Sigrid Eldh and Serge Demeyer

The Next level of Software Test Automation

Preprint of a bookchapter, to be published by Springer

March 4, 2025

Springer Nature

Contents

Part I
Software Test Automation

Chapter 1
Setting Up Software Test Automation

Abstract This chapter lists actions you should consider as a test automation engineer when setting up your test automation. We provide a detailed guide on setting up software test automation addressing typical scenarios that organizations are facing when embarking on a test automation trajectory. We emphasize the importance of strategic planning, effective tool integration and change management to achieve a fast uptake and avoid common pitfalls.

Learning Outcomes

After reading this chapter, you should be able to describe, analyze and argue the typical actions you need to take when starting a test automation trajectory.

1. The typical scenarios for introducing test automation in an organization.
2. The key aspects to consider when you are automating your test execution for the first time.
3. The key aspects to consider when moving from manual tests to automated ones.
4. Positive and negative impacts on your testing when you adopt CI/CD or DevOps practices.
5. Where to start to automate your test and why.
6. Consequences and risks of changing a manual testing culture into an automated one.
7. Challenges will you need to address when transforming the common ways of working.

1.1 Introduction

Setting up test automation for the first time can be a daunting task, especially for organizations that have relied solely on manual testing. This transition involves a multitude of considerations, spanning people, processes, and technology. Whether you are starting from scratch, integrating automation into an existing system, or shifting to agile methods, the journey requires strategic planning and careful execution.

This chapter aims to guide you through the essential steps and best practices for establishing a robust test automation framework. We explore various scenarios, from automating tests for new systems to transitioning from manual testing, perhaps even adopting a DevOps approach. We discuss the key aspects to consider at each stage. As such you can avoid common pitfalls and ensure a smooth and effective implementation of test automation.

Much of the insights is how you make these transitions to avoid issues later in the automation trajectory. Because nothing is more frustrating if you made an investment to automate your test, and then you are faced with an entire new set of problems that could have been easily avoided, if you just had the information of doing it right from the start.

1.2 Three Different Automation Scenarios

There are many different scenarios when setting up a test automation. Below we list the three most common situations. In reality they will seldom be completely separated and hybrids forms will surely exist over the systems life-cycle. The common theme in all of the three is that your entire development process is moved into a CI/CD build pipelines.

- *From scratch.* The easiest is to set up test automation for a new or existing system with no prior testing other than a handfull of manual tests. This is typical for start-ups or launching a new product-line in larger organizations. The requirements are vague but there is the vision of a minimum viable product that allows to get early feedback from the customers. The system will expand, hence it is important to automate tests in order to ensure continuous and smooth growth.
- *From manual.* The second scenario is based on the most common situation: an existing system which is manually tested based on a series of end-to-end test scenarios. Defects pop up late in the life-cycle and when new features are added it takes a long time to execute the manual test scenario. Here you can gradually migrate your manual tests to automated ones.
- *Towards DevOps.* The third scenario is when you have a fully automated build - test - deploy process in place, where automation is a natural task and where the team is aware of its benefits. Here the main area for improvement is monitoring (and mitigating!) problems while the system is in production.

For every of the main three scenario's, there are several concerns to take into account. We list them below; organized in three main categories

1. Test-oriented concerns
    a. The quality of the available manual test case that are to be automated
    b. The type of tests that should be automated
    c. The coding language and patterns for implementing the automated tests
    d. Templates for test cases (beyond set up + stimulate + verify + tear down)

e. Test Design aspects
f. Test Data preparation

2. Tool-oriented concerns

a. Seamless integration of the numerous tools
b. Test oracles and post processing of results
c. Metrics and dashboards to monitor progress

3. People-oriented concerns

a. Strategy, planning and progress reporting during the transformation
b. The available skills within the team
c. Change management: new roles and workflow
d. Training and resources

Table 1.1 summarize the most important distinguishing aspects for each of the three automation scenarios. We explain them in more detail in the subsequent sections

