

Automation Selection Criteria – Picking the “Right” Candidates

So there you are. You’ve done a little research and made the business case to upper management regarding test automation and they bit on the proposal. Surprisingly, they supported you all the way and are extremely excited about how much faster testing can really move...

Or

Upper management comes to you with an edict to start automating your testing. You’ve got to improve testing cycle and turnaround times and do it with fewer resources. They clearly believe automation is the only way to achieve these goals. You’re given a nearly blank check to quickly bring automation into play for the next major release—3 months away...

In either case, you’re in the enviable position to be poised to begin developing test automation. Automation can provide a tremendous boost to you team in technical challenge & contribution while wowing the business folks in driving testing cycle times down and coverage up. Frankly though, this can also be an intimidating time—especially if this is your first time trying to sort out where and how to begin, which is exactly the focus of this article.

This is the second installment in a series of articles targeted towards improving your management and setup of testing automation. The first was focused towards the Automation SDLC and in the next one we’ll explore developing a solid Business Case.

One of the basic assumptions I’m making for this article is that you’ve been creating test cases and manually testing as an organization for a while. That is, you’ve built up some sort of repository of manual test cases that are potential automation “candidates”. Given that you can’t automate everything at once, the question of where to start and how to properly orchestrate your efforts over time becomes a challenge.

I’m also assuming that you do not have infinite resources nor time to produce visible results. That is, you have other testing responsibilities besides the automation, for example testing and releasing your products. So prioritization and establishing a work balance becomes a challenge as well.

We’re going to examine three key areas to help you craft a strategy to meet these challenges:

- 1) First I’ll explore a few common anti-patterns that impede good test case selection.
- 2) Then we examine a solid set of good practice patterns for test case selection.
- 3) Finally, we’ll wrap up with prioritization and criteria adjustment factors so that you can truly be nimble over time.

Common Anti-Patterns for “Starting” Automation

Anti-patterns have been applied to software design, coding, configuration management, and just about any activity within software projects. I think exposing what *not to do* is useful in setting the stage for some of the recommendations I’ll be making later on, so here are a few general anti-patterns for selecting good test candidates for automation development.

We Don't Need No Stinkin' Criteria

It simply amazes me how many groups simply dive into automation development without a strategy surrounding what test cases to automate. They simply start automating somewhere within their pool of test cases, often picking an arbitrary starting point such as first character in the tests name, and then moving serially forward from that point. Another part of the No Criteria anti-pattern is never *reevaluating* your lack of criteria as you make forward progress.

A Good Start – But What's Next

I also see teams who start well, for example, picking a set of “low hanging fruit” automation candidates that make sense to initiate the automation program. Typically the set is small, intended to get the effort going quickly and to realize some short term successes. However, after the team accomplishes their initial plans, they fall back into a no criteria, select anything you want mentality towards automation.

By George, Let's Do It All

Another frequent anti-pattern is the view that all test cases need to be automated. This creates churn because the team is frenetically trying to do it all. Frequently working in parallel with a mainline software project and struggling to automate test cases on a moving functional target. It also drives the mentality that all test cases need to be automated independent of the level of effort or returned value associated with that endeavor. Which is simply not business savvy nor realistic.

In Tools We Trust

This anti-pattern is focused towards the myriad of automation tools available. Often they can lull the team into this false sense of security that the tool will take care of the details surrounding automation—thus removing the need to understand the tools and technologies being used. It also masks the teams from understanding the strengths and weaknesses of each tool as it relates to different technologies. And trust me every tool has “problems” that need to be considered.

Make A List, Then Stick To It!

In this final anti-pattern, teams do everything well at the beginning. They take an inventory of their test cases and pull together a thoughtful ordering that starts building good automation and contributes positively to their projects. However, they become stuck in their initial list and never adjust it for discoveries and changes as they progress. As time moves on, their original approach becomes more and more irrelevant.

Clearly these anti-patterns set the stage for the remainder of the article—changing the focus towards what to do regarding selecting the best candidates for automation and developing your overall implementation strategy.

Selection Criteria – Patterns

Taking a key from the anti-patterns just mentioned, there are some general selection patterns that I normally use to govern my thinking around selecting good test cases. We'll examine these practices with the view towards establishing them within your own automation efforts.

Low Hanging Fruit

As I alluded to in the anti-patterns section, this pattern is focused on generating momentum and not in picking the best, most impact producing automation targets. It usually happens in the beginning of an automation effort or when you've received a major new version of your

automation tools. In any event, low hanging fruit test cases are just that – they’re usually small, localized to well understood and stable functional components and generally quick to implement.

I’ll usually focus new automation developers on these cases for learning curve and to measure their capabilities before assigning them more challenging work. A final important part of the pattern is making your successes visible—both internally within your testing team and across your organization. So showing early success off a bit is an important part of the pattern.

