abstracta

A Complete Introduction to

Functional Test Automation:

Everything you need to know before getting started

Federico Toledo Ph.D

1.

CHAPTER 1:
Getting Acquainted with Functional Test Automation

2.

CHAPTER 2:
Knowing the Basic Principles

3.

CHAPTER 3:
Avoiding Common Pitfalls of Automation

4.

CHAPTER 4:
Running Your Automated Tests

5.

CHAPTER 5:
Final Comments

"Test automation is computer-assisted testing."
– Cem Kaner

INTRODUCTION

It is often said that "Automating chaos just brings faster chaos" and not only faster, but also (paraphrasing a Daft Punk song) harder, faster, stronger...chaos. Yet, seemingly everyone is moving into an Agile, DevOps or continuous integration/continuous delivery environment. Automating tests is increasingly necessary to be successful in said environments. This ebook focuses on the automation of functional tests in general, showcasing the benefits it brings in the most objective way possible. It goes without saying that if you automate without sound judgment, you won't reap any benefits from it. What you are about to read is not a user manual for a tool. Neither is this ebook intended to convince anyone that automation is like a magic wand that will make all of our tests better. As Jim Hazen says, "It's automation, not automagic!" The goal of this ebook is to provide you with a thorough introduction to functional test automation so that you can determine if it's right for you and if so, how to go about it in the best possible way.

CHAPTER 1: GETTING ACQUAINTED WITH TEST AUTOMATION

As we are focusing on **functional** test automation, we first have to discuss regression tests. Although it's one of the most popular types of tests to automate, it's not the only use for test automation.

REGRESSION TESTS

Regression tests are a subset of scheduled tests selected to be periodically executed before every new product release for instance. Their objective is to verify that the product hasn't suffered any regressions. There are three types of tests that we generally automate which I will touch upon in chapter two: unit tests, API tests, and UI tests.

Why are they called regression tests?

At first I believed it meant to ***go back*** to execute the same tests, given that it's related to that. After a while, I realized the concept is actually associated with verifying that what I am testing **has no regressions**. I imagined that "not having regressions" referred to there not being a regression in quality or functionality, but I heard the rumor that the concept comes from the following situation: if users have version N installed, and we install N+1, and the latter has bugs, we will be tormented by having to go back to the previous version, to ***regress*** to version N. We want to avoid these regressions! And that is why these tests are carried out.

IT'S MISTAKEN TO THINK THAT REGRESSION TESTS ARE LIMITED TO VERIFYING THAT THE REPORTED BUGS WERE FIXED, AS THEY'RE JUST AS IMPORTANT FOR SEEING IF WHAT USED TO WORK IS STILL WORKING PROPERLY.

Generally speaking, when the tests for certain functionalities are designed, a decision has already been made about what tests are being considered within the set of regression tests such as the ones that will be executed before every new product release or in each development cycle. Running regression tests consists of executing the previously designed tests all over again.

There are those who argue that by having a checklist of steps to follow and what things to observe, one is not really testing, but **simply checking**. James Bach and Michael Bolton, two experts in the field of testing, often discuss the differences between testing and checking. Testing is where one uses creativity, focus, searches for new paths to take, and asks oneself, "How else can this break?" By checking, one simply follows an afore

mentioned list, thought of by someone else.

A problem arises with this view of regression tests: it makes them sound boring.

Boredom fosters distraction. Distraction leads to mistakes. Regression tests are tied to human error. It's tedious to have to check the same thing again! With a tedious task, it's easy to pay less attention, and in addition, it can lead to a situation where one wishes something to work and subconsciously sees what they want to see in order to bring about the desired result.

Note: I am not saying that testing is dull! I love it! I am stating that routines can be monotonous, therefore, prone to error. Moreover, techies, at least, have a habit of seeing things that can be automated and wonder how to program them so they don't have to do them manually. *That's when automated testing can be introduced, given that robots don't get bored!*

TEST AUTOMATION CONSISTS OF A MACHINE BEING ABLE TO EXECUTE THE TEST CASES AUTOMATICALLY, SOMEHOW READING THEIR SPECIFICATIONS WHICH COULD BE SCRIPTS IN A GENERAL PURPOSE PROGRAMMING LANGUAGE OR A TOOL SPECIFIC LANGUAGE, OR FROM SPREADSHEETS, MODELS, ETC.

For instance, Selenium (one of the most popular open source tools for test automation of web applications) has a language called Selense, offering a format for each command (action) that it can execute; so a script would be a series of commands that abide by that syntax. The tool also allows one to directly export to a JUnit test in Java and other test execution environments.

Here's what some stakeholders might say in favor of using automation:

Developer | *I want to make changes to the application, but I am afraid I might break other things. Testing them all over again would be too much work. Executing automated tests gives me peace of mind by knowing that despite the changes I've made, things that have been automated will continue working correctly.*

Tester | *While I automate I can see if the application is working as required and afterwards I know that whatever has been automated has already been reviewed, giving me the opportunity to dedicate my time to other tests, therefore obtaining more information about my product's quality.*

Developer | *When I have a big system in which changes in one module could affect many functionalities, I hold back from innovating in fear of breaking things.*

User

When I am given a new version of the application, there is nothing worse than finding that what used to work no longer does. If the error is in something new, then it's understandable, however, when it has to do with something that usually worked and now doesn't, then it's not so easy to forgive.

WHEN CAN WE SEE THE RESULTS?

We may be prone to believe that if the tests find a mistake, that's the moment in which we're reaping the benefits and then we can measure the amount of bugs detected by the automated tests. In reality, the benefits immediately appear from the moment we start modeling and specifying the tests to be carried out in a formal way. Afterwards, the information resulting from the execution of the tests also provides great value.

Detecting an error is not the only useful result, but also the confirmation that the tests are verifying what they should is useful as well. **An article in *Methods and Tools* states that a large amount of bugs are found upon automating test cases.** When automating, one must explore the functionalities, test different data, and so on. Generally, fiddling around with the functionality to be automated takes a little while. Afterwards, one executes it with different data to prove that the automation went well. At that time, a rigorous testing process is already taking place.

Note, if we automate tests in one module, and we consider that it's good enough with those tests, do we stop testing? The risk here is that the automated tests aren't covering all the functionalities (like in the the pesticide paradox). It depends on the quality of the tests. We might have a thousand tests and therefore believe that we have a solid amount of testing, but those tests might not be verifying enough, may be superficial, or too similar to each other.

THE VALUE OF AUTOMATING IS NOT IN THE AMOUNT OF TESTS OR THE FREQUENCY IN WHICH THEY ARE EXECUTED, BUT IN THE INFORMATION THEY PROVIDE.

WHY AUTOMATE AND TO WHAT END?

If we take the traditional definition of automation from industrial automation, we can say it refers to a technology that can automate manual processes, bringing about several other advantages:

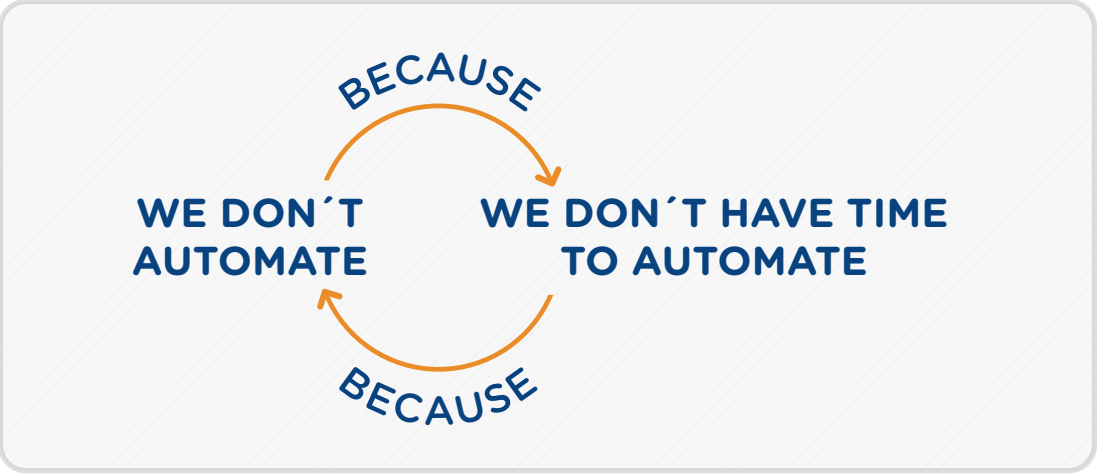
- Improved quality, as there are fewer human errors
- Improved production performance, given that more work can be achieved with the same amount of people, at a higher speed and larger scale