	From Scratch	**From Manual**	**Towards DevOps**
Key issue when automating	1. Choice of process, framework, and tools. 2. Incremental approach: start small and build it up 3. Integrate the development and test in a build pipeline.	1. Train testers in coding/ automation 2. Avoid manual test habits in automated tests 3. Build for data variation	1. Establish a continuous improvement culture 2. Keep test focus and test skills in the agile team 3. Identify and maintain key test levels and phases
Metrics for progress reporting during transfer	1. What is automated and what remains manual (checklist). 2. Growth of number of automated test cases per week 3. Code Coverage % 4. Number of bugs/faults (warnings) also on tools 5. Turn-around time: from code to completely tested	1. Number of automated test cases / remaining manual test cases (= % automated) 2. Number of new test cases 3. Total test execution time	1. Feedback time: from code/test commit to test result 2. Throughput time: from code commit to deployment 3. Automatic problem reports from customer (incl. bugs/logs etc.) 4. Time: from bug in operation to problem solved
Test case language	Keep same as coding language for unit tests (at least). Type and domain dependent.	Tool choice dependent. Java/Python have many test tools but probably domain specific.	Heavily dependent on system context and technology stack.
Change Management	Invest in training.	Let testers feel in charge of the change.	Do not loose system test aspects.

	From Scratch	From Manual	Towards DevOps
Test Design	1. First automate the "normal cases"; the happy-day test scenarios. 2. Then complement with what has not been covered (code *and* requirement coverage).	Here is test where design techniques will help you in building a stronger test suite. 1. Use input variation; insert test data via parameters. 2. Consider mutation testing to measure the effectiveness of the test suite.	1. Consider a variety of quality metrics including requirements and architecture 2. Ensure that non-functional requirements are checked by specific tests.

Table 1.1: Most important aspects to consider for each of the automation scenarios

1.2.1 Starting from Scratch: Building up a system and test

There are some advantages if you are doing a start-up or a building a completely new system and a completely new way of working, with no prior testing. This happens seldom unfortunately; often at least part of a system, a prototype or a demonstration of concepts exists before you are starting. Most system evolve over time, and it can be hard to realize that you should have automated your tests from the start if you simply started with manual test. Do not make that mistake. You now have am opportunity to take control of your software development, delivery, and testing and automate it from the start. You will save so much time doing it from the beginning.

Independent of the process you use we recommend to adopt CI/CD flow from the start. This is by far the best and easiest way to create a development environment with good habits early on. It is worth to create a complete automatic set up with the elementary steps: you check in your code, build then test. These steps are executed after every commit.

1.2.2 From Manual to Automated Testing

Changing a (mostly) manual testing process in a (mostly) automated one has a big impact on the culture within the team and within the surrounding organization. Hence you need to do careful change management. Are all people involved (including management) on-board with going for automation? Are you targeting a specific part of the testing (e.g., an organization, part of the software, a particular level)? Do you want to separate automating a specific test (type) with automating the actual flow. This will impact several people, roles and even organization if it is to work. Therefore it is best to do a small but realistic pilot to convince people how this could work.

Other that that, you want to keep what testers know best: creating new and novel test cases and keep reusing these tests until they are obsolete. However, when

automating manual tests, there is a lot more to consider. First of all there specific and peculiar things about the system under test which have grown organically and are implicit in the way of working. It might be that it is an embedded system, has real-time components, or handles sensitive information. You need to plan for such peculiarities specifically when setting-up the automation as these risk becoming major hurdles.

1.2.3 Moving to Agile, CI/CD and perhaps DevOps

Changing your process to agile continues integration (CI) and continues development (CD) means that automation will be one of the key focus areas for success. You are to create an automatic flow from requirements, until the release and to operations and monitoring, and then a flow back of data (bug information, but also data to form new requirements) from the operations. It was the increased development and testing speed in the Continues Integration (build and test) and Continues Development (CI/CD) that really forced the operations to be able to ramp up for more frequent deployment. There is no idea to release frequently if there is no receiver. Therefore, moving to DevOps really means preparing for automatic installation, deployment and automating the acceptance testing. DevOps was also very timely as the flow from the operation back to the development suites the constant need of more and accurate data from the real environment - the operations. This could be monitoring data, fault data and other information.

Another pillar of the CI/CD drive is the agile team that is that the team is supposed to be contain (or have representatives) all parts of the DevOps lifecycle. This means it is supposed to include both representatives from operations (and customers) in addition to developers and testers among some roles to be self-sustained and self-organized. This should be carefully taken into consideration when transitioning into an agile development method - as it is very easy to lose the testers expertise and focus with such a transformation of process. As this is not a book on how to change your organization to agile, we are not focusing on other consequences of the agile transition, but only on test automation. We can though suggest that for larger organization, it could be of value to differentiate the goal of some teams create specific test expertise teams to perform some difficult testing that is not easily pushed to everyone - as not every person can be expected to know everything. Especially important is to automate and push non-functional system tests regression tests down to all teams to gain insight and teach all about aspects of, e.g. performance testing, robustness testing, stability testing, installation testing and security testing - even if this can also be kept as specific quality assurance QA teams.