Fleshing Out The Infrastructure

All good automation efforts establish an infrastructural layer that wraps their tools in order to support efficient automation development. Usually this is focused towards implementing automation development templates, standards, naming conventions and other guidelines that enforce consistent practices and procedures for development. It often also involves writing some wrapper code that programmatically reinforces the consistency and serves to reduce startup time and learning curve.

These practices need to be verified and debugged, so selecting good candidates to flesh out and verify the infrastructure becomes important. Usually these candidates are distributed along the architectural boundaries of the Application Under Test (AUT)—so that the infrastructure is verified to support all aspects of the application environment.

Minimizing Rework

There are clearly two factors that can drastically effect your efforts and influence rework— requirement stability and application stability. While you can’t always control either, you can choose when to automate and thus mitigate impacts from the two. In my experience, the easier to control is requirement stability. In this case, you simply want to wait until there is sign-off or agreement on requirements. Another indicator here is that the development team is beginning construction. Normally this is a good indication that if the requirements are stable enough for development they might be stable enough for automaton.

Application stability is a tougher factor to control. I usually like to trigger automation development until after we’ve at least had a chance to exercise applications features manually. If manual execution is impossible, for example with an API, than any demonstrated testing and positive results will serve as a trigger for automation development.

Of course a good strategy in both of these cases is to defer your automation construction until after the specific application release is deployed. In this case, you take both aspects virtually out of play and drastically reduce rework risk.

Driving with Value

At a fundamental level, automation value is driven by the coverage it provides contrasted against the time it takes to automate it versus the time saved by the automation. The potential lifetime of the automation then is a strong factor in determining its value. I see many teams who write automation for very transient application features. They’ll run it only a few times and then the application morphs in another direction.

They’re stuck either taking the maintenance hit to change the automation or retiring it. In both cases, the value of that particular piece of automation was impacted. Value is also determined by the customer value associated with the specific feature. How central is it to their business needs and frequency of use?

You should think of the return potential for every automation decision. More times than not, you're looking to automation capabilities that are stable—so that the automation has a high lifetime before maintenance is required. These features also need to be of high customer value—leading towards heavy interest in repeated testing cycles.

Planning – Consider Complexity & Packaging Tempo

One of the more important factors to consider is simply how hard things are to implement. I wish more product development teams would consider this when developing their products. If you have 100 test cases that you want to automate and they're all exceedingly complex, then you're somewhat stacking the deck against yourself for delivery.

What I prefer to do is create a more normally distributed set of test cases, say 25% very complex, 50% moderately complex, and 25% relatively straightforward, when developing an automation release package. I'll also create a goal for the package that represents one of the overall patterns in this section. For example, after my first automation *Low Hanging Fruit* package development I'll often go for a cycle time or speed focused goal of *Project Impact – Consider Speed* release. This way I immediately start leveraging automation impact on the project. Regardless of the approach, you should develop a packaging strategy that uses goals and short iterations to create a tempo for your automation release.

You also never want to underestimate hand-off considerations as the story in Sidebar #2 illustrates.

Project Impact – Consider Speed

As testers we always want to maximize the impact we have on our projects, which usually focused on risk mitigation. Automation also has the capacity to drastically impact testing speed—so examining how quickly a package of automation can be pulled together for product execution becomes a very compelling view. In this case, identifying test cases that take the longest time to execute manually and automating them can have a drastic effect on testing cycle time.

Of course you need to consider all aspects of time when picking these candidates, including test setup time, data preparation time, execution time, and results analysis time. Clearly you don't want to automate the smallest or fastest tests first if you want to have a high impact on cycle times within your project.

Project Impact – Consider Risk

I've found that one of the most powerful applications of automation is as a risk mitigation factor within your projects. From that perspective, you may want to choose candidates that align with some of the more challenging areas of the application. Areas where you know the development team will struggle and where you can make a difference. This may mean that you take less on—in order to have a larger impact on the products' maturity stabilization.

One key practice to consider here is the Pareto Principle or 80:20 rule. Basically it states that 80% of the bugs (or risk) will be driven by 20% of the overall application. If you can identify and isolate these 20% areas and apply your automation efforts towards them, you'll be well on your way towards effectively mitigating project risk. And your developers and project managers will love you for it!

Project Impact – Consider Maintenance Effort

One of the truly hidden properties behind developing test automation is that it is indeed a software project. Given that, it also succumbs to maintenance issues both in the short term, as you

try and automate a volatile application, and long term as the application evolves and matures. I've found that nearly 20-30% of my automation development time is spent in handling traditional maintenance level activities. And this was, at least from my point of view, using well architected automation solutions so it can potentially get much worse than this.