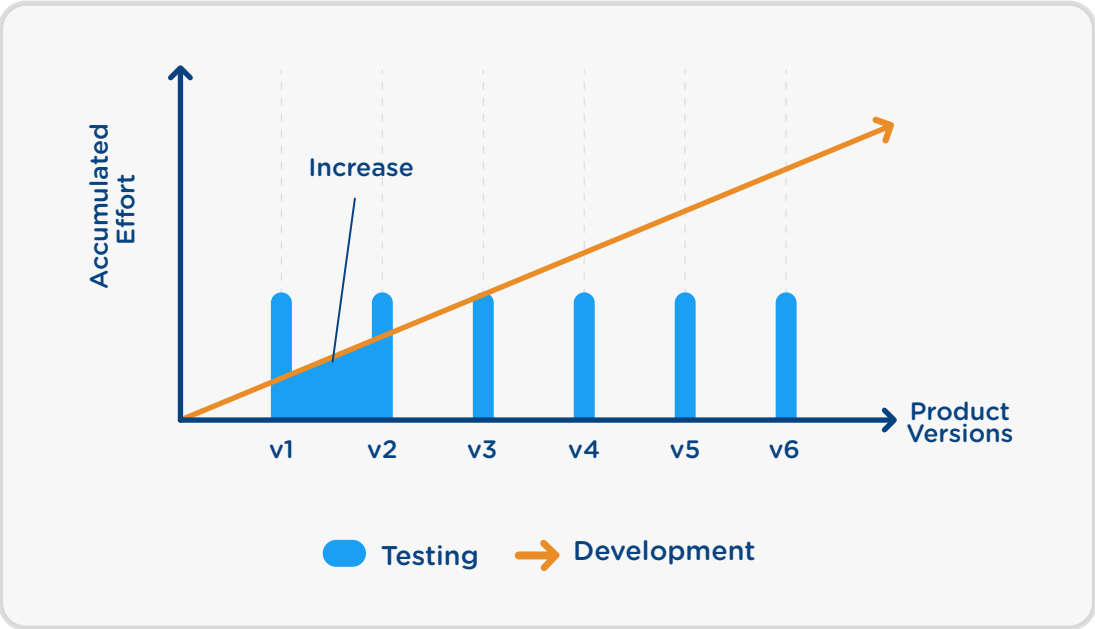
This definition also applies perfectly to software test automation (or checking).

Now, I would like to bring the "zero accumulation" theory forward. Basically, the *features* keep growing as time goes by (from one version to the next) but the tests don't grow with them (I haven't heard of any company that hires more testers as it develops more functionalities).

AS A PRODUCT GROWS, WE HAVE TO CHOOSE WHAT TO TEST AND WHAT NOT TO TEST, LEAVING SOME THINGS UNTESTED.

The fact that the *features* grow with time means that the effort put into testing should grow in a proportionate manner. Herein lies the problem of not having enough time to automate, given that there is not even time for manual testing.





Ernesto Kiszkurno, an Argentine consultant at a firm specializing in quality and process engineering, says that the hardest thing (*aka most expensive*) in testing is design and execution. We would consider that the design is cumulative, given that we design and record it in spreadsheets or documents. The difficulty is that test executions are not cumulative. Every time a new version of the system is released, it's necessary (well it's desirable, yet should be necessary) to test all the accumulated functionalities, not just the ones from the last addition. This is because it's possible that some of the functionalities implemented in previous versions change their desired behavior due to the new changes.

The good news is that **automation is cumulative**. It's the only way to make testing constant (without requiring more effort as time goes by and as the software to be tested grows). The challenge is to perform testing efficiently, in a way that pays off, where we can see results, in a way that adds value, and so that it accompanies the *accumulation* of the development.

NOTE! TEST CASES HAVE TO BE EASILY MAINTAINABLE. IF NOT, THE ACCUMULATION CANNOT BE CARRIED OUT EFFICIENTLY.

WHAT'S THE RETURN ON INVESTMENT?

The cost of a single defect in most organizations can offset the price of one or more tool licenses.

Before we dive into the "how," "who," "what," and "where" of automation, let's look at its business and IT value to understand the "why."

To find out if we should invest, we must analyze whether this investment is related to just the cost of quality or if it will lead to minimizing other costs associated with the lack of quality down the road. We will need to look at it in numbers to understand it and believe it.

For these tests (checks), although they are "automatic" and "executed by a machine," we will need skilled people. As Cem Kaner once explained, a tool does not teach your testers how to test and if the testing is confusing, the tools will reinforce the confusion. He recommends correcting the testing processes before automating.

In addition, the idea is not to reduce the amount of staff dedicated to testing, seeing as manual testing is still needed and automated tests require some effort for their construction and maintenance. So, if we don't save money on staff, then where are the financial benefits?

Let's put the benefits to the test.

We'll look at the case study from Paul Grossman's white paper, "Automated Testing ROI: Fact or Fiction?"

Consider the case of performing manual testing only. If a tester on average costs $50 an hour and if a senior tester who creates automated tests costs $75 an hour, that would cost about $400 and $600 respectively per day per tester.

Now, consider a team of 10 testers, five senior-level and five entry-level, with a monthly loaded cost of $105,000 (for 168 hours per month). We'd get a total of 1,350 hours costing $78.00/ hour (this is assuming each tester realistically works 135 hours per month due to breaks, training days, vacations, etc.). **If we automate testing, the cost of labor would remain the same, but for the effort of 3 test automation engineers, we'd achieve 16 hours a day of testing and will run 5x more tests per hour.** This results in the equivalent of 5,040 hours per month of manual testing created by the three test automation engineers. Then, consider the rest of the team doing manual testing (7 people x 135 hours/month). That amounts to 945 hours more, ending with a combined total of 5,985 hours of testing at $17.54/hour ($105,000 divided by 5,985 hours).

Manual	Automated
Hours (10x135) = 1,350 hours	**Hours** (3x21x16x5) + (7x135) = Total of 5985 hours
Cost $78/hour	**Cost** $17.5/hour

IN THIS SCENARIO, WE'VE DRAMATICALLY REDUCED THE COST OF EACH TEST HOUR FROM $78 TO $17.54, WHICH IS A BENEFIT THAT THE CFO WILL CLEARLY UNDERSTAND.

Or you could look at it this way; we have increased testing from 1,350 hours to 5,985 equivalent hours and gained $315,000 worth of testing per month for the same cost (5,040 times the average hourly cost of a tester).

Not only do we test quicker, but the test coverage is expanded, which means we can find more bugs! But, finding bugs certainly means we will have more work to do and need boatloads of more money to fix them, right? Not necessarily.

It costs much less to fix bugs that are detected earlier in the development cycle. In the chart below, you can see the cost of correcting a defect detected by the stage in which it has been found (development, integration, beta testing, or production). We will assume that it costs $75/hour to fix bugs. These costs don't include hidden ones as well such as loss of reputation, user confidence, and even equipment wear.

	Coding/Unit Testing	Integration	Beta Testing	Post-Release
Hours to Fix	3.2	9.7	12.2	14.8
Cost to Fix ($)	240	728	915	1,110

Data Source: (Planning Report 02-3, "The Economic Impacts of Inadequate Infrastructure for Software Testing," Prepared by RTI for National Institute of Standards and Technology, May 2002, p 7-12.).

As you can see, the sooner we find bugs, the cheaper and easier it is to fix them. If we practice test automation, it's more likely that we'll find more bugs before the beta testing and production phases. It's difficult to estimate how much, but in general for every bug that we find in the early stages, we will save $200 (not bad)!

CODING DEFECTS FOUND POST-RELEASE COST FIVE TIMES MORE TO FIX THAN THOSE FOUND DURING UNIT TESTING.

It's safe to say that there is a high ROI of test automation and that it is a GOOD investment because it provides value in two ways:

BUSINESS VALUE

- Improve software quality
- Avoid operational problems
- Maintain a good customer image
- Avoid legal problems and minimize risk
- Decrease the cost of fixing bugs by 5x

IT VALUE

- Test in parallel, in an unattended manner, on different platforms
- Simplify routine tasks
- Run more tests without increasing costs in the same amount of time
- Increase scope of coverage
- Find the hard-to-detect defects earlier, when they are easier to fix
- Improve overall software quality

WHAT TO AUTOMATE AND WHAT NOT TO AUTOMATE?

As I have already stated, after designing the tests, we have to execute them every time there is a change in the system (like before every new release of a different version). Even though its benefits are well known to all, it can also be argued that it requires a certain effort to automate and maintain regression tests. Almost all automation tools provide the possibility of "recording" the tests and later being able to execute them, which is known as record and playback. This usually works for simple tests and when learning how to use the tool. However, when we need to carry out more complex tests, it is generally necessary to know the way the tool works more in-depth, how to handle sets of test data, manage test environments, the test databases, and so on. Once these are taken care of, we can execute the test as many times as we want to with very little effort.

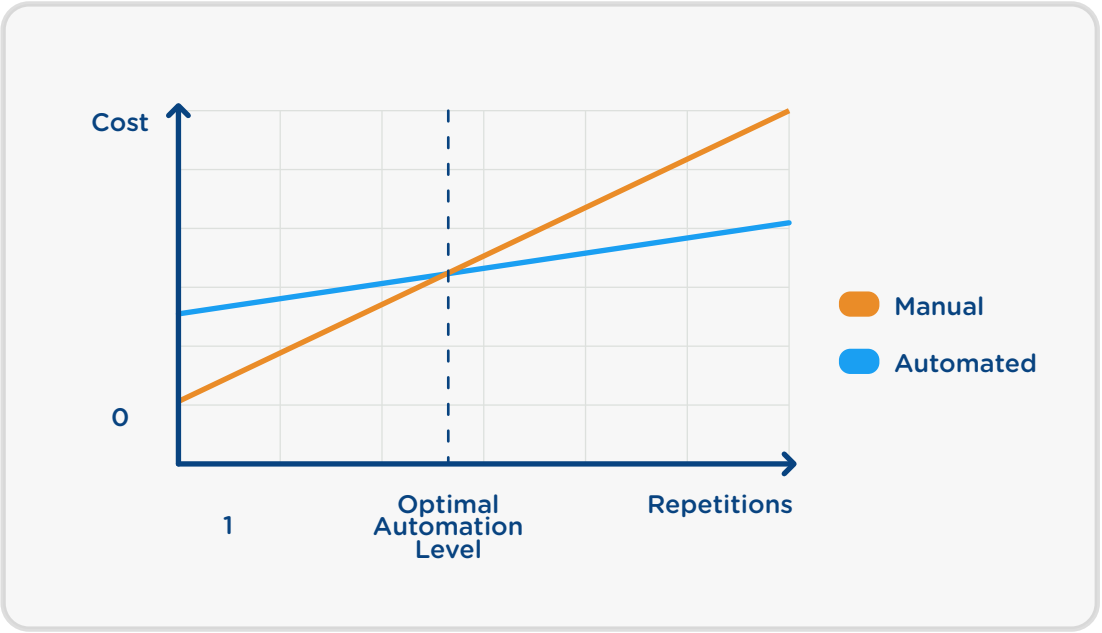
The best tests to automate are the ones which are quite repetitive, given that it's necessary to execute them many times (either because it is a product which will have a lot of versions or due to making frequent fixes and patches or because it has to be tested on different platforms).

If tests for a development cycle are automated, the automated tests for the next cycle can once again check what has already been automated with little effort, allowing the testing team to increase the volume of the set of tests, therefore increasing coverage. Otherwise, we would end up having test cycles that are larger than the development cycles (and they'd keep getting larger every time) or we would choose to leave things untested, accepting the risk that that involves.

One of the most significant factors is the amount of times we are going to have to repeat

the execution of a test case.

However, the cost of a single repetition is larger in the automated case. The graph below represents this hypothetically. Where the lines cross is the inflection point at which one makes more sense cost-wise than the other. If the test case is executed less than that amount of times, it's better not to automate. Conversely, if we are going to test more than that amount, then it's better to automate.



The amount of times we execute a test is determined by many things:

- The number of versions of the application that we want to test
- The different platforms we will be executing on
- The data (Does the same test case have to be run several times with different data?)

COMMON CHALLENGES OF STARTING OUT AND HOW TO OVERCOME THEM

Unfortunately, a lot of automation projects fail because it is so much easier said than done. When a team decides to start automating, it's a very tall order that should not be brushed off as just another task. These are just some of the typical problems that can stop your automation efforts dead in their tracks and how you can overcome them.

CHALLENGE 1:
RECEIVING THE GREEN LIGHT FROM MANAGEMENT

As with any company department, employees are always asking for things that may or may not be allowed for in the budget. Testers may already know that automation offers both business and IT benefits, but how can testers convince the appropriate stakeholders to allocate the necessary time and funds for implementing test automation?

To prove to management that the financial benefits are substantial, one can show them the simple breakdown I did of the ROI of test automation. He or she should be impressed by how a team of 5 senior and 5 entry level testers could hypothetically reduce the cost of testing from $78/hour to just $17.58/hour and increase testing from 1,350 hours per month to 5,985 equivalent hours, gaining $315,000 worth of testing via automation. Not to mention all of the qualitative benefits of automation that we have gone over.

It is important to also be very transparent. Don't lie to and say that automation doesn't require much effort up front, because it truly does, but in the end, it's worth pursuing!

CHALLENGE 2:
SELECTING AND USING THE APPROPRIATE TOOLS

Often times, teams don't get past this phase due to several reasons. They may lack the expertise to use a certain tool, the tool they want doesn't exist, the tool in consideration doesn't offer 100% test case coverage, or the cost is too high, etc.

If you don't have a sufficient base knowledge for how to use a tool, you have a few options:

- Take an online course
- Have someone that helped create the tool come and give training sessions
- Hire a consultant to help you master it
- Outsource your automation efforts. It may be quicker to simply hire someone who already has the expertise to use it and employ him or her for your automation project.

Learn more about the do's and don'ts of software testing outsourcing here.

If you think a tool doesn't exist, it might be good to confirm it with the testing community. Go onto forums like uTest, Stack Exchange, or Testers.io where fellow testers are often found discussing developments in testing.

If you can't find the specific tool you need, you might want to see if it's feasible or

worthwhile to create it yourself. In our case, we have created several tools that we have made available to the open source community on our Abstracta Github account. We also created an automation tool for GeneXus called GXtest which enabled the software development platform to reduce the time invested in designing and maintaining regression tests by over 50%, making it possible to execute millions of test cases per month. Learn more about it here.

In case the tool you have doesn't do everything you need, consider using a multi-tool approach. Remember, it's impossible to test absolutely everything, but you can use the tools that test the most important things.

Lastly, if a tool is out of budget, do a quick cost vs. benefit analysis and present your case. You can measure the damage done by a previous bug you have encountered and show how much time and money you could have saved if you had had the tool in place.

CHALLENGE 3: IDENTIFYING A STARTING STRATEGY

Ok, so you might have all the tools and support to begin automating, but what do you actually automate and how? Unfortunately, the tools themselves do not tell you what to automate, just as new parents find to their dismay that children do not come with a handbook! Will you raise a generation of outstanding automated tests or will they turn out to be spoiled rotten? Of course you'd hope for the former!

In reality, you can't automate everything so you have to be strategic. You can use two approaches to help with this which we will go over in the next chapter of this ebook:

- Risk-Based Testing
- The Automation Pyramid

CHALLENGE 4: SETTING REALISTIC EXPECTATIONS OF AUTOMATION

No matter how great your tools and processes are, it's important to remember that testing is **never complete**. Test automation is not a panacea for bug laden systems and shouldn't be used in place of, but in conjunction with non-automated tests. Remember that the value of a test comes from the information that it provides, not the quantity of tests executed, nor the frequency. What we care most about is if we are getting the right information so that we can make the best possible decisions when improving the quality of our systems.

Make sure your team and management agree on and understand the desired outcomes from your automation plan so that everyone is on the same page!

CHAPTER 2: BASIC PRINCIPLES OF TEST AUTOMATION

He who thinks a tool can fix all problems, has a new problem.

Generally speaking (and even more so in the tech world), we tend to search for a tool which will fix **all** of our problems. The fact of the matter is that having a good tool is not enough for doing a good job. *If you do not know which tests are the most important and which tests are the most applicable for automation, the tool will only help perform a bad test faster. (Fewster & Graham)*

In this chapter we will begin to look at some of the considerations to take into account so that your tests really do provide the benefits you want and to avoid common causes of failure. You will find that this section has an almost chronological order, in the sense that I begin with the basic concepts and afterwards take a look at the different activities and ways of designing tests in the same order in which you can do them in your test automation projects.

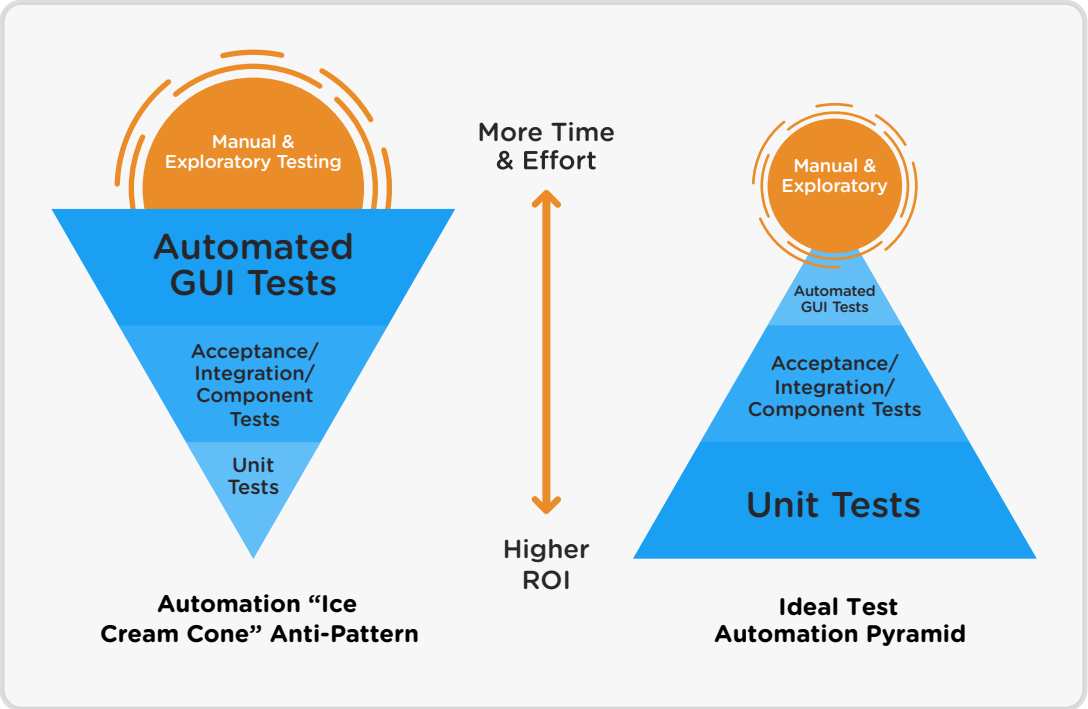
THE AUTOMATION PYRAMID

Many agilists adopt automation as it helps to speed up the process of testing and the entire development process. If you want to understand more about agile environments, you can find a good explanation here. In non-agile software development, many people end up inadvertently falling into the "ice cream cone anti-pattern" for testing by putting more emphasis on automating at the UI level.

I'm more fond of the practice that flips that ice cream cone upside down. Made popular by Mike Cohn, the agile test automation pyramid gives you the most bang for your automation buck, improving the ROI of automation and guaranteeing that you will receive the most benefits from automation.

WHEN MOST OF OUR EFFORTS ARE FOCUSED ON AUTOMATION AT THE UI LEVEL, THE FOCUS IS ON FINDING BUGS, WHEREAS WITH THE AGILE PYRAMID, THE IDEA IS TO PREVENT THEM.

In the figure below, you can see how the two approaches differ.



BASE LAYER: UNIT TESTS

Most of one's testing should take place in the development stage, running unit tests after every build. These tests are the easiest, cheapest, and fastest to complete and are an important aspect of test driven development. Running more tests at a lower level allows us to "check our work" as we go, getting feedback immediately and allowing us to know exactly where the bugs are when it is much harder for them to hide. Here, the bugs will also have a shorter life span, having been born and removed in less than a minute, perhaps. During the UI tests, bugs will have lived for much longer and will put up a greater fight since they've lived there very comfortably for a longer period of time (perhaps even a couple of days).

MID-LAYER: API/ INTEGRATION/ COMPONENT TESTS

After we run all of the unit tests and they pass, we can move onto the API/ integration/ component testing phase. Integration tests are run to make sure that all the components work together properly. This is where we can test most of the logic and business processes without going through the UI. It is best to automate here as much as possible. If you have to decide whether to automate at this level or at the UI level, here you'll have fewer

problems, easier maintenance, faster test execution (meaning finding bugs sooner and decreasing their lifespans) and you get to test the logic of your system. These tests are slower and more complex than unit tests, but they are still faster and less brittle than UI tests.

TOP LAYER: UI TESTS

Last and run least are UI tests. It's best to run as few as possible as they are costly, more difficult to prepare and maintain, and take a long time. Here you just want to make sure that the user interface itself works properly, knowing that all the other aspects of the system should have already been tested. Automate only the most critical tests end to end. For example, starting from the user login and ending with the approval of an invoice. It's also helpful to focus on things related to the browsers or the UI. Be careful with these tests as they are more likely to provide false negatives and false positives. After running the UI tests, manual and exploratory testing can be conducted (as shown in the sphere shape above the pyramid).

TL;DR?

THE PYRAMID IS A STRONGER, MORE BENEFICIAL AND COST-EFFECTIVE WAY TO IMPLEMENT TEST AUTOMATION.

It provides a strong testing base in the unit testing phase from upon which to build further testing in the integration and UI phases.

UI AUTOMATION APPROACHES

There are several automation approaches, and for every context some will be more useful than others. It is good to bear them in mind when selecting the test strategy and even more for selecting the adequate tools.

Let's now turn to three approaches that based on my experience, are the most common and most beneficial (typically when thinking of automating at the user interface level).

SCRIPTING

This is one of the most common approaches to test automation. Usually tools possess a language where one can specify the test cases as a sequence of commands that manage to execute actions on the system being tested.

These languages can be tool-specific, as in the case of Selense from the Selenium tool, or they could be a library or API for a general purpose language like JUnit for Java.

The type of commands provided by the tool will vary according to the level of the test case. There are tools that work on a graphic interface level so we'd have commands that allow actions like clicks or data input in fields to be executed. Others work on a communications protocol level, so we'd have actions related to those, for example, at an http level like the HttpUnit tool, which gives us the possibility of executing GET and POST at protocol level.

Imagine the following example: a JUnit test invokes a functionality of the system directly onto an object being tested. We use a certain input value for its parameters and the output parameters are checked. In this case, the execution is on an object, whereas for Selenuim, the parameters will be loaded onto existing inputs in the websites, and then the execution of the functionality will be performed by pressing submit on the corresponding button. Now let's visualize a Selenium automated test case. First the values are added in two inputs with the "type" command and then we click the button to send the form.

In order to prepare automated tests following this approach, it's necessary to program the scripts. For this we need to know the language or API of the tool and the different elements of the system we are interacting with. For example, it could be the buttons on a website, the methods of the logic we want to execute, or the parameters we need to send in a GET request of a web system.

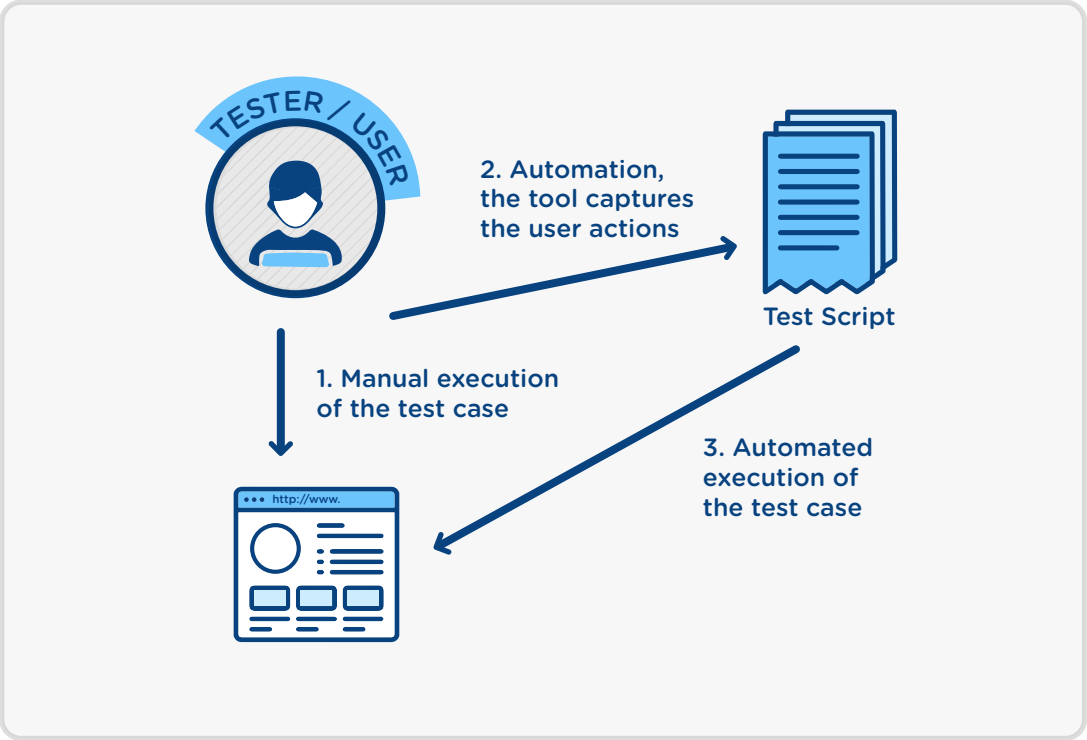
RECORD AND PLAYBACK

Given that programming scripts is usually an expensive task, the paradigm of "record and playback" will allow us to create (at least the basics) of the scripts in a simple way.

The point is for the tool to be able to capture the user's actions (record) on the system we are testing, and can later put that in a script that can be reproduced (playback). Let's try to imagine this process by breaking it down into three parts:

- The user manually executes on the system being tested
- At the same time the tool captures the actions
- It creates a script that can later be executed onto the same system

Without this sort of functionality, it would be necessary to manually write the test cases and in order to do so, as previously mentioned, insider knowledge of the application and the language for scripting of the tool would be essential. Maybe it would be possible if done by a developer, but for a tester it can prove more difficult. That's why it's desirable to posses this functionality.



The scripts created by the recording of the user's actions usually have to be modified, therefore we must know the language and the elements of the system being tested but, fortunately, it's much easier to edit a generated script than to program one from scratch. Among the changes that might be necessary or useful, we could mention test parametrizing, so that the script includes different test data (*following the data-driven testing approach*) or by adding certain logic to the scripts. For instance, we can use structures like *if-then-else* in case different workflows or loop structures need to be pursued.

Scripts can then be recorded from our execution of a test case on the application. For automation, it is necessary to first design the tests and then record with the tool. Bearing this in mind, we could argue that the most difficult and expensive task is that of designing tests.

MODEL BASED TESTING / MODEL DRIVEN TESTING

The next level of automation implies automating not just the test execution but also its design. I suggest following a model based approach, which can come from two different sources:

- Models created specifically for testing purposes
- Models created specifically for development purposes

On the one hand, this approach can rely on the tester somehow developing a specific model for test creation, for example, a state machine or any other type of model with information on how the system being tested should behave. On the other hand, you can generate tests from the information of the application or the different artifacts that you have produced during the development process. These could be the UML diagrams from the design stage, use cases, user stories, the database diagram, or the knowledge base (if we are talking about a system developed using GeneXus).

The results obtained will depend on each tool, but generally speaking, it will be specifically designed test cases in a certain language, tests data or scripts of automated tests in order to directly execute the generated test cases.

This way, the tests are based on an abstraction from reality via a model. This allows one to work in a higher degree of abstraction, without having to deal with the technical difficulties, focusing only on the model of the problem, and making the tests easier to understand and maintain.

I will continue talking mainly about automation with scripting, relying on tools like *Record and Playback* that allow us to parametrize their actions in order to follow a *Data-driven Testing* approach. In addition, I'll make suggestions related to test design, and different aspects of the automation environment, considering the design will be done manually, not necessarily with model based technique tools.

THE MOST IMPORTANT TEST AUTOMATION PATTERN: PAGE OBJECT

As I mentioned before, it's VERY important to work with maintainable code for the test automation, otherwise your test cases will not be considered useful, will not have the expected ROI or will die because they will be more expensive to maintain than executing the test suite manually. What can we do in order to have maintainable test code? Well, basically we can do the same thing that we do to have maintainable code for our applications, such as paying attention to different internal quality metrics to using proper design patterns.

Design patterns are a well-known solution for this problem. They are adaptable to different contexts so that we don't need to reinvent the wheel every time we face similar problems.

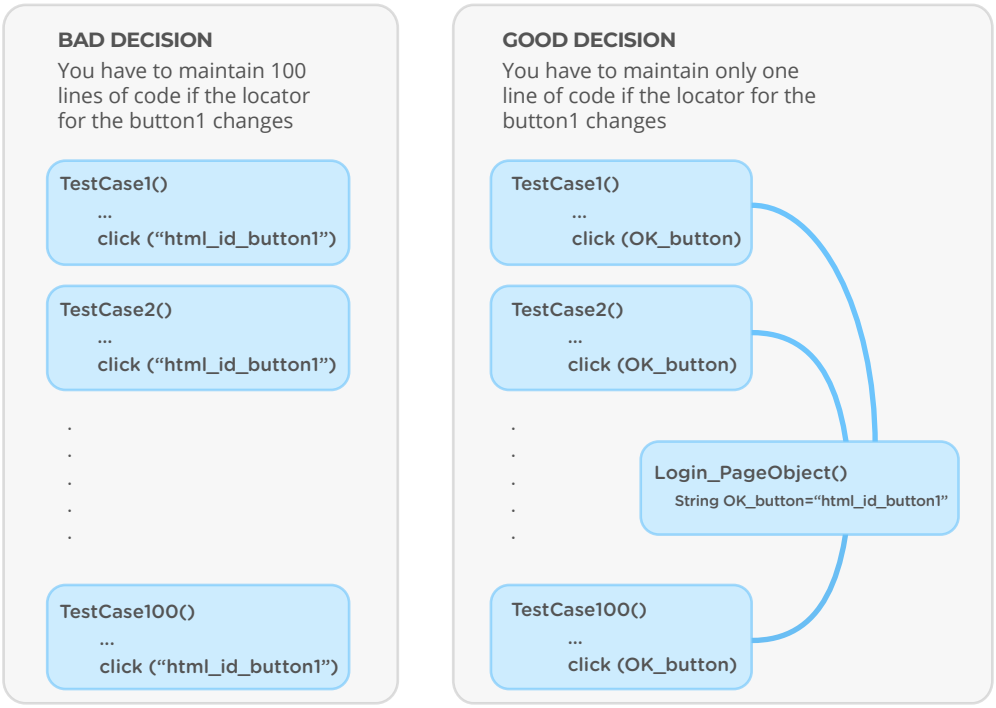
As you can imagine, creating and updating test code in an efficient way is a very common problem. The solution mainly focuses on the abstraction layers, trying to encapsulate the

application in different objects that absorb the impact of the changes that our system under test could suffer during its development. It's pretty typical that the User Interface gets modified from its structure to its elements or its attributes. So, our test framework should consider that these elements could potentially change and we must be prepared for that.

What can we do for that? Well, the page object pattern proposes having an adaptation layer conformed by specific objects to manage the interaction between the test cases and the application under test. For that, we mainly need to store the different element locators in a very organized way. For example, we could have a class for each web page (if our system has a Web interface, which is the most common situation when we apply this pattern) and we could have different attributes for each element with which the test interacts.

If you want to see some good examples, check this.

Which problem are we solving by having maintainable code? If we have 100 test cases which interact with a certain button and the development changed the element locator for this button, then we would need to maintain 100 test cases or at least 100 lines of code! The solution for that is very simple: encapsulation. We have to have the element locator defined in one single place and reference the element from the 100 test cases. Then, if the change happens you only need to maintain one line of code and all your test cases will work properly.



Going one step further, there is more encapsulation and abstraction that could be added to our test architecture. We could define different test methods in the page objects, including common actions. The most basic example is when you have the login page, you could have a login method which executes the following steps:

- Access the URL
- Type the username
- Type the password
- Press the button
- Verify if the login was successful

Again, if you do not do that, then you will have many lines of duplicate code, undermining maintainability.

Therefore, when you find yourself starting an automation project and designing your test framework, take into consideration at least the two following things as the basis:

- Page object pattern
- Data-driven testing

TEST DESIGN ACCORDING TO GOALS

As with everything in life, we must have an objective in mind for test automation. We must think about what we want the automated tests to be used for and act accordingly. So, we will have to make certain decisions about going one way or another, selecting certain test cases instead of others and designing them with a certain approach or strategy.

Even though we might think that our test automation objectives are trivial, they can actually vary widely from one company to another.

These are just a few noteworthy goals of functional test automation that an organization can have:

- Consistent and repeatable testing, making sure the same actions are always being executed, with the same data, and verifying every single thing that has to be verified, both at the interface level and at the database level
- Run test cases unsupervised
- Find regression errors at a lower cost and at an earlier stage
- Run test cases more often (for instance, after every commit to the code repository)
- Improve the software's quality (more testing = more chances for improvement) and therefore increasing user confidence
- Measure performance
- Test different operating systems, browsers, settings, DBMS (Database Management Systems), and so on without doubling the execution cost
- Reduce the release time to market/run tests faster
- Improve tester morale, executing routine tasks automatically
- Follow a continuous integration approach, and therefore detect bugs earlier and run test cases at night
- Have a set of test cases to run before every new product version
- Have basic tests such as smoke tests or sanity checks to know if the version released for testing is valid or catches fire easily
- Make sure that the incidents reported don't come back to the client

Even though some of these might look similar, it is important to ask ourselves which of these objectives we want to accomplish and how to measure whether or not we are hitting our targets before beginning with any automation project.

None of these objectives are exclusive, they even complement each other.

It is also said that automation allows us to find mistakes more easily than if done manually. For example, memory errors that happen after having run a functionality many times, not to mention if we are dealing with concurrency or performance tests.

SOME THINGS ONLY A MANUAL TESTER CAN OBSERVE, AND OTHERS ARE MORE LIKELY TO BE FOUND BY AN AUTOMATED TEST.

For instance, if our aim is to verify that every server request gives us an error free response code (in the case of http would be no 404 or 500 error) or if we want to see that all URLs are configured with https, that can be programmed and automated so that it is verified in all tests, whereas a manual tester probably wouldn't pay attention to it every time.

It is just as important to keep the objectives in mind as it is to define the objectives. A possible danger could be, for example, when the person in charge of automating is a programmer and when using the tool, they find it to be technically challenging and entertaining. So much so, that they end up automating a lot of functionalities without having analyzed beforehand how relevant they actually are in order to reach their goals.

RISK-BASED TESTING

With an objective in mind, it will be easier to determine which test cases to automate. For this, we use "risk-based testing." This test strategy gives higher priority to testing the elements that are most at risk of failing, whereas, if said failures were to occur, they would carry the greatest negative consequences.

With this in mind, it's paramount to run a risk analysis to decide which test cases to automate, taking into account different factors:

- How important or how critical it is for running the business
- The potential financial impact of the errors
- The probability of failure (it would be a good idea to ask developers who would know, for example, which module had to be finished in less time than others)
- Service Level Agreements (SLA)
- **If there is money or lives are at stake** (it may seem dramatic, but we know that many systems deal with highly sensitive information)

When thinking about the probability of errors popping up, we must also think of the ways the system is used. Not just what the most popular functionalities are, but also which flows, data, and options are most popular among users. In addition, we should combine how critical the operation is because as an example, paying wages might only be executed once a month, but a mistake in that functionality would come at a high cost if say, it paid wages ten times a month!

When thinking about how critical a test case is, one must consider how critical the product as a whole is and not just think about the next release, given that the criteria will more than likely be different.

For categorizing tests by priority, a very widely used method is MoSCoW, which is an acronym for Must, Should, Could and Won't (and yes, there are test cases to which you will say, "No, I won't automate that").

Once the priority of the tests has been established, it would be advisable to check them every once in awhile, given that the business or client requirements might change.

How much should be automated?

Every case will be different, but some recommend starting off with an aim of 10% or 15% of regression tests, until reaching approximately 60%. For me, it would be important not to automate 100% of the testing and substituting manual testing with automated tests as that would go against any tester's work ethic.

Defining the steps, which data, what response to expect, etc. is just as important as the test case selection.

HOW TO AUTOMATE

Let's say we already have our test cases designed. We will start by checking the functionality inventory (or backlog or wherever you store this information) and assign priorities to each. Afterwards, we will assign priorities to each test case prepared for each of the different functionalities. This organizing and prioritizing will help divide the work (in case it's a group of testers) and to put it in order, given that grouping the test devices by some criteria, for example by functionality, is highly recommended.

Test case designs for automated testing are better off being defined on two levels of abstraction.

On the one side, we have what we will call **abstract or parametric test cases** and on the other hand, the so called **specific test cases or concrete test cases**.

Let's review these concepts and apply them to this particular context. Abstract test cases are test scripts that when indicating what data will be used, do not refer to concrete values, but to equivalency classes, or a valid set of values, such as "number between 0 and 18" or "string of length 5" or "valid client ID".

On the other hand, there are concrete test cases, where abstract test cases have specific values, like for instance, the number "17", or the "abcde" string, and "1.234.567-8" which could be said is a valid identifier. These last ones are the ones we can actually execute and that's why they are also called "executable test cases".

It is important for us to make the distinction between these two "levels" as we will be working with them at different stages of the automation process in order to follow a **data-driven testing approach**, which greatly differs from simple scripting.

FOR AUTOMATED TESTS SCRIPTS, *DATA-DRIVEN TESTING* IMPLIES TESTING THE APPLICATION BY USING INFORMATION TAKEN FROM A DATA SOURCE, LIKE A CSV FILE, SPREADSHEET, DATABASE, ETC., INSTEAD OF HAVING THE SAME DATA HARDCODED IN THE SCRIPT.

In other words, we parametrize the test case, allowing it to run with different data. The main goal is to be able to add more test cases by simply adding more lines to the test data file.

In addition, we must consider the test oracle. When a test case is designed, the tester expresses the actions and data to be used in the execution, but what happens with the oracle? How do we determine if the result of the test case is valid or invalid? It is necessary to define the validation actions that permit us to fully capture an oracle capable of determining whether the behavior of the system is correct or incorrect. We have to add sufficient validations in order to reach a verdict while also, step by step, pursuing the goal of avoiding false positives and false negatives.

TEST SUITE DESIGNS

Usually all tools allow us to group test cases, in order for them to be organized and run them all together. The organization can be defined by different criteria, from which we could mention:

- Module or functionality: grouping all test cases that act on the same functionality.
- How critical it is: We could define test cases that must always be run (in every build), given that they are the most important ones. Then another medium level (not as critical), that we run less frequently (or perhaps only selected if changes occur in some particular functionalities) and one of less importance that we would choose to run if there were time to do so (or when a development cycle ends and we want to run all possible tests).

These approaches could even be combined by having a crossed or nested criteria.

Defining dependencies between suites can be highly interesting, given that there are some functionalities that if they fail, they directly invalidate other tests. It makes no sense to waste time by running tests which we know will fail. Meaning, why run them if they will not bring any new information to the table? It's better to stop everything when a problem arises and attack it head on and then run the test again until everything is working properly (this follows the *Jidoka* methodology).

CHAPTER 3: TAKING INTO ACCOUNT IMPORTANT CONSIDERATIONS

With what we have learned so far we can begin our first steps into test automation, but when we start to really get into the thick of it, we will face more daunting situations that are not so easy to fix. In this chapter I will discuss different approaches to help tackle certain problems I have faced several times in the past, borrowing from my favorite projects and past experiences at Abstracta.

START OFF WITH THINGS CLEAR

Over the years of a product's shelf life, we will have to maintain the set of tests we automated. When we have a small set of tests, it isn't difficult to find the test we want to tweak when necessary, but when the set of tests begins to grow, it can get quite messy. It is therefore essential to clearly define the organization and denomination to be used, which in the future, will help us deal with the large set of tests we'll have in a simple manner.

DENOMINATION

One must define a denomination of test cases and folders (or whatever the automation tool provides to organize the tests). Even though this practice is simple, it yields great benefits.

Some style recommendations:

- Use names for test cases in such a way that it is easy to distinguish the ones we run from the ones that are part of the main test cases (also recommended when following a modularization strategy). The test cases we are definitely running, which we could consider as functional cycles, that call upon more varied test cases, could be named *Cycle_XX*, where "XX" will generally refer to the most important entity or action related to the test.
- It is useful to define a structure of folders that allows for separating the general test cases (typically login, menu access, etc.) from the different test case modules. The aim of this is to promote the recycling of test cases designed in such a way that it is easy to include them in other cases.
- On many occasions as well there is a need for temporal test cases, which could be named with a common prefix such as *pru* or *tmp*.

THE WAY YOU DEFINE A DENOMINATION DEPENDS ON YOUR PREFERENCES AND SPECIFIC NEEDS. HAVE THIS CLEAR BEFORE PREPARING SCRIPTS AND BEFORE THE REPOSITORY BEGINS TO GROW IN A DISORGANIZED MANNER.

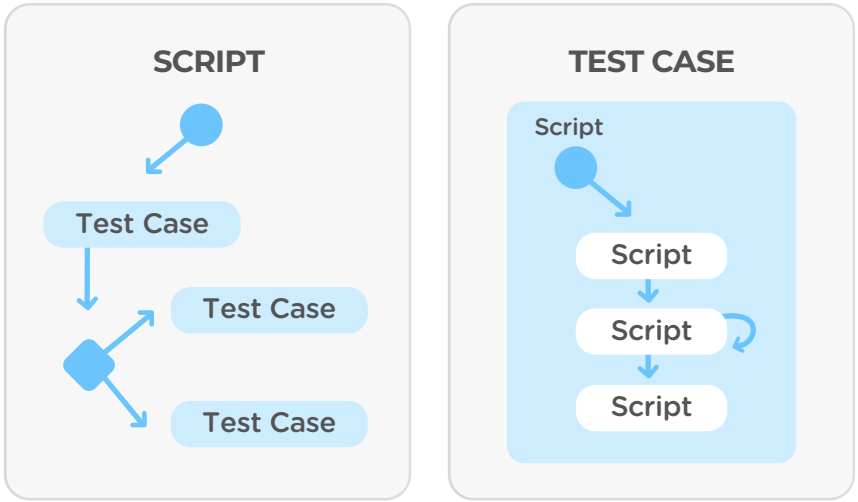
COMMENTS AND DESCRIPTIONS

Every test case and datapool can have a description that roughly describes its objective. In addition, we could include comments that illustrate the different steps to follow for each test case. Inside the datapools, we could also add an extra column to write down comments in which the objective of each concrete data used for the tests is indicated, telling us what it's trying to test.

THE LINK BETWEEN TEST CASES ANDF AUTOMATED SCRIPTS

How should scripts be done with the tool? One for every test case? Could a script be made that tests different test cases at the same time?

Below, both options are represented. On the left we have a script that runs different test cases. When it runs, it might analyze various options upon the data or the system state and according to its evaluation, decide to execute one test case or another. On the right, we could have a test case modularized into different scripts. It would have different smaller test cases that are run by a script that includes and manages them all.



As with anything in software engineering, in this particular instance, it all depends on the test case. Some propose to think of how many logical bifurcations present themselves in the test case. From my point of view, the best way would be to take a modular approach. Meaning, to have different modules (scripts) that carry out different parts of the test and then a script that manages all of those parts. This way, we can reuse the small parts and run different tests that compose them in various ways.

In that case the relationship would be a *test case made of several scripts*.

Advantages

- Maintenance:
  - Easier to maintain
  - Modules can be reused
  - The flow can be changed at different levels
  - The test case script is clearer, as one can see the whole flow going through the "bigger picture," and then dive deeper into the interesting parts

- It's easier to manage the flow of the test case
  - For example, it's easier to make a certain module repeat itself a certain amount of times (fixed or variable). In the typical example of an invoice, if we modularize the part where we enter a line of the invoice with a certain product and quantity, we can make that part execute a certain amount of times, with the aim of testing the invoice with different amounts of lines.

- It's easier to analyze the reports

If we have documentation of the test cases (if they used to be manually executed for instance), a good practice would be to have a matrix that connects all the test cases with the different scripts involved. This allows us to know what to verify when certain requirements change that have an impact on tests and consequently, on some scripts.

An alternative would be designing test cases in a linear manner in case the results are deterministic and only if they have some undefined variability beforehand, add different flows, but the best option is to keep things simple and sequential. A lot of times, coming from a programming background, we tend to make very generic test cases (that cover all cases) and they end up being too complex.

If only one test case is designed to contemplate all options, it will probably be more difficult to comprehend. Therefore, we have to analyze what is being checked in each decision (bifurcation), what is being done with one flow or the other, and so on, unless we are very careful and fill the test case with comments to simplify that analysis. Anyway, a sequential test case with a descriptive name informs us of what it is and what it does.

However, if one day we decide to add a new case, where should we do it? How do we add the bifurcation? How do we handle the data associated with it? If on the other hand, we create a new test case, a sequential one, with a datapool for the case, it rather simplifies that task.

AVOIDING FALSE POSITIVES AND FALSE NEGATIVES

When dealing with automation, one of its most delicate subjects is results that lie, otherwise known as *false positives* and *false negatives*. Those of us who have already automated know this to be an issue and those of you who are about to begin, let me give you fair warning that you will encounter this problem. What can we do to avoid it? What can we do so that the test case does what it is supposed to do? *Doesn't that sound like, testing?*

These definitions come from the medical field:

- **False Positive:** an examination indicates a disease when there is none.
- **False Negative:** an examination indicates everything is normal when in fact the patient is ill.

If one were to translate this to our field, we could say the following:

- False Positive: when a test is executed and despite it running correctly, the test tells us there is an error (that there is a disease). This adds a lot of cost, as the tester will waste time searching for the nonexistent bug.
- False Negative: when the execution of a test shows no faults even though there is a bug in the application. This, as much as the false negative, can be due to an incorrect initial state of the database or problems dealing with the test environment setting.

If we believe that the false positive is a problem due to the extra costs, with a false negative, errors are there but we are not aware of them and we feel at ease! We trust all functionalities are covered and that they are being tested. Therefore, they must not have any mistakes.

We obviously want to avoid results lying to us! The expectation of an automated test case is that its results should be reliable and that we aren't wasting time on checking whether the results are correct or not.

The only choice is to carry out a proactive analysis, checking the quality of our tests and anticipating possible errors. We must actually think about the test and not simply just do a *record and playback*.

To lower the risk of environment or data problems, we should have a controlled environment that is only accessible through automated tests to avoid some major headaches. Moreover, we should check the actual test cases! Because who can assure us they are programmed correctly?

AT THE END OF THE DAY, THE TEST CODE IS CODE AFTER ALL, AND AS SUCH, CAN EXHIBIT FLAWS OR BE IMPROVED. AND WHO BETTER THAN US TESTERS TO TEST IT?

IN SEARCH OF FALSE POSITIVES

If the software is healthy and we don't want it to display any errors, we must make sure the test is testing what it wants to test, which means verifying the starting conditions just as much as the final ones. Meaning, although a test case tries to execute a determined set of actions with certain input data to verify the outgoing data and the final state, it is highly important (especially when the system we are testing uses a database) to make sure the initial state is what we expected it to be.

Therefore, if for example, we are creating an instance of a particular entity in the system, the test should verify if that information already exists before beginning the execution of the actions to be tested because if so, the test will fail (due to duplicate key or similar), but in reality, the problem is not with the system but with the data. We have two options: checking if it exists, and if so, we've already used that information, or we finish off the test by saying the result is "inconclusive" (or are *pass* and *fail* the only possible results for a test?).

If we make sure all the things that could affect our result are in place just as expected, then we will reduce the percentage of false errors.

IN SEARCH OF FALSE NEGATIVES

If the software is "sick," the test must fail! One way of detecting false negatives is to insert errors into the software and verify that the test case finds the mistake. This goes in line with mutation testing. It is very difficult when not working directly with the developer to input the mistakes into the system. It's also quite expensive to prepare every error, compile it and deploy it, and so on, and to verify that the test finds it. In many cases, it can be done by varying the test data or by playing around with different things. For example, if I have a plain text file as input, I can change something in the content of the file in order to force the test to fail and verify that the automated test case finds that error. In a parameterizable application, it could also be achieved by modifying some parameter.

The idea is to verify that the test case realizes the mistake and that's why we try to make it fail with these alterations. Anyway, what we could at least do is think about what happens if the software fails at this point, will this test case notice it, or should we add some other validation?

Both strategies will allow us to have more robust test cases, but keep in mind: would they be more difficult to keep up later? Of course this will not be done to every test case we automate; only to the most critical ones, or the ones really worthwhile, or perhaps the ones we know will stir up trouble for us every now and again.

SYSTEM TESTS THAT INTERACT WITH EXTERNAL SYSTEMS

What happens if my application communicates with other applications through complex mechanisms? What happens if it uses web services exposed to other servers? What happens if my application has a very complex logic? Can I automate more tests in these situations?

Let's imagine the following: A button in the application being tested executes a complex logic, there's communication between several external applications, and a rocket is fired.

The automation tools (at least the ones we are focusing on here) have the objective of reproducing the user interaction with the system, therefore these background complexities *almost* don't matter. Once the user presses a button, the logic being executed due to that action could be simple or complex, but to the tool this is hidden (just as hidden as it is for the user). It doesn't matter if you shoot a rocket or something else, what's *important* to automate is the user interface in this case.

Sometimes the test case requires other actions to be carried out that cannot be done in the browser or the graphical user interface of the system being tested, for example, consulting a database, copying files from a specific place, etc. For these actions the tools generally bring about the possibility to do them by carrying out a special functionality or by programming in a general purpose language.

The fact that an application with a complex logic doesn't add difficulties to the automation process does not mean it doesn't add difficulties at the time of thinking about and designing the tests. Two aspects that can get the most complicated are the data preparation and the simulation of the external services used. Particularly regarding the latter, there are times in which it would be preferable for the system being tested to actually connect to the external service and other times when it would be better to simulate the service and even test the interaction with it. The device that mimics the external service is generally known as *Mock Service*, and there are tools to implement it with ease. For example, in the case that the service is a *web service*, you could consider the SoapUI tool. It has a user friendly interface to create *Mock Services* and to test Web Services as well.

THINKING OF AUTOMATION WHEN PLANNING THE PROJECT

A lot of people still believe that testing is something that should be left for the end of the software development life cycle... if there is time to spare. In reality, it's a task that should be well thought out and planned from the beginning, even before planning development. And more than that, it should be considered part of the same activity: development.

When it comes to automation, these are a few of the tasks you need to plan for:

- Automation
- Maintenance
- Executions
- Verifying and reporting bugs
- Fixing detected bugs

Whether you are in an agile team or not, one must decide when to start automating (from the beginning or after a certain stage in which a stable version of the application is achieved) and consider the upkeep it will incur. This is inevitably linked to the tool we choose and the conveniences it brings.

CHAPTER 4: RUNNING AUTOMATED TESTS

Using tools like *Record* and *Playback* sounds easy but as we've already seen, several matters must be taken into account for the moment before *Playback*. Now we will also see there are some important aspects to consider for the moment of Playback.

MANAGING TEST ENVIRONMENTS

It is of paramount importance to properly manage the test environments, especially if you are integrating the automated tests into your pipeline for CI/CD. For that to happen one must consider many elements that are part of the environment:

- The sources and executables of the application being tested
- The test devices and the data they use
- If the information is related to the database of the system being tested wherewith we would have to manage the outline and the information of the database that corresponds to the test environment

Let's add the complication that we might have different tests to be run with different settings, parameters, etc. So for this we have more than one test environment, or we have one environment and a lot of database *backups*, one per every set of tests. This adds the extra complexity of having to carry out specific maintenance for each *backup* (for example, every time there is a change in the system where the database is modified, it will be necessary to impact every backup with those changes).

But if one of these elements is out of sync with the rest, the tests will likely fail and we would be wasting our resources. It's important that every time a test reports an error that it be due to an actual bug and not because of a false alarm.

HOW TO EXECUTE THE TESTS, WHERE, AND BY WHOM

Now let's discuss more in depth another topic that doesn't have to do with the "technical" side of testing: Planning. It is necessary to plan the executions, but not just that. Ideally the testing would be considered from the beginning (Yes, I am repeating myself, but it needs to be clear!) and if one is to automate, to think about the tasks associated with it from the start.

When to run?

The first thing that comes to mind is as frequently as possible. However, resources may be slim and depending on the quantity of automated tests, the time it takes to run them could be quite long. Don't forget that the goal is to get early feedback on the most risky aspects of the application, so, the decision could be made following this pseudo-algorithm:

If we don't have a lot of tests or they run in a short amount of time, then execute: ALL OF THEM.

If they take too long to execute, select what to run:

- Consider priority based on risk
- Take into account impact analysis (based on the changes of the new version to test)

Know that larger amounts of executions mean you will see a higher return on investment (ROI).

It is not enough to test, we have to correct as well and the time it takes to do so must be considered when planning.

Besides planning *when*, one must think of *whom*. Usually one could aim at having some very separate environments. For example the:

- Development environment (each developer)
- Development integration environment
- Testing environment (within the development team)
- Pre-production environment (testing in customer testing facilities)
- Production environment

The set of tests and the frequency of the same in each of these environments might be different.

For instance, in development, one needs more agility, given that we would want to run the tests more frequently, after every major change, before doing a *commit* in the code repository. For that, it would be convenient to only run the necessary tests. The aim of these tests is to provide the developer with quick feedback.

Once the developer frees his or her module or moves to the consolidation stage, *Integration Tests* would be run. Ideally they would run automatically with your CI engine, maybe at night, so in the morning when the developers arrive, they have a list of possible issues to solve, and feedback from the changes introduced the day before. The less time between changes and the test results, the faster they will fix it (one of the main benefits of Continuous Integration). This would mean preventing things that don't work from moving onto the testing stage. They would be like smoke tests in a way.

Then when the application is passed on to testing, a larger set of regression tests should be run to try to assure that the mistakes that have been reported and have been marked as fixed aren't there in this new version. This set of tests might take longer to run. They don't have to be periodic, but they can adjust to the project's schedule alongside the foreseen release dates.

When the deliverable version of the application is achieved (approved by the testing team), the same is released to the client. The client would generally also test it in a pre-production environment, which should be completely symmetrical in settings as that of production (this is the difference with the development team testing environment). Having an automated set of tests at this point would also add value.

This set of tests could at least be like the one ran at testing and one could even give the client the automated test set along with the released application, providing them more security and confidence by knowing that it was tested prior to its release.

WHAT SKILLS DO I NEED TO AUTOMATE?

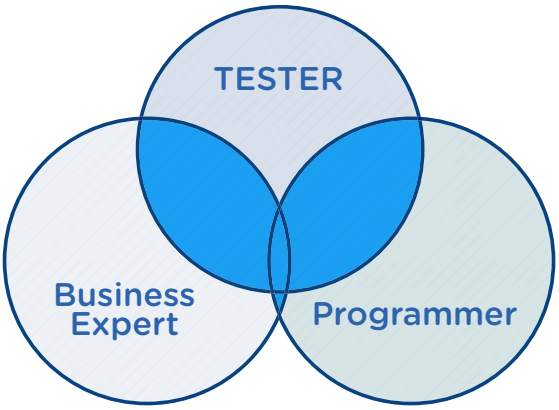
According to testing guru, James Bach, you do not need special conditions to automate. Ideally, the same testers who have been responsible for the traditional functional testing should address the task of automation because they already know the requirements and the business function of the application. This would prevent the automation from falling into the lap of someone who only knows how to use the tool but is unfamiliar with the app. These testers are better suited for the task for several reasons:

- No competition would be generated between manual and automated testing
- It would help ensure the correct choice of which tests to automate
- The automated testing tools could also be of service in generating data for test cases
- Letting the manual testers perform automation would also eliminate any reason for them to fear being replaced

Things you should know about:

- The application and business domain
- The automation tool
- The platform with which you are working (for typical technical problems)
- Testing (techniques for generating test cases)

Each skill will add value in different ways, making our profile move in different areas of knowledge shown in the figure below. Clearly, the closer we come to the center, the more capacity we will have to add value to the development of a product, but you might want to specialize in one of these areas or any special intersection. This is also known as T-shaped skills.



In parallel with the manual work, one should begin training with the tool that will be used as well as read material about automated testing methodology and experiences in general. To begin, a recommended reading besides this ebook is the 4th Edition of the Testing Experience Magazine which focuses on automated testing.

FINALLY, NOTE THAT TEST AUTOMATION IS NOT SOMETHING THAT CAN BE DONE IN ONE'S FREE TIME.

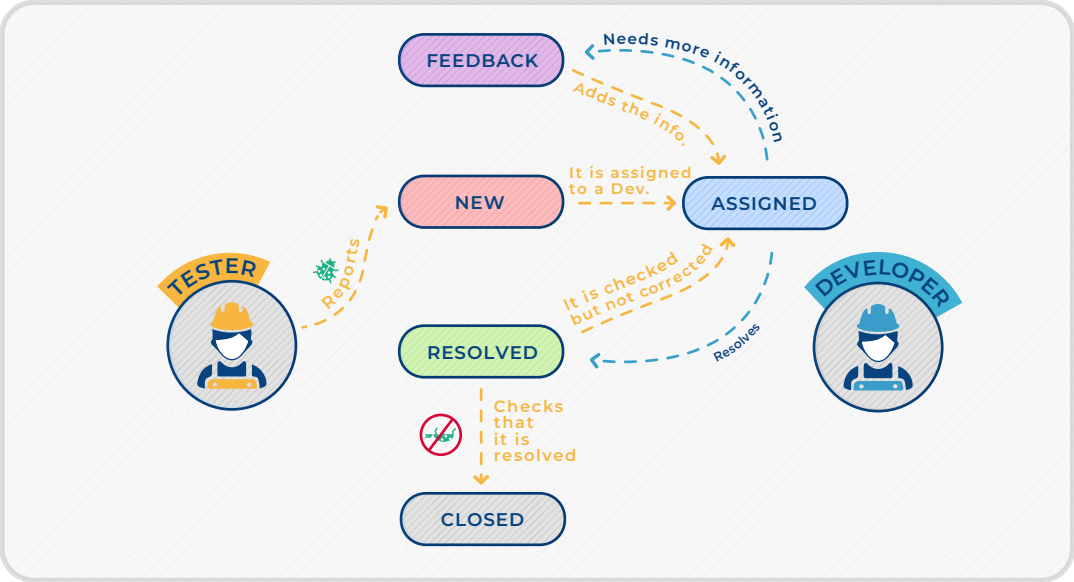
WHAT DO I DO WITH A BUG?

Lastly, I'll comment on incident management in automation. If a report is telling me there was an error, the first step is to determine if it's due to the test cases. One must make sure the error does not lie with the test before reporting it to a developer. Doing so will also improve the relationship between testers and developers.

It's necessary to manually verify the cause of the bug, see if it always happens, see if it had to do with the data, the environment, if it occurs by chance, etc. Afterwards, the tester must figure out what's the easiest way to reproduce the bug so that it's easier for the developer to fix it.

What comes next is reporting it in the incident manager system just like how it's done with bugs that are found in manual test runs.

A LOOK AT THE LIFE CYCLE OF A BUG:



This is a scheme for bug reporting and fixing. There could be a thousand variations of it based on the size of the team, how it is organized, etc. One of the most common problems I have found when working with clients is that once a developer fixes a bug, he or she marks it as resolved, but it is imperative that a tester double check to make sure it was fixed! How smoothly teams manage incidents is often where you can see if the testers and developers feel like they are part of the same team that shares the same goals.

CHAPTER 5: FINAL COMMENTS

Steven Rowe
(Tester at Microsoft)

"In the beginning, testers had a software to test, they would push buttons, invoke functions with different parameter values, and then verify that the behavior was as expected. Then these functions became more complex, each time with more buttons, more sophisticated systems, and testers couldn't handle all of it. Developers had to wait too long for the testers' approval before releasing it for sale. Therefore, the solution was in automated testing, which consists of something as simple as a program that runs the functions with different data, pushes the buttons the same as the testers, and then programmatically verifies if the result is correct."

After reading this ebook, I hope that you would get the sense that we would disagree with this view of automated testing, as testers cannot be replaced by machines!

The above mentioned paragraph belongs to a post from Steve Rowe's blog, which James Bach responded to in his blog, criticizing his opinion. Among other things, at Abstracta, we emphasize this quotation, which we believe sums it up pretty well:

James Bach

"Automation does not do what testers used to do, unless one ignores most things a tester really does. Automated testing is useful for extending the reach of the testers work, not to replace it."

From my humble opinion, I agree more with James, as I don't believe automation can replace a tester's job, but it can make it better and more encompassing.

Not everything should be automated, and we shouldn't try replacing manual testing for automatic, for there are things which cannot be automated and sometimes it is easier to manually execute something than to automate it. In fact, if all executions could be run manually, it would probably be much better in the sense that by executing manually, other things can be found. Do remember that automated tests check but don't test. The problem is that it takes more time to do it manually and that is why it's convenient to automate what is worth automating.

On behalf of Abstracta, thanks for reading our test automation ebook! We hope it helps you in your endeavor to automate functional testing. Please feel free to shoot me an email at federico@abstracta.us if you have any questions regarding the topics raised in this ebook or just want to say hello!

ADDITIONAL RESOURCES:

Here are some blogs, papers, sites etc. that I highly recommend to read up on automation and testing.

- Read James Bach's blog. All of it.
- Read the Abstracta Software Testing Blog.
- Read my Ultimate Guide to Continuous Testing.
- Visit Abstracta's Resources Page for webinars, infographics, case studies, and more.
- Last but not least, check out my Spanish testing blog.

ABOUT THE AUTHOR

Federico Toledo Ph.D,
CPIO and Co-founder of Abstracta

Federico has over 10 years of experience in consulting, research and testing related to the area of development as well as more than 7 years of teaching experience at various universities. He has a Bachelor's degree in computer engineering and graduated cum laude from University of Castilla-La Mancha, Spain with a PhD in testing. While receiving his doctorate, he was a part of the eminent Alarcos Research Group which received the 2008 Quality Award in R&D, awarded by the Association of engineers of Castilla-La Mancha and the Federation of Enterprises of Technology. He has published scholarly articles and is frequently invited to participate in international conferences and seminars. He published "Introduction to Information Systems Testing," one of the first books in Spanish on testing with a practical approach.

CONTRIBUTORS

Germán da Cunda,
Lead Graphic Designer, Abstracta

Germán has completed a degree in graphic design in visual communication from the Technology University of Uruguay and a degree in web design from the ORT University of Uruguay. He has attended numerous courses and workshops related to the creative and graphic areas. He likes to solve visual communication problems in an efficient and creative manner. He is detailed oriented and minimalistic in his approach.

Kalei White,
CMO of Abstracta

Kalei graduated cum laude from California Polytechnic State University San Luis Obispo with a B.S. in Business Administration. She has several years of marketing experience from working for software companies to marketing agencies. She enjoys analyzing data, creating content, and disseminating it throughout the software testing community. She manages Abstracta's digital marketing activities.

ABOUT ABSTRACTA

Formed by PhDs and passionate computer engineers, Abstracta is a world leader in quality assurance and testing focused on improving the performance of software applications. With offices in Silicon Valley and Latin America, we have over 50 years of combined expertise working not only with leading-edge proprietary and open source testing tools, but developing specialized tools for financial, retail and technology companies such as BBVA Financial Group, Verifone, GeneXus software and the largest retail bookseller in the United States. Our main products are: GXtest (used in more than 15 countries worldwide) and Monkop, a first-of-its-kind tool for mobile testing.

abstracta

We are specialists in:

- Test automation
- Performance testing
- Test case design, execution, and reporting
- Functional testing
- Mobile testing, etc.
- Corporate and individual testing training

We are comfortable working within a variety of software development environments such as Agile, continuous delivery/continuous integration, waterfall, etc. Our testers are based in Uruguay, making our services convenient (minimal time difference), cost-effective and friendly (we share your business culture and speak your language). Learn more about our complete services at www.abstracta.us.

READY TO GET STARTED WITH TEST AUTOMATION?

SCHEDULE A MEETING WITH OUR QUALITY ENGINEERS TODAY!

Contact Us

I HOPE THIS EBOOK WAS USEFUL FOR YOU.

PLEASE FEEL FREE TO SHARE YOUR THOUGHTS AND REACTIONS!