1.3 Where to Start your Automated Tests?

There are several ways to initiate a test automation trajectory and there is no single best answer on what you should do first. Below we list a few common approaches, explaining benefits and common pitfalls

Regression Tests. A commonly adopted approach is to start with the regression test suite, as repeatable test is not only wearing testers out, but can often feel like a wasteful activity. Yet every tester who has done so, knows that good regression suite is bound to stumble on some faults. Addressing the regression suite gives not only the possibility to free up some time and get through the regression suite faster, as well as provide a good safety net for the product. Automation also adds the unique opportunity to express expert testers knowledge in a repeatable manner.

Snapshot Testing. A low-hanging fruit approach for regression testing is called *Snapshot Testing*. You launch the system (subsystem, component, . . .), bring it in the given state, provide a fixed input and then let it dump the output in a file. You compare this file against a previously created file containing the expected output and as long is these files are identical you mark the test as `passed`. But if as much as a single byte is different the test `fails`. This is very easy and fast to set-up. However such regression tests are very brittle. And once they fail it is very difficult to locate the fault.

End-to-end Scenario tests. There are several ways to create test automation depending on where you start from. If you are starting new or have no written tests cases your first focus is of course to define clear use cases and automate them. This is occurs so often that we have different names for it: requirement driven test, behavioral tests, end-to-end scenario tests. In the start of automating such tests, it is often easy to put test cases in a specific order, and then build it up. Basically, you start with login, and then you do the first feature, followed by the next etc; all in a single test case. And then the execution continues with the next test case (that pretty much takes the state, data etc., where the last test case ended). When a test case fails, the entire test suite fails, and you need to run the entire test suite to get to the same point again.

Monkey Tests. There are also tools that makes it possible to automatically traverse the GUI and exploring all execution paths. This gives at least a first check that the system is up and running and can find untested links and combinations in a monkey test fashion. The main reason for not starting with the user interface is that this often changes a lot during initial development hence the automated tests are very brittle. Today there are a lot of automated tools (classified as “capture-playback”) that rather easily traverse e.g., a web interface or a graphical user interface — and capture input fields with support to enter a variety of random data. If you intend to adopt such a tool, pilot it diligently — they don’t necessarily integrate well with the rest of your build pipeline. And they are quite vulnerable to cosmetic changes in the user-interface. Testing a system only through its user interface are not sufficient for most software systems.

Developer Tests. Many start instead from the developers own testing of the code, as this is the first in the pipeline for a new system. It will not take long for developers

to create a build and test framework (or test harness) to run at every code change. If you build from bottom up, more integrated functional test that goes beyond one small unit are needed. The goal should be to get a working system up as fast as possible so when the system is coming together it will allow to check e.g., usability and performance. In general, it is best to run all tests every time you check in code, trying to check in frequent and often. The limitation is of course when the systems grow big, and feedback for a change starts to take too long. Then plans must be made of short, nightly, and weekly test runs.

Unit Tests. If you already have an established a test process, it is often easiest to start automating the unit test framework for the developers and the regression suite of the behavioural functional test. By automating the most boring and repetitive test, you will not only free up time but also get instant quality for some aspects of the tests, as it is easy to make mistakes in tedious work tasks. Try and automate as "different" test cases as possible for a good variety of checks that traverses as many aspects of the software system as possible — and to learn how to create different test patterns for your test library.

1.4 Common Pitfalls and How to avoid them

It should be clear by now that introducing test automation in an organization is a marathon and not a sprint. So it is essential to foresee future problems and avoid them as best as possible. Below we list a series of common pitfalls and how to avoid them. We organize them in three distinct categories: test-oriented, tool-oriented and people-oriented.

