abstracta

THE ULTIMATE GUIDE TO CONTINUOUS TESTING

Everything you need to know to shift-left testing and reach testing maturity

CONTINUOUS TESTING: TODAY'S GOLDEN STANDARD

At Abstracta, we believe that Agile development, along with the practices that it promotes such as continuous integration, continuous testing and continuous delivery (CI/CT/CD), is the key to remaining competitive in today's technological landscape.

For an Agile environment to flourish in your organization, testing needs to happen earlier on in development than it does in traditional development environments like waterfall. We call this **"shift-left testing"** and it's imperative for Agile teams to truly succeed.

There are several software quality assurance activities to focus on that will help you in your efforts to reach an efficient continuous integration environment, allowing for the quality checks you want to have in each build.

In this guide, we will tackle the various areas in which we can group these activities so you may have a clear picture of what your team must work on in order to make progress in your testing maturity and, ultimately, reach continuous testing!

Let's Begin!

TABLE OF CONTENTS

ABOUT THE AUTHOR

Federico Toledo is a co-founder and director of Abstracta and holds a PhD in Computer Science from UCLM, Spain. With over 10 years of experience in quality engineering, he's helped many companies to successfully improve their application quality. He is dedicated to testing education, having written one of the first books in Spanish on testing and formed Abstracta Academy. He is also a co-organizer of TestingUY, the biggest testing conference in Latin America.

INTRODUCTION

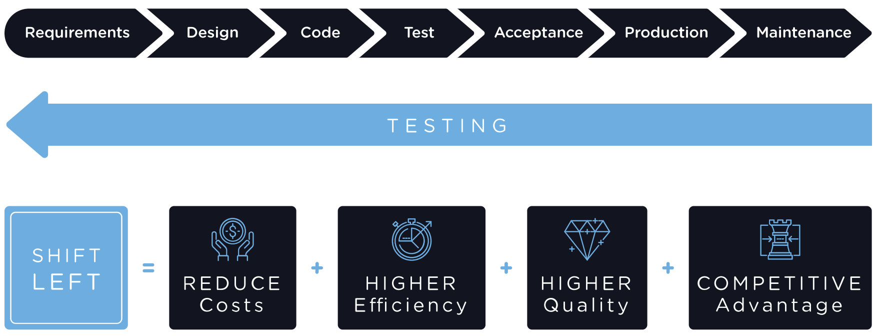
"Agility basically facilitates competitiveness. To compete in today's environment, you must act and react fast, otherwise your competition will simply beat you to it. Today, the barrier to compete is minimal, and the only way to defend one's stature is by innovating in short iterations and basically meaning adopting Agile."

— Alon Girmonsky, CEO, BlazeMeter

When you shift testing left, among many other benefits, you can achieve **continuous testing**, executing automated checks as part of the software delivery pipeline to obtain immediate feedback on the business risks associated with a software release candidate. Continuous testing also involves early and frequent testing, proper code management, and adjusting the various quality-related tasks over time to maximize the efficiency of the whole process.

Some of the key benefits of continuous testing include:

- Lower the cost of development
- Reduce waste
- Reduce risk upon release
- Increase system reliability
- Release to production faster
- Compete more fiercely in the marketplace



Many teams today are trying to build or refine their continuous testing machine. We believe that in order to improve the results of software production, it's necessary to consider three fundamental pillars that are closely linked together: processes, tools and people. Teams can improve by bettering their processes, but the tools and the team members must also adapt in order for the new and improved processes to stick.

PEOPLE
TECHNOLOGY
PROCESSES

For example, within Agile, teams may find themselves in frameworks like Scrum that create a **process**, for which they will find themselves using various types of **tools** for everything from communication to task management and there will typically be great focus on the motivation and commitment of the **team**.

All aspects of Agile approaches are well designed to be adaptable to change, which is its main focus. The idea is to assume that what the customer needs is not fixed nor established in a contract. Thus, it is essential to work on constant adaptations in a way that does not cause the project costs to skyrocket or become unmanageable.

What is needed to adapt to change and yet maintain a high level of quality?

For any team, the most typical problem that arises when introducing changes in a system is fear: fear of breaking something that was already working before and not realizing it until after it has reached the customer's hands. When a user is given a new version of the application, there is nothing worse than finding that what always used to work no longer does.

To address this problem, **teams must be able to detect any errors of any kind as soon as possible**. When speaking of errors, we are referring to any kind of error or bug: a miscalculation, performance problems, security vulnerability, aspects of usability, maintainability problems (e.g., code quality factors) or anything that may inconvenience a user. And what we mean by "as soon as possible" is as soon after when the error was inserted into the system as possible.

Thus, important practices like continuous testing, which goes hand in hand with continuous integration (CI), emerge and gain importance for Agile teams.

Basically, continuous integration proposes building and testing the solution frequently, building a potentially customer-shippable product with a frequency that depends on each team, varying from one new version after each commit to one every day or week. Continuous integration tools allow for defining a sequence of tasks to be performed automatically, among which typically include:

- Updating code to the latest version
- Building the solution
- Automated checks
- Code quality checks
- Security checks

And the list can go on with hundreds of integrations and tests that one may wish to add.

There are dozens of tools to support these automated systems, some as old as Ant, Maven and MSbuild, configuration management tools as Chef, Puppet or Ansible, to popular CI tools such as Jenkins, CruiseControl, Bamboo and TeamCity. And if that were not enough, there are also cloud solutions like Amazon Pipeline Code, TravisCI, CircleCI, Codeship and Microsoft's Visual Studio Team Services (to name a few).

So, all one has to do to have continuous testing and continuous integration is simply implement one of these CI tools and then quality will be baked into everything. Right?

Wrong.

The problem is that many teams fall into this very fallacy and overlook many preconditions for continuous testing and CI that must be met beforehand, resulting in failure. Some of the preconditions include the proper handling of code version management, incident management, data and environment management, automated checks at different testing levels (unit, build tests, API, UI), performance tests, internal code quality and so on.

Fortunately, this guide will help you navigate through several of the hoops your test team has to jump through on your path to continuous testing (the highest level of testing maturity according to our maturity model) all the while igniting the shift from traditional quality assurance to quality engineering.

THE SOURCE CODE

This is one of the most precious assets of any development team because in it lies all of the work. In fact, it is the only artifact that every project will always need to build. Many teams don't document it, nor make certain types of tests for it, **but without code, there is no software**. So, code is the one thing that software development teams will always have. This provides the guideline that it's a fundamental area to pay attention to in order to control quality and reduce risks and costs, no matter what the organization's goals and context may be.

When considering source code, we are referring to several things, but we're mainly referring to code *management* and *code quality*. Both aspects should be considered in order to achieve a good continuous testing approach.

Code Management

With code being such an important asset, a risk that's essential to have under control is making sure to properly manage it and any amendments made thereto. There are specific tools to do this that are made for working with a centralized repository where all team members can reliably store it.

Today's tools have evolved in terms of task focus, going from simple schemes like CVS to more sophisticated and flexible ones like Git. What they allow, some to a lesser or greater extent, is to have the code in a centralized location. Each developer can submit code, consult it, retrieve previous versions (which is useful when your code is not working properly and you need to go back to the previous working version), add comments to each "commit" made (e.g., comments indicating why one decided to send certain code to the repository) and much more.

These tools are also accompanied by a methodology. In particular, this is reflected in how the branches of the versions are administered (approved, removed, merged, etc.), which allows for different people to work on different versions of the code. There are those that maintain a branch for each customer (if each has different code), or a branch for larger versions of code, or at a finer level, where a new branch is opened for each functionality or fix someone makes.

Code Quality

On the other hand, it's important to pay attention to the code quality for promoting maintainability. Code quality is typically an internal attribute of quality, as it's not visible to the user. It's possible that, even though the user perceives a software to be of high quality (easy to use, fast, without critical bugs, etc.), it can actually have troubling code quality problems. But since users can't see nor analyze the code themselves, *they would never know*. Conversely, quality factors like performance, reliability, and usability, among others, are external, meaning, *users do perceive them*.

But, there may come a moment when internal quality factors transform into external ones. Take, for example, what happens when a change is made to the system in response to a user request. If the code quality is so poor that it takes several weeks to modify the system, then there is a huge maintainability problem which not only affects the development team, but also the user. The longer it takes for the change to be made due to the bad code, the worse the user satisfaction will be.

Maintainability depends on many aspects. There is a tired phrase that goes, "Code is written once and read many times," meaning, it must be easy to read. But this doesn't only include following the well-known conventions like when to use capitals and when to use lower case, or using the correct indentation, but it's also necessary to keep the code complexity, coupling, size of the classes, amount of parameters, and many more factors in mind.

Years ago the term "technical debt" arose for easily explaining the problem of code maintainability to someone without programming skills. It makes an analogy with the more well-known concept of financial debt. It essentially says that if one schedules a certain functionality, and does it hurriedly due to lack of time, without complying with the team's "definition of done," (for example, skipping documentation by putting comments in the code, not meeting standards and coding conventions nor implementing unit testing) then at that very moment, "debt" accumulates. The system is owed whatever it may be that was hastily foregone, for example, unit tests, putting them off for when time allows.

What's the problem with that? Besides repaying the debt, "interest" will be added on so to speak, since it's likely to take longer and more energy to fix things after some time has passed than doing it right in the moment.

Each maintainability problem generates technical debt and *someone* always pays: sometimes just the developer, or worse, the user who ends up suffering from the poor quality of the system.

Some very common defects in code quality are:

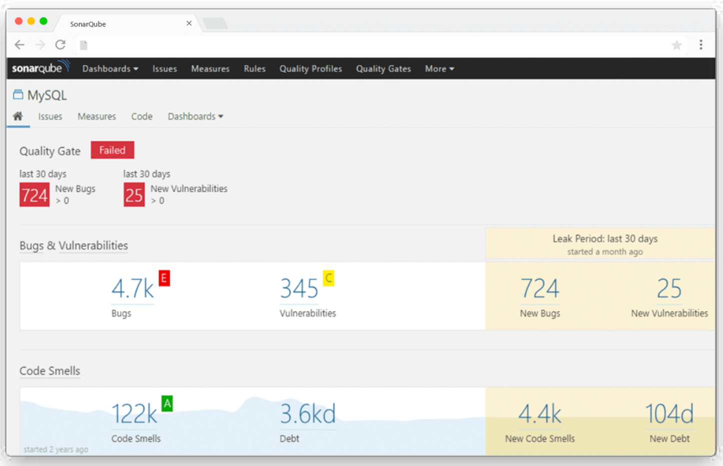
- Duplicate code
- Lack of unit testing, lack of comments
- Spaghetti code, cyclomatic complexity, high coupling
- Methods that are too long
- No adaptation to standards and coding conventions
- Known security vulnerabilities

Modern development environments help immensely with some of these problems such as Visual Studio or Eclipse (to name a few popular ones), to the point that if some of the restrictions are not met, the code will not compile. At any rate, there are many things that can be analyzed with tools that were specifically made for analyzing code quality. Today, the most popular tool for this purpose is SonarQube, but there are several others such as PMD, Kiuwan and CheckStyle.

These tools perform a static study of code and are white-box, that is, they analyze the code without executing it. They check for the common defects mentioned above, such as duplicate code or cyclomatic complexity.

Something very impressive is that SonarQube and tools alike are able to indicate what exactly the technical debt is, calculating how many minutes it would take to resolve each issue it detects. For example, the following image is shown publicly on the SonarQube site as the result for MySQL. One can see that the software has a debt of 36,000 days, which means 36,000 days are needed to resolve all the detected incidents.

This much debt is not what anyone wants for their software!



While it's great that SonarQube can provide information as to where the technical debt is, it is also important to know *how* to pay it. But, just sorting these issues by priority isn't enough. **The key here is finding the code that will give the best ROI if improved**. It's necessary to cross the information of the issues and their severity with the number of commits that there are in each class. In doing so, efforts can be focused on improving the maintainability of those classes that are modified more often. However, if there are maintainability problems of classes that are seldom touched, but they function properly, it's better not to tackle them. As the saying goes, "If it ain't broke, don't fix it."

To achieve continuous testing, a CI environment must be built, and to do so, it's clear that code needs to be accessible in a well-organized repository, with various quality checks on the code being made, utilizing something like SonarQube.

ENVIRONMENTS AND INFRASTRUCTURE

Presumably, something everyone can agree on is that **there** are at least three different environments to work within: one for development, where the code is compiled and tested locally upon each modification, one for testers to test the "closed" versions that are ready to test, and the production environment, where the system code that users face is found.

It never occurs to *anyone* to go from the development environment straight to the production environment, right? Because that would be plain crazy...

"Everybody has a testing environment. Some people are lucky enough to have a totally separate environment to run production in."

Michael Stahnke

Unfortunately, some teams do just that, run tests in the wrong environment. There are many that don't have even the three basic environments to carry out these practices, which causes many consequences. For example, this is when one might hear the excuse from developers, "Well, it works on my computer," when something isn't working for the user (when it matters most).

To really test software before it reaches the user, it is essential to have a specific environment for it, and, sometimes even that is not enough. Especially for automated checks, it is essential to run these tests with the data for them in isolation.
Furthermore, it's advisable to have a separate environment to ensure that they will not be modified (if they are modified, there is a risk of obtaining false positives).

It's fundamental to manage the test environment well, considering many of its different elements:

- The source files and executables of the application
- The test devices and the data they use
- In turn, the data are related to the database system in each environment, so it's necessary to manage the schema and data from the correspon-ding database

If there are different tests to be performed with diffe-rent configurations, parameterizations, etc., more than one test environment will be needed, or an environment and many database backups, one for each set of tests. This means that one has to perform specific maintenance for each backup (for instance, whenever there are changes in the system in which the database is modified, one must hit each backup with these changes).

But if one of these elements is not in tune with the rest, the tests will probably fail and lead to inefficien-cy. It's crucial to be sure that each time a test reports an error, it is because there really is a bug instead of it producing a false alarm.

To solve all these problems at the environment level, there are typically special roles within a team. An immensely popular industry trend now is to have DevOps. There is a slight controversy over the term because the original concept was related to the cultu-re of a team, and not designated to individual roles. But, today you will see companies hiring a "DevOps Engineer." According to Puppet Labs, teams that employ DevOps deliver 30 times more frequently, have 60 times fewer failures and recover 160 times faster.

Basically, the role of DevOps is to combine the vision of operations management (infrastructure and environment), development, and business. Connec-ting these areas makes it possible to quickly and efficiently resolve all the needs of the team while placing the focus on the customer and the end user.

As for tools, DevOps usually takes advantage of the facilities of virtual machines as well as other more advanced ones like Docker and Kubernetes. When it comes to the variety of mobile devices needed for testing, there are several solutions for avoiding buying and maintaining dozens of devices every month, from device laboratories to shared solutions in the Cloud that offer access to devices on a pay as you go basis.

Thus, having multiple, properly managed environ-ments helps avoid questions like, "Are we sure we're testing on the latest version?" It is also fundamental to have all the environments that testers need, which may include providing access to different operating systems with different configurations, different brow-ser versions, or even having access to various mobile devices. Due to the vast array of devices on the market and the fragmentation within Android and nowadays even with iOS, it's necessary to test on as many of the most relevant devices as possible.

Concerning mobile device coverage, a strategy that Abstracta has seen great results with is "cross-covera-ge over time." Whereas testing on a variety of devices requires more runtime, the cross-coverage strategy aims to improve coverage over time. This strategy simply proposes organizing executions in such a way so that they are not all ran in each execution cycle, but rather ran alternately, improving coverage after many cycles. The following images exemplify this strategy:

	S5	Nexus 4	HTC One
Test 1	%		
Test 2		%	
Test 3			%

Execution 1

	S5	Nexus 4	HTC One
Test 1		%	
Test 2			%
Test 3	%		

Execution 2

	S5	Nexus 4	HTC One
Test 1			%
Test 2	%		
Test 3		%	

Execution 3

Whereupon, in the third execution cycle full coverage is reached:

	S5	Nexus 4	HTC One
Test 1	%	%	%
Test 2	%	%	%
Test 3	%	%	%

This does not guarantee 100% coverage in each run, but by managing to toggle the test runs, greater coverage is successively achieved.

This same strategy applies to web browsers, parameters, or many more variables with which testers have to "play".

To achieve testing maturity, the necessary environments for test execution are required to ensure that there aren't any additional problems besides those that one hopes to uncover through testing.

BUG AND INCIDENT MANAGEMENT

Incident management is a basic point of efficiency within a development team. From how bugs are managed, it's easy to tell whether or not an organization's testers and developers feel as if they are a part of the same team, with the same goals. This assertion is not only aimed at developers, urging that they collaborate better with testers, but also at testers. Testers should avoid complaining (as they sometimes do) when a bug they reported doesn't get fixed and must understand that not everything needs to be fixed, depending on the team's global vision.

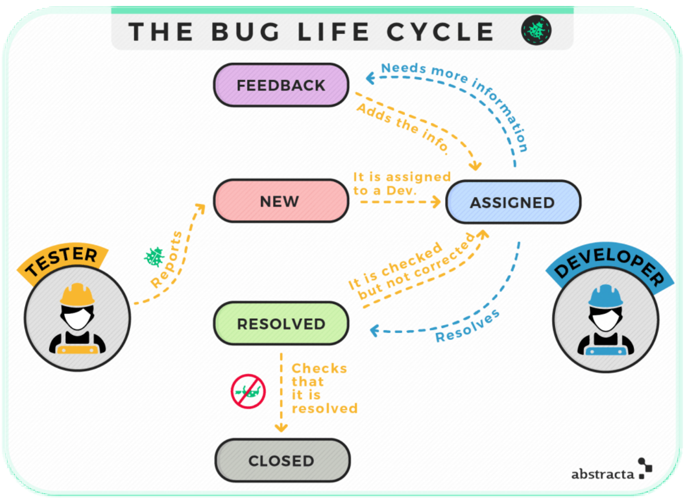
One poor incident management practice that we always find irksome at Abstracta is that with the hundreds of tools available for incident management, some teams still choose email as the designated channel for reporting bugs. Or worse, not using anything at all, and the tester simply taps the developer on the shoulder after finding a bug.

What's the problem with reporting bugs without recording them with an adequate tool?

Basically, it's impossible to follow up, keep a record, know the status of each incident (if it was already solved, verified, etc.), or even, if there's a team of testers, have it clear what things were already reported by another tester that shouldn't be reported again.

Of course, even when using an incident management tool, there will still be a need to speak face to face, but they do aid in adding clarification. For example, one can comment on and show directly, in the moment, what happened and thus communicate more effectively.

The following scheme summarizes a possible life cycle of a bug (there could be a thousand variations of this according to the team's structure, size, etc).



One of the most typical mistakes in incident management that our team has seen while working with different companies across traditional and Agile environments has to do with the moment when a bug is deemed "fixed" and the developer marks it as closed.

It is important to remember that the tester must double check that a bug has been properly fixed before closing it!
First, the tester reports it and then the developer fixes it, but the person who should ultimately determine if it has been resolved is the tester that reported it in the first place. What if the developer thinks it's gone, but upon further testing, it is still there? What if a new bug has been created in its place? A tester should always go back and check that the bug no longer exists in order to minimize risk.

There are several other valid variations of this scheme. For example, there could be someone with the responsibility of reviewing the incidents that come in and deciding which ones to fix and which ones not to fix. This situation could work within the flow above, whether it be the tester or the developer who marks a bug as "resolved," "won't fix," or through a dialogue (via the tool or a different medium), both the tester and developer could decide together what is the appropriate decision, showing mutual agreement between the two.

At this point you may be wondering, "Why is this important for continuous testing?" As stated before, this is the main point of interaction between testers and developers, whether they are under the same roof, or in distant countries. Moreover, there are tools like Jira that allow traceability among the issues reported in the tool and code, which, in order to make the most of these features, everything must be well mechanized.

The most mature testing teams have a perfectly oiled incident resolution machine, (using a designated incident management tool) as the greatest interaction between testers and developers is in the back and forth of these mechanisms.

TEST MANAGEMENT

Quality to software is not as sugar is to coffee.

One does not simply add "two teaspoons" of quality at the end of development and suddenly have a sweet, high quality product. If testing is left for the end, it means there is only time for the checks that merely verify that the software works. Many times, Gantt diagrams show "testing