions.

In many cases, the considerations and constraints of manual tests and very different to those of automated tests. For example, a test with a lot of steps may be a problem for manual testing (tester fatigue; high chance for making an error) but for an automated system, lots of steps is not a problem at all - the computer doesn't mind… (I do assume here that there is a good reason to have all these steps in the first place). Tests that call for exact timing or for accurate, reproducible movement – are possible with automated tests. On the other hand, some actions are hard to do automatically, such as certain image processing actions (think how hard it is to automate Captcha decoding – which is why it is such a good human-identifying mechanism).

Before starting to code, you should take a step back. Try to forget the list of tests you have and re-think the test strategy and test design steps. Look at the whole testing challenge and evaluate how you'd test it with automation.

You may end up with more or less the same test list; but you may end up with something completely different.

While you do this evaluation, you will probably identify some actions that will need to be done many times. This in turn will influence the test automation architecture. Keyword Driven Testing (KDT) may be in order; or at least some common libraries that will be shared among tests of different features. By looking at the complete test challenge (even if you only plan to do partial automation right now), the resulting system will be more efficient, scalable and maintainable.

Summary

A lot of things can go wrong on the way to efficient test automation. Apart from the technical difficulties[6], this paper outlined a number of non-technical failure patterns that impact the ability to deliver high quality automation systems. These patterns are highly related to human and organizational behaviors and are therefore hard to prevent. Recognizing the patterns as they appear and taking a number of practical steps, can avoid or at least significantly mitigate their impact.

[6] A good source covering many of the technical difficulties of test automation, as well as offering advice, is Test Automation Patterns Wiki (https://testautomationpatterns.wikispaces.com) created by Seretta Gamba, in collaboration with Dorothy Graham and contributions by many others.

Acknowledgements

I noted the mushrooming pattern a few years ago. I wanted to know if this is something that is prevalent in the industry, and discussed it with Alon Linetzki (www.best-testing.com). Alon, besides being a good friend and very knowledgeable in software testing, is also leading the activities of SIGiST Isreal. Alon suggested we hold a SIGiST meeting and have an open discussion with attendees from different companies. We will present the idea, see if others experienced it, and discuss possible counter measures.

The clear feedback was that the pattern appears in many companies. We also got a solid list of suggestions what steps to take at each stage. This resulted in a paper we presented at the annual SIGiST Israel conference (2011).

Later, I added other patterns and created a paper which I presented internally at my company and then at STAREast 2013. The copyright for the "Mushrooming" term goes to Lee Copeland, who suggested it as part of his review of my paper.

Dorothy Graham, who attended my talk, encouraged me to write an article covering the ideas presented – and this is how this eBook came to life. Dorothy also spent time reviewing this document and made it better both in terms of content and in terms of English, which is not my native language. Thanks!

References

[Brooks95] The Mythical Man-month (2nd edition), Fredrick P. Brooks, Addison Wesley Longman Inc. 1995

[Daniels95] Breakthrough Performance: Managing for Speed and Flexibility, William R. Daniels, ACT Publishing 1995

[Fewster&Graham99] Software Test Automation, Marks Fewster & Dorothy Graham, Addison-Wesley ACM Press 1999.

[Gauss&Weinberg89] Exploring Requirements: Quality before Design, Donald D. Gauss & Gerald M. Weinberg, Dorset House Publishing, 1989.

Web resources

http://testautomationpatterns.wikispaces.com/ . The purpose of this wiki is to share information, ideas and experiences about test automation, structured around the notions of Test Automation Issues and Test Automation Patterns

promotion

EuroSTAR Software Testing CONFERENCE | copenhagen 06-09 NOVEMBER 2017

PRE -LAUNCH SPECIAL

This year we're celebrating our 25th annual EuroSTAR Conference and we want you to be there. To help make that happen, we're giving you 20% off until April 19th!

get discount code

EuroSTAR Software Testing

want more?

Enjoyed this eBook and want to read more?

Check out our extensive eBook library on Huddle.

EuroSTAR Software Testing | Huddle

Join us online at the links below




EuroSTAR Software Testing

www.eurostarsoftwaretesting.com