Establishing your Automation Development Lifecycle

Frequently I engage clients in assessing and improving their automation efforts. The discussion normally starts from a position of frustration –

> *We've invested heavily in automation. We've purchased the tools and the training. We've focused one tester fulltime on automation development and allocated 20% of everyone else's time for it as well. Three months into things and we're struggling to gain any automation traction. I don't see any improvement in our overall testing cycle times. In fact, testing seems to be taking* ***longer****.*
>
> *What's wrong? Do I have the wrong tool, wrong team, wrong approach, what? I need automation to be working…now!*

My initial response to these circumstances is the same. It revolves around explaining that automation is a software development effort. In fact, that's EXACTLY what it is. I explain that it takes time to start most worthwhile automation projects and that many organizations struggle in the beginning at properly setting the stage.

Another important aspect of the challenge is that all automation efforts have to *integrate* with a development project—so resource and schedules are often shared. So instead of having a stand-alone project, you have two interrelated development projects running in parallel, with automation understandably taking the *priority back seat* to its development counterpart.

The intent of this article is to help you "connect" your automation efforts to traditional SDLC activities. While some test teams are getting better at it, I still see far too many clients that manage their automation outside of good software development practices. I'd like to see that trend change much more aggressively. We'll cover four key points in the article including –

1) Establish the major drivers for creating an Automation SDLC
2) Explore a few of the key success criteria behind a solid Automation SDLC effort
3) Review Automation SDLC extensions from your own product SDLC
4) Finally, consider how to integrate automation correctly with your mainline development efforts

I should also mention that this is the first in a three part series focused towards critical automation start-up management activities. The series will provide guidance regarding the SDLC, share strategies for forming the right selection criteria, and show you how to develop a coherent business case so that you can better manage and avoid the frustrations illustrated above.

Drivers for the Automation SDLC

There are three key drivers behind the need for an Automation SDLC. First, it helps to align your automation development methods with your software development methods. This will improve the *understanding* the various teams, managers, project managers, and most importantly, senior executives have of your automation efforts. Essentially it will level the playing field for understanding the goals, challenges and how automation *fits* within the overall product development lifecycle.

The second driver is establishing a framework that illustrates the processes, phasing, and release cycles associated with proper automation development. This will enable improved planning and quality within the automation effort. Again, I often see test developers that simply “dive in” and begin automating test cases without a thought towards architecture or strategy and we need to stop doing that.

Having an Automation SDLC also serves as a vehicle to synchronize with the Development SDLC. Since the phases and efforts will obviously be tightly coupled, it helps to have things well defined so that interdependencies and integration points between the two life-cycles are more easily understood. While this is more of a planning level point, it also helps with tactical decisions, for example, determining feature readiness for subsequent automation development.

Automation SDLC – Key Success Criteria

There are several considerations when defining and connecting your automation efforts to an SDLC. While this section isn’t intended to be exhaustive, it will illustrate automation concerns that are quite similar to those encountered by software developers building great products. There are key success factors that you ignore at your own risk. Most of these aren’t a day-one problem and you can often skip many of them in the short term and gain a sense of progress and contribution. However, over the long term, a successful Automation SDLC must establish sufficient context surrounding these foundational points.

Business Case
The first thing to establish is the high level business case for your automation efforts. Part of this effort is establishing clear goals that link to the stakeholder and business expectations. In a follow-up article I’ll spend a lot more time on the components of a solid business case. Here we just focus on goals, from the cost, time and scope perspectives. That is, how much automation coverage (scope) is required, within what sort of timeframe and supporting what sort of project delivery targets (time) and limited by what budgetary (cost) factors.

Another way of looking at the business case is developing a Charter for your automation project. The primary purpose of the charter or business case is to ensure everyone is on the same page as to what the automation effort will actually do for the project or organization and equally important what it will not do. This is the first step to combating the confusion I often see in key stakeholders who fundamentally do not understand the value proposition, challenges, and most importantly reoccurring costs associated with a solid automation program.

Architecture
Another important connection is towards architecture. While today’s testing teams seem to be gaining in the ability to define proper automation framework architectures, its still one of the larger challenges for most. Many still focus on creating automated scripts without thinking about defining an architecture that will support their development efforts longer term. Even if you purchase off-the-shelf tools, you still need to wrap them with a thoughtful model to support your efforts.