One way to handle maintenance work is to steal a technique from the development lifecycle and have periodic maintenance releases of automation. I'll accrue maintenance work for these sorts of releases and time them to coincide with possible slack time within my projects. This also allows me to better understand, isolate, measure and communicate the costs as well.

Clearly you don't need to consider all of these patterns in every selection decision. And they all have different weights depending upon your specific context. The primary goal is to create a breadth to your analysis and consideration that truly targets test case selection towards making a visible and fundamental impact for each of your projects. It should clearly be seen as a differentiator when the team discusses project success factors. Sidebar #3 serves as a decision-making checklist for establishing this broader view.

Changing Your Selection Criteria

One of the challenges associated with selection criteria is that they don't remain constant, or at least they shouldn't. All sorts of conditions change with your automation scheme and projects that should cause you to pause, reflect, and adjust your criteria. Here I'll simply list a few of the primary conditions that usually cause me to reflect and potentially change selection criteria. While I'm certain there are others, these will certainly illustrate the point for criteria adjustment.

- 1) **Changes in Skill Set** – Either positive or negative can certainly change your criteria. If your capabilities increase, you can and should take on more with each project and automation iteration. If they decrease, then you must take on less, which places more importance on what you select.

There are many factors that can influence this, attrition, outsourcing, changing project priorities, and changing tool sets come to mind as frequent drivers. The real point is to closely monitor your performance capabilities and adjust as they do.

- 2) **Changes in Application Technologies** – Quite often developers adopt new technologies that clearly disrupt ongoing automation efforts. If you're lucky, you get an early warning so you can experiment to ascertain the impact prior to product release. However, most of the time you'll become aware of automation implications at the point of implementation.

If I anticipate technology changes that will impact your automation tools or strategies, I often plan for a small, impact evaluation iteration. Within it I look to fully understand the impact of the changes, evaluate the impact to my tools and infrastructure, and carefully estimate the change impact to my existing set of automation. Once I have a good feel for the *full impact*, I'll adjust my plans and criteria as required.

- 3) **Changes in Team Balance** – I often like to think in terms of “producers” (developers) and “consumers” (testers) when planning my testing and automation efforts. That's why developer to tester ratios are so important for capacity and workflow planning. If you get a

drastic change in your ratio, then it will certainly have an effect on your automation capacity. Again, as in the case for Changing Skill Set, I want to reevaluate my prioritization to ensure that I'm selecting the more relevant candidates.

If you don't have a dedicated automation team this becomes an ongoing challenge as you multi-task and reallocate automation resources towards other project-driven testing activity.

- 4) **Ongoing Setup Time** – The time to setup and teardown your automation run-time environment can be a significant factor in how you select automation candidates. This becomes particularly important in some database applications that require vast amounts of time sensitive data in order to physically setup and run the automation. Or any automation candidates that simply requires exorbitant time to establish an initial state.

I usually prefer to attack time sensitive test candidates quite early in my automation efforts as a risk mitigation strategy. Often, I don't understand the full implications of the setup nor the impact it will have on my scheduling other, much shorter, automation.

- 5) **Evolution of Maintenance Costs** – I've found that automation, depending upon how it's architected, can be particularly sensitive to maintenance costs. Part of this burden is naturally driven when you try and implement too early—before interfaces or functionally within the application have stabilized. Another aspect is internally driven, for example when you make infrastructural changes or upgrade versions of your automation tool-set.

One reality of maintenance costs is that you need to pay attention to detect them and address them immediately. However, there is good news in that they are frequently cyclical in nature so that once they are addressed you can fallback to your normal selection scheme.

- 6) **Evolution of Development Methodology** – Clearly if your organizations SDLC or methods change it will probably impact your automation efforts—see Sidebar #1 for an overview of the factors. The most drastic case of this resides in the Agile Methodologies. In these cases, you'll want to shift your automation focus towards each of the development iterations, while helping to automate the unit and acceptance testing activity within the context of each.

In addition, you'll want to target automating candidates as they “exit” the iteration. However, there can be tremendous volatility across the feature and interface set, so you'll want to carefully consider the stability of each feature or candidate before implementation.

- 7) **Changing Business Conditions** – The can take two primary perspectives, budget & investment or changing business value and requirements. Often it has the same effect as changing skill sets or technology in that it causes you to change your view to automation selection either positively or negatively.
- 8) **Retirement Consideration** – Just as software incurs technical debt over time, so do your automation efforts as test cases can lose their relevancy and connection to the current application. Or what was a high priority candidate is now something that needs to be executed infrequently or not at all. As part of my changing selection criteria planning, I usually drive reevaluation of automation from a retirement perspective.

Retirement can imply, just that, a removal of an automated test case from the execution environment or a reduction in the frequency of execution. In either case, you should formally note the change and, if removing it, move it to a retired automation archive.