1.4.1 Test-oriented Pitfalls

States and Starting Position for Test Cases in automation. For all test automation, you need to create a "starting position" for each of your test cases. For example, you can assume all your test cases runs from a logged in position at a main menu, thus your set up of your test case would be a library function that explicitly brings you to this state. Having different starting positions allows then to vary test executions, creating a more independent test suite. The goal is for each test case to run in isolation — containing all information it needs. This implies that you can run your test case in arbitrary order, or that you can rerun only those that failed. This is how most advanced automatic suites are run today. One must note, that if you have a starting position for a test case, you probably also have a clean-up' in the end. This is to restore values, or empty the database etc. This makes the test suite far more usable, effective, and easy to manage, prioritize and re-execute. However automated test cases then distinguish from real use, when actions are happening continuously and there is no always a clean up in between actions. Ideal is to be able to run test cases

in “both” set ups, meaning, having a clear understanding of what clean up that can be skipped, or what natural order would dismiss the start-up. These are research areas that could be investigated more, and much to little advice has discussed these nuances.

Separate Execution Steps from Test Case Input; Creating Triplets. To create good, automated tests is to always separate data (input) from the oracle (expected output) and the execution steps by using variables or paramneters. This opens an entire world of how to change the input or utilizing tools. Then you need to define and understand the input and how every input is related to an expected result — meaning — you always have triplets for each execution: (Input Values defined for all Variables in the test case), (Expected output for the test case) (Execution steps). Instead of using three test cases to test boundary value (one in-point; one on-point and one off-point) you only have one test case but you use the variables to feed the necessary data. Then you can clump all “allowed” input (e.g., define an allowed range) with the test case and the correct output, and then all “disallowed” outside range values, together with the expected “fault handling” as the result of the test case in another group. This will diminish the number of duplications of the test case, using the parameters as variations. This can also be taken further, e.g., creating an “input generator” as input for a field, and having size of the field varied. The approach is really the key how to easily to quickly create a set of allowed and not allowed input to a test case.

Some tools aid in making this automatic: you can even select that you want 80% of your test runs to go with the “allowed” input ranges, and 20% of the executions to select from the not allowed input. This forces you to declare ranges for what is allowed and not allowed — which is always a good thing to think through when creating test cases, and it automatically varies the selected input in the tool. The tool of course saves what input that has been used. So if a allowed input suddenly creates a fault, you can easily know what value was used. There are many tools that aids in this set up by arbitrary selecting values. And surprisingly many systems fail with strange input. Testing the fault handling is therefore equally important. Here, automation of input selection and the test design separation can make a huge difference.

Avoid copy-paste duplications; Create Library Functions. When should you be creating library functions? Smart library function, or the use of macro’s or keywords that expands are all tricks to make the test case compact and readable, so it is easy to understand. This should not be underestimated. It is often surprising how much is repetitive in a test case. Duplications or “clones” of a test case is by default up between 25% to 40% of all test suites, but we have also experienced some code - with parameter control, have over 90% overlapping or duplicated code. The only difference was really the input - output. It is good to have this in mind when automating test cases. You need to utilize the tool (framework) to invoke a library function when patterns repeat. The idea is that you would use more building blocks for your test case, instead of copy and pasting another test case - and that your test “footprint” can become small (and easy to understand, find and use). Once you have done a good test case, you do not have to repeat the code writing again for a similar one. For a library function to be used, it has to be a clear awareness among the testers

how naming, searching for these libraries and using them works. And there is also got to be tool support for identifying when you are using something that exists in a library function. This is probably not the case in the beginning of automation, and a typical example of where you need a test architect to keep track and teach, as well as review test cases for creating efficient test code. Naming and searching for usable patterns to speed up test automation creation is an area of research that needs to happen.

Smart Names of Automated Test Cases. Naming of the test cases is important. Having encountered a lot of "TC42" (meaning test case number 42 abbreviated), is a name that soon will become meaningless and forces the tester time to actually look what it is. Only in situations where the test case is created new every time it is used and thrown away is the naming something to ignore. Using name as link to other artifacts (e.g. TC_Projectname _componentname_versionA_R1_var4 to describe the fourth test case that addresses requirement 1 in project A belonging to a specific component) often has limitations, and deteriorates if not thought through carefully, as there are often many versions and updates to a test case during creation and during the test and software life-cycle. Here also the test architecture shines through in the naming conventions. Grouping them to type, and location e.g., having a "main" for a performance test case or Sec_func_23_ for a specific security functionality, to describe part of the system it addresses etc. could be helpful. Usually this comes rather naturally. We do not name software functions arbitrarily so why be ignorant on the names of test?