Beyond architecture, you also need to agree on the appropriate automated testing model for your product domain, skill set and requirements. Some of the leading approaches seem to fall into the following three categories:

1. Simple tool driven record & playback
2. Data, Keyword, or Action-word driven
3. Structured, modular, or programmatic engine based

As you go down this list, the complexity and initial development effort goes up, while ongoing maintenance effort goes down. The more complex models also require much more development-centric test engineers to build the infrastructure.

Sidebar #1 defines the elements that I think of for a proper defined architecture. However, consider it a thinking tool rather than an exhaustive checklist.

Requirements
As testers, one of our greatest challenges surrounds the requirements gathering and management processes. You've all been there. Requirements are late in development, slow to solidify, and quick to change—often very late in the game. All the while code is quickly being written. It's one of the ongoing battles in most software projects.

Therefore, we all have relevant experience with the need for requirements and the problems associated with their lack. Don't fall into this same trap when developing your test automation architectural framework and automated test cases. Take the time to develop a solid set of requirements. Ensure that you include key development and business stakeholders in this process so you collaboratively construct the automation capabilities vision. Who knows, your example may even become a model for your development team partners to copy.

Implementation Strategy
Rarely is it a good idea to start with test case #1 and automate through test case #10,000. Instead you should come up with an automation implementation strategy that guides you towards your goals as established in your business case and requirements. Often the early strategy is simply learning and establishing your automation framework. Rarely is there much of a focus on return on investment at this stage. Instead you are simply experimenting to find the right automation tools and techniques that will work within your product domain and project culture.

However, you need to quickly redirect this strategy towards more traditional automation goals. For example increasing coverage, find defects faster and improving overall production product quality. In the next article in this series, I intend exploring test case selection criteria in much more detail, which essentially focuses on developing these strategies.

Project Synchronization
This final success criteria extends from your product development SDLC. Clearly your automation efforts have to be effectively *extended* from the base product SDLC. While you can and should extend this into your Automation SDLC, which is the topic of the next section, you really need to understand the nuance, both technically and from a business and project management perspective of each project you're attempting to automate.

Extending from Product SDLC to Automation SDLC

Surely we don't necessarily want to reinvent our own life-cycle for automation unless we absolutely have to. One reason for that is that automation is inextricably linked to your

development methodology. For example, if you're using one of the Agile methodologies for product development, say Extreme Programming, then your automation focus will be heavily influenced by its short iterations and acceptance testing demands.

Another reason is simply to avoid confusion. Having separately defined methods will often require ongoing explanation across your project teams. It will also require you to define some sort of integration view between the two methods. And it will drive your Project Managers crazy as they try to schedule across the two different perspectives.

Instead, I prefer to *extend* the existing SDLC methods to include considerations for automation development. This isn't too onerous because they typically already include testing activity anyway. In this section I want to discuss a few of the more critical parts of the extension model for automation – thus creating a group understanding of the Automation SDLC.

Extension #1, Make Automation Decisions from *Completed* Manual Test Cases

Don't change your manual test case development process for automation. In fact, you might even want to tighten it up a bit. Only when the manual cases have been developed, executed, and proven to be correct, should you analyze the need to automate and schedule them for automation development. There are several reasons for this. First, it helps you to have a consistent format and workflow for developing all of your test cases. It ensures that the test cases are essentially "complete" before worrying about automation. It also allows you to queue up a set of well defined work items that can be rearranged as priorities and needs change.

Consider manual test case development the *early design phase* for your automated test cases. It will also sort out many of your requirements clarity issues before you ever start coding. Finally, if there are parts of features that are not functional or are problematic, at least you'll be aware of them in advance of automation development.

Extension #2, Review from an Automation Perspective

Here I'm assuming two review points in your traditional process. First, that testers are invited to requirements review sessions and second, that testers expose their test cases for external review as part of their test development process.

Even though we're deferring automation development decisions, we do want to surface automation concerns throughout the artifact review process. Oft times, automation requires data and environmental control that may drive product enhancements to truly enhance the effectiveness of the automation. You want to get these on the table early on.

You're also looking for prioritization information on testing attributes for speed of execution and cycle time and or coverage, so that you can focus your automation efforts in higher impact areas. The overall product technical risk areas should also come into play—driving automation decision-making.

When reviewing requirements, you want to ask questions surrounding feature dependencies and testability. As you begin to more heavily automate, you'll notice more synergy between development unit testing and your automation efforts. The reviews are a wonderful place to share ideas, techniques and tools around testing each feature. The key to good automation requirements definition is to begin blurring the lines a bit between the development and testing perspectives.

Extension #3, Establish a Highly Iterative Automation Model

If I’ve learned one thing from operating in Agile teams, its that short iterations work extremely well in providing feedback on your direction in software projects. If this is true in software development, it’s even more applicable when developing test automation.

When developing any automation effort, you’re stuck behind 2 requirement curves—the product and the automation. You’re also trying to keep up with the implementation evolution for the product. These three forces create a lot of uncertainty and drive up risk. The ONLY way to combat those risks is with a finely grained iterative model.

One of the challenges facing that model is integrating your automation with the product release stream. Rarely do feature delivery and functionality align perfectly to support your automation efforts. I’ve become quite creative at negotiating early, finely grained functional releases from development whose primary purpose is to help our automation efforts. Call them automation Alpha candidates or something similar—to foster a willingness to share early and often.

This has a two-fold benefit. It allows the automation to be developed in small pieces iteratively, while also surfacing product defects early as well. It has a downside in that you’re implementing automated tests on unstable features which may drive higher degrees of rework, but I think that an acceptable tradeoff.

Extension #4, Establish Clear Hand-offs for Infrastructure, Automation, and Execution
From a workflow perspective, it’s particularly important to have automation hand-offs defined properly. By hand-offs I mean the internal release of automation packages within your teams. Think about how you want software delivered to you for testing—with release notes, exit/entry criteria, and in some sort of versioned package that has been tested. These same sorts of workflow process exchanges need to be planned and executed for automation deployment as well.

If you’ve opted in your organization for a 3 phase automation model as illustrated in Figure 2, then these hand-offs should align across those boundaries:

1. Infrastructural components are created or modified, packaged and released to the automation development team (or individuals) for test development
2. Completed, tested and packaged automated test cases are released to the automation execution team (or individuals) for execution
3. The regression or automation execution team (or individuals) run the automation and provide feedback to the test developers for enhancements, efficiency / speed improvements, or general bug fixes

Following this workflow not only helps to manage expectations and planning for project support, but it also defines roles and expectations for deliverable quality within the automation development team.

Extension #5, Extend Traditional Tools & Processes for Automation Support
Many teams forget that automation requires many of the same tools and practices associated with production software development. A good example of this is the configuration management / version control, bug reporting, and change control tool sets and processes. Rarely is it a good idea to create your own environments for these tools.

Instead, you’ll want to extend your existing tools to include automation development as a project or series of projects. And of course you’ll need naming and version conventions for your automation development. You’ll also want to get good at prioritizing defects (in automated tests)

with customer input (from the test runners). Planning and packaging is always important as well. On larger automation efforts I've even instantiated more formal change control and a Change Control Board, just to bring some rigor and control to the automation development process.

The more reuse of tools, techniques and process you can create here the more efficient your automation efforts will become simply because you don't have to train your team or establish unique tools & processes. You can even use examples to "lead" the product organization by exposing best practices from the automation team and suggesting they be adopted more broadly.

Extension #6, Look to Analyze, Increase & Communicate Automation Run-time Efficiency
As your automation development matures and the overall coverage increases, it becomes more and more important from a project management delivery perspective. Thus the PM's will take more and more interest in your plans, schedules and overall execution performance. You'll want to take the time to explain to them and other stakeholders the nuance of your automation strategy. Discuss where the strategy can increase overall testing efficiency and where it can not—then work with them to tailor your plans to maximize project impact and success.

You'll want to share some of the challenges associated with running automation, including automation maintenance, debugging & fault isolation, test environment management, and tools integration. Draw on the similarities to traditional software development as a means of effectively communicating automation state.

You'll also want to share your efforts at making automation run-time more efficient or how it can actually become a speed differentiator in supporting your development projects. This becomes an imperative if you're doing any outsourcing or implementing distributed automation development and/or execution.

While stakeholder communication is important at the beginning of your automation efforts, I consider it an ongoing priority and an extremely important part of your role. You'd be surprised with how many stakeholders seem to *forget* about the value proposition and nuance of test automation and how often they need to be *reminded*.

Extension #7, Properly "Connect" Automation to your Project Phasing

It's important to note that any automation effort is deployed within the context of a higher priority project—the development project it's associated with. This brings into play an interesting set of dynamics where the two are inextricably linked, but also where one clearly outshines the other.

Typically what happens, and it should happen, is that the automation delivery is skewed or phased outside of the mainline product development activity. The testing team becomes responsible for testing the product (the prime directive) while trying to develop automation on early stabilizing and maturing features. The degree to which they can develop this automation AND have it deployed and usable within the current product iteration is dependent upon several factors:

- The organizations software product development life-cycle
- How quickly features within the software are maturing
- Whether the software development team is "on schedule"
- Overall resources available to the SQA team and
- Whether the testing team is "on schedule"

Figure 1 illustrates the typical skewed development cycles for automation within most development projects. Essentially you deliver automation one release *behind* the product delivery schedule. This allows the features to stabilize and sufficient time for proper automation development and deployment into your regression testing cycles. It also keeps automation development off the critical path of the primary development project.

While you can achieve compression of automation into the current release, it requires many of the factors listed above to be mitigated. It also is far more resource intensive if you try to skew the automation work to be truly in parallel with product development. Sidebar #2 contains a good example of handling these compression challenges.

Wrap-up

The key focus of this article surrounds developing your own view to an Automation–SDLC, but not by reinventing the wheel. Instead by leverage your existing product development SDLC and extending its tactics, processes, tools, and practices to include automation development.

If your organizational practices don't include solid development practices, for example chartering, architecture development, requirements definition, and solid reviews, then simply lead the way by implementing them from within your test automation development efforts.

Also, never lose sight that automation development is a software development activity and all of those same rules and methods should clearly apply. Ensure that your stakeholders understand this. Finally, and we didn't formally explore this within the article, but you never want your automation efforts to become a critical path item within your primary product development schedule. Instead skew them out so that you have the time and focus to implement them correctly and without de-railing the project.

Establishing a connected and clearly defined Automation-SDLC is one of the first steps you need to take in your automation path. In the next article in this series, we'll discuss something equally challenging—how to pick the right tests to automate.

Sidebar #1 - Elements of an Automation Architecture

When I think of an *appropriately* defined automation architecture, some of the following come to mind –

Team Elements	✓ Charter your team with clear architectural and leadership roles ✓ Assess & establish a base level of programming / development skills ✓ Develop training materials, on-line guidance, templates and checklists ✓ Establish mentoring or pair-testing relationships ✓ Obtain training for writing, running and framework automation development and maintenance activities
Process Elements	✓ Create coding conventions, templates, and design patterns ✓ Adhere to naming conventions ✓ Use proper test case size guidance for decomposition, granularity & estimation ✓ Establish review, promotion, and removal models for test cases within the regression framework
Tool Elements	✓ Map tool selection criteria to your applied automated testing model ✓ Establish standards for your Configuration Management and Defect Tracking tools ✓ Define *wrapping* conventions that target how you will insulate dependencies and workarounds
Architectural Elements	✓ Decompose your AUT into meaningful layers or components for test case development and efficient regression testing ✓ Establish a versioning model for controlling "packages" of tests for new feature and maintenance updates ✓ Wherever possible, defining libraries of re-usable, well tested, automation components ✓ Develop standards for logging errors (files, formats, review practices) and interfaces to reporting systems (logging, web reporting, defects) ✓ Gather lab or H/W interface information for test environment interrogation, scheduling and control ✓ Use traditional test management – scheduling, results interrogation, defect reporting, metrics

Sidebar #2 – Sensitizing the Development Organization to the Automation–SDLC

I was working in an organization where we were making a tremendous investment in automation—committing to achieve nearly 95% automated test case coverage release over release. To meet this objective, we had to engage new automation in the current product release schedule and we established a parallel test development team to do it. In my experience, this is a rare automation strategy because of the substantial costs—since you are essentially running the two testing efforts in parallel.

However, simply having the people automating evolving features within a products' release cycle isn't the only challenge. In this environment, we were developing UI functional automation for an evolving storage platform management application. Since the application was new, the interface was highly volatile as we exposed it to customers and gleaned their input on look & feel, feature

operation and usability. This volatility impacted our automation efforts to varying degrees. In many cases, we would be blindsided by a “simple” UI change that could cause us massive rework within our test automation. Because we were running in parallel, this had a direct impact on our testing project and release criteria.

We finally decided to sit down with the development team and sensitize them to our automation architecture and approaches—trying to share the impact, both trivial and major, that various development changes could have on our automation. This *impact & pain sharing* went a long way towards leveling the understanding between the two teams.

While we would still receive many changes that impacted us, the number of severe impact changes dropped to near zero. Cross team communication improved as well. This is also one of the valuable side effects of solid automation development effort—bringing the two teams’ closer, gaining technical impact understanding, improved collaboration and in this case, driving mutual respect as well.

Article Figures

1) Something that illustrates the automation skew from production software

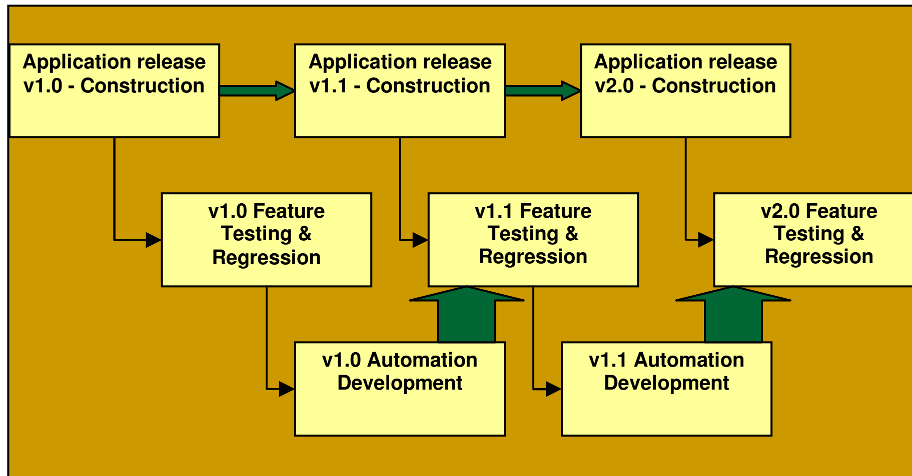

Figure 1, Phasing for automation vs. project skew

2) Illustrates a typical 3 tiered automaton model:

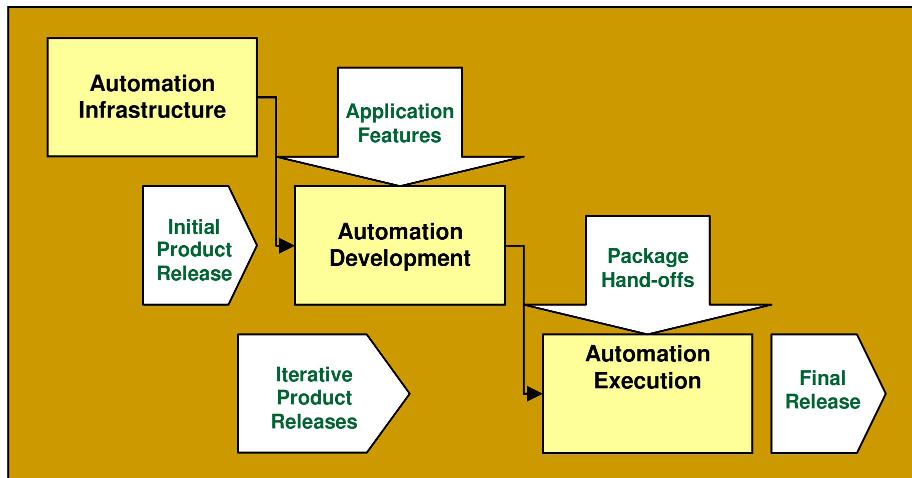

Figure 2, Typical 3 tiered automation development model

1. *Infrastructure (Tools & processes)*
2. *Automation (Test case automation)*
3. *Execution (Environment, scheduling, results analysis & reporting)*