My regular interval for reevaluating my automation environment and selection criteria is either on a quarterly basis or before starting a significant new project. That way I cyclically evaluate the landscape and consider any important changes that might influence an adjustment of my overall selection criteria.

Wrap-up

The overall focus of this article was to influence your planning and strategies surrounding the selection of automation candidates. A few of my early reviewers reacted badly to all of the factors—concerned that considering all of them would bog down the automation process with overbearing planning and simply take too long.

However, my experience is just the opposite. I've found that when I develop a selection criteria that maps to my overall automation strategy that I actually develop *more* automation. It also has greater impact on the business and projects, while increasing the visibility of my testing team in the overall product development lifecycle.

One final point, don't ever think that your automation efforts are done. It's an ongoing challenge and I think you need to stay on top of new tools, techniques, maintenance, products, etc. Viewing it as a short term exercise with a succinct end point is certainly not the right mental model.

This has been the second article in my automation management series. In the next and final installment, I'm going to explore Automation Business Case development.

Sidebar #1 – Methodology Implications for Automation Selection

Methodology	Automation Selection & Construction Implications
Waterfall	<p>While the waterfall method has been challenged as non-agile and inefficient, its need to define requirements early offers benefits to automation, including the ability to select candidates earlier in the SDLC. Waterfall also makes selection more global in nature thus permitting more multifaceted decision-making.</p> <p>One down side is frequently the need to hold automation until a subsequent release before being able to positively impact the product with automation.</p>
RUP – Rational Unified Process	<p>Generally RUP is front-loaded with analysis and iterates through construction. This means that you should be able to gain a firm handle on automation candidates roughly 1/3 of the way through the SDLC.</p> <p>However, implementation will truly iterative in nature. It's best to connect the automation strategy to the development construction strategy, with a focus on early construction iteration speed, risk or coverage goals.</p>
Agile Methods	<p>Agile Methods truly disrupt traditional automation methods and approaches. The first disruption point is the focus on up-front automated acceptance testing. Typically the tools used for this fall outside of traditional test automation tools and preclude any sort of reasonable reuse for other sorts of automation.</p> <p>The finely grained iterations make it difficult to focus on more than one specific goal or criteria per automation release. It helps to mirror automation releases with their development counterparts, even though they may be skewed by an iteration.</p>

Sidebar #2 – A Story of Hand-off Considerations

I was once working on an automation effort where we did a solid job of considering what to automate, how much and how to deliver the automation code. I was quite proud of the development effort and felt that we were really doing well.

However, after a few automation development cycles I asked how the automation was performing within the regression cycles. It turned out that only about 50% of what was produced was actually running in the regression cycle and I was quite surprised by this.

After some digging I realized that the root problem was our poor hand-off of the automation to the test execution team that was chartered with running the automation. In too many cases, they were having problems with it—from struggling with setup, to physical execution, to figuring out how to analyze failures.

It was then that I realized how important deployment planning was to any automation effort. Not only should this cover training and the physical handoff, but it should also include usability as a design goal. Clearly, automation that is *on the shelf* is doing you no good at all.

Sidebar #3, Automation Prioritization – Weighing the Factors

Once you have started the effort and are comfortable with your techniques, you should examine selection from a wider lens.

This prioritization step takes time and experience to achieve, so consider these factors before selecting candidates.

Selection Factor	Considerations	Weight for New / Mature Apps
Project Impact	Will this automation directly help your projects speed, coverage, risk, and/or development nimbleness?	L / H
Complexity	Is the automation effort simple to implement, including data and other environmental challenges?	H / H
Level of Effort	How much time is required to implement the automation, including time spent on packaging with other work?	L / H
Early Requirement & Code Stability	Will the application requirements or early coding prove stable within the scope of the driving project?	L / M
Ongoing Maintenance	How volatile will this code be in the longer term, will the functionality change/evolve? How about the toolset?	L / M
Coverage	How broad will be the coverage impact in features and functionality exercised, of this automation? Will it cover critical features?	H / M
Resources	Does your team have the human, physical and data resource required to run the tests?	H / H
Hand-off Efforts	Do test execution resources have sufficient skill and time to run the automation?	L / H

Automation decisions also can vary depending on the stage of the application's lifecycle. It's helpful to divide selection criteria into 3 primary categories. 1) Automation targeted for new or start-up efforts; 2) Automation for ongoing efforts; and 3) Automation for mature or end-of-life efforts. The checklist also might evolve as your automation efforts mature.

For example, I've assigned sample weightings for new versus mature applications of Low, Moderate, and High for consideration. For new application automation, the Complexity, Coverage, and Resource areas are the ones that should be driving the priority of your automation decisions. Once you focus in on what's most important, you can rank potential candidates from 1-5 in each area and arithmetically select your candidates.