Test code often surpasses source code in size. Therefore, it is important to keep good structure of the code and avoid creating large test cases. Beware that chopped up in snippets could also be hard to follow, so bundling test cases in a way that they are easy to find for the code in question and making it easy to avoid unnecessary repetition of test cases should be on the test architect and testers mind.

Document the Automated Test Cases. It is important to set up a good test code documentation structure from the start — what is the goal or idea of the test case, what is the type of test case, what is the version of the test case, originator/creator (updater), dates, and of course traceability to requirement specifications, design documentation, source code or other things to "bundle" your test case with. This can be done outside the code — or inside — in form of a header to the test case.

If the test cases are well documented, this could speed up the automation time, but will limit the opportunity to improve the overall test suite as the automation is often bound to the existing test cases. It is important at this point to have focus on what and how the new automated test cases are created by common reviews, creating common test patterns and discussing ways to vary the test case.

Locating and storing test cases. Depending on the size of your system, it is often convenient to locate test cases near the source. One way is to choose a "mirroring" system, where all test cases have a similar dependency tree as the source code, components, and subsystems. The issue with this is that as the system changes - it is not automatically that the test system structure also changes - and mirroring will after a while be out of sync, and must be regularly synchronized, something that often feels like unnecessary work. To remedy that, test code then becomes embedded

with the actual source code. This has both advantages and disadvantages. As test code often surpasses the size of the source code, the system can grow in ways that is unexpected. Locating test cases in their right "node" in the dependency tree could be extremely helpful - it is fast to locate test cases when a specific change has been made - and it is easy to determine what parts of the system has been impacted. If using e.g., micro-services, this seems to be the obvious way of handling test cases. The disadvantage is a lot of unnecessary re-work can be done, if no inner sourcing is made of test cases. As a developer once said, how many times do we need to test the sorted list. The need for test architects as a role to cater for such redundancies are necessary, as well as defining and designing the scope, naming, storage, and test structure. The problem arises as the systems grows larger and more complex. For smaller systems, this might not even be an issue.

Generating Test Cases from Requirement Models. A more advanced approach is to model your system requirements up front and utilizing this information to generate either source code or test code. As it is difficult to encapsulate all aspects in a system up front, these systems often need to be complemented by testing of some aspects. This approach of generation tests from a model is often used in industry as a prototype, or early focus on main normal use cases. It is also used extensively for hardware test code verification as well as for safety-critical software. As these approach is maturing in both scalability, tool support as well as more people are trained in these modelling tools, they are becoming more and more common in use. As there are many books specifically focusing on modelling software, systems, and tests – which all are bound by the tool, language, and concept, e.g., timed automata, ADSL, constraint tools, or formal verification concepts, we abstain from discussing this approach in this book.

Utilizing AI/ML models or Large Language Models (LLMs) to set up and automate new test cases. Another scenario is what to think of if you are utilizing for example Large Language Models (LLMs) or other AI/ML models to set up and automate you testing. It differs greatly if the particular model you are using are trained on code in the first place, and on test code in particular, as there is a distinction between source code and test code. Another concern is the fact that the model should be train in the language you are expecting your test cases to be in. The main problem with training the LLM for test code, is that the existing test code you have available might not be of such good quality, hence you are not optimizing your tests. The main reason is that in most test code repositories, there exists a lot of so called test clones, i.e. duplications that not necessarily are good varieties of code - but simply duplications. This can be both an advantage and disadvantage for your training.

Currently giving good and bad examples, and mastering prompt engineering with all its variants of asking the model to self-critique, and self-check its result, might be a way forward. Currently it is advisable invoke external tools to verify the validity of the generated results. We expect that within a few years automating and generating test cases with LLMs and other AI-augmented learning models will definitely be much more common, and also a key aspect in the field - to be able to verify and validate the models outcomes in general, but for test cases in particular.

1.4.2 Tool-oriented Pitfalls

Cross-platform. Difficult steps could be to automatically set up environments for testing (and prepare the data) for the test cases, especially if the customers have different context where your software systems run. In a web development context for example, you maybe need to set up test suites to automatically traverse a set of different browser environments, running on different operating systems. Cross-browser, cross-platform test is not something easy to achieve, but often necessary. Often people stick to one context and framework for their first instances. Note that you also need to set up a set of checker tools, like static analysis, security analysis, dynamic analysis including memory checking - as well as proper debug, trace, and log environments.

Monitoring and Tracing. All tools should be able to have bug tracking in addition to the actual system under test, as well as support, monitoring and tracing abilities. All results of such tools should be automatically displayed in dashboards, for instant view from all, with timestamps, and who did what, and with the aim to drill down data in the logs. This data is then invaluable to be able to present with trends and changes. Discarding old data could be very costly. These data collection(s) could also be used for security reasons, automatic bug identification and many more advanced uses — as it is important also to baseline, manage and maintain over time. It is easy to forget that also the testing, test tools, patches and scripts must be baselined together with the software under test. If you must go back to an older released system you need to be able to reconstruct the entire environment. Even when the tool is gone, and all have forgotten the special scripts that made them work. Best is also to make every test case clearly pointing to specific sections (files, components) in the code it is supposed to address (at any level, both unit, integration, and system level functionality). The advantage with this is that the bug-fixing is a much easier task.

Dependencies. Thinking on dependability issues in your software is very important: e.g., utilizing call-graphs or static dependencies to really know which test addresses what part of the system can save you tons of time as you do not need to execute all tests but only those tests covering the changed parts. Having a setup that can trigger only "dependent" tests is the best (connect test cases to the code structure). Some languages have this feature "for free", making regression testing of a change very easy, as all involved test cases are automatically executed, but not the entire system. The caveat with testing in isolation is that most system are delivered as one unit, and you have an entire group of bugs that comes from other types of dependencies — not so easily tested for. These hidden bugs come from all types of timing and concurrency, environments, resource utilization and other hardware or operating system aspects that can impacts the software and system.

Configuration of Tools. Build and test framework come with numerous configuration options, from compiler optimizations to run-time flags. Best it to to start simple, and then automatically re-run in more and more complex (with more flags set) modes utilizing the tools to constantly scrutinize your code. Setting up good (not only statement) coverage goals as release criteria is a good practice. It is easy to only do "user" functionality or use case tests - as a complement to the testing the

code logic, but very often in a new environment, you forget aspects like robustness, performance, and interoperability issues.

Automating the Test Oracle and Post Processing. Often when starting test automation, testers focus "only" getting the test case executed automatically and forgets the test oracle and post processing of the test case. This happens mostly when it comes to non-functional tests. It is important to spend time in setting up good test oracles, and test verdict functions, that automatically observes the outcome, so you do not have to inspect tedious long log-files to tell if problems have occurred. This might be more difficult in some systems, e.g., if good redundancy exists and is automatic, you might miss that there was a problem. Often you enter a period of semi-automated, or automated assisted test suites, but there is lot to gain to complete the entire automation process. A goal should be that the "entire" test automation, from starting the execution of first test case, to running the entire test run and collecting all passes and fails as results and presenting of the results is completed. Judging e.g., that performance test continually does require clear criteria for measurement periods and requirements on to pass the test case.

Staging your Tests. Regardless of how you set up your test flow, do you check in every hour and run the entire suite? Do you separate your test suite into "smoke tests" (the quick regression suite) and the nightly tests? Do you separate the new test and test cases from the existing system, until the debugging has been done and merge it then - or do you merge it immediately and make the test suite find the integration bugs? Or - do you perform full unit tests with all static analysis warnings cleared - all memory and security tools run before you submit it to the system? This latter part is often used in combination with "gating" - in environment that really do not want poor code to enter the entire system build - as doing that would be very costly as it would affect a lot of other development teams. Or maybe in smaller context you always check in everything - and make sure your build works. Well, some advocate that you have a set of checkpoints in you code and test - and maybe create some staging. It was popular to compare the code with a washing machine. First you "soak" your test in the "wool cycle", if a test (and its code passes) as a checkpoint then you take "tougher and harder test cases at a much higher temperature - and each of the checkpoints provide a way to stop tests and send them back to be fixed. Maybe the first level is all made on a simulated environment, where the last environment is as close to a real environment as possible - before you release it.

Staging test cases makes sense, when you have larger and more complex systems - where a lot of different sub-systems must be present - as well as some real hardware to be able to e.g., run performance test with any expectation the values would be valid - or for that matter - running specific functions, but some cannot - and the expectation is that the entire system (of-systems) is up and running to be able to get accurate measures.

1.4.3 People-oriented Pitfalls

Some Planning Advice for Automation of Tests. Do not be surprised that the first test cases you make takes a lot longer to create. We found that the first test cases to be created often carries the burden of the entire automation set up. Measuring the "final" time cannot be justified. The equation is not that simple - it is of course easy to compare the manual execution time on the critical time-line of the project vs the automated execution time of the test suite — that will be a lot of gain. But people often do not understand that creating test cases takes almost the same amount of time as creating the source code, with similar time to set up and grasp the context, test environment.

From experience the first could take 20 times as long to create, but the next 100 test cases might be much easier to create - as the pattern, set up, etc. are now already created.

Initial investment. One difficulty - where you often need to convince your management that this is correct, is that more than half of the time is used to set up and prepare the test environment and automate so it works with the test suite. This could for some be a bit unexpected. When using simulators and other equipment, or complex databases, with different content and data this could be a source of issues to be solved to get system in a state that the automation drives seamless and continually.

Understand the complete process. The one thing to consider is your current development, delivery, and test process. It is often wise to understand the steps in them, the decision points, and the current tools in use, the code repositories under test and the various roles involved in implementing the system. Many are in the position to move from manual test cases into automation, often with separate groups of testers and developers, but not necessarily so. Starting and automating test cases from scratch is time consuming, especially in the beginning of change to a new process, and it will remain so, until most people are onboard with the change and sufficiently trained. As manual tests are relative time consuming to execute, significant effort will be required because the slow manual tests will be in competition with the automation efforts.

Incremental approach: start small and build it up. To keep up with the releases, specific effort and planning must happen for this scenario and be managed by delaying a release or add extra resources. The actual efforts of automation will often be in the setup of tool and creating the first test and the group of different test patterns or templates that will then serve as library functions. The other effort is to have a clear and well-understood process for the new automation. It is worthwhile to make a clear plan and gain quick benefits, to free up time for continuing the automation.

Testers are in Charge. Note that many (manual) testers might not feel so comfortable in automating and coding test cases. Maybe they came from the user side and has not sufficient coding background. Note that these people might be the only people really knowing and understanding how your system works hence their experience should not be easily discarded. Instead of simply letting the developers "take over" the testing, it might be better to create teams where the testers are in charge, and the developers' job are there to teach and learn how to automate the test case in

the best way. This might be a learning beneficial for both developers and testers. It is extra important that coding is taught with the notion of writing test code as well. A variant is if you have a manual tests to automate and are at the same time changing your process to an agile process, which implicitly means you are grasping more test levels in your agile team and you are relying on an CI/CD build pipelines.

Think differently about your automated tests. A cause for concern when automating manual test cases, is that if you are still thinking in a manual way you might miss out of utilizing the possibilities of the automation. Maybe the test case can be iterated across the system in the same manner, maybe you can create a loop and test the same thing with different input. The result is that you will have fewer test cases that tests more of your software. Another advantage of thinking like this, is to think "libraries" of test case — aspects in a test case that is repeated often should be a call to a library function. Manual test cases often select "one" thing — the most important thing to test, but with automation — you can iterate "all" things — as this often cost very little in coding time — and as execution will be so much faster, it might not be an issue to test all things.

1.5 Conclusion

In this chapter we have addressed four main scenario's of setting up test automation, and as such dealt with the unique issues that might arise from them. With the insights and practical guidance provided in this chapter, you are well-equipped to implement and manage a robust test automation framework.

Study Questions

1. Why are the initial automated test cases taking so much longer to code? Why is it still worth investing that extra time?
2. Create a checklist to incorporate when reviewing test scripts. Start with a list of things that are easy to forget?
3. What are benefits and drawbacks when staging your automated tests?
4. What are risks and benefits with using copy-paste of test cases? How can you remedy these problems?
5. What does a test case triplet contain? Why is it advantageous to create such triplets?
6. In what order should you start to automate your test?

Longer Exercises

1. Make a plan for setting up a new open source test tool for your team. Then download that open source test tool and then set it up for use for the entire classteam. What do you need to do, how long does it take, and what guidelines and education do you need to teach you fellow classteam mates so you can start creating test cases for a common application?
2. You are a team member of an agile team (5 individuals) working an app for following the stock exchange market. You are already passed 7 2-week sprints into development of the initial prototype; a minimum viable product. Your team adopted a test automation strategy based on first automating a unit test suite. Was this a good idea? Give pros and cons. What alternatives could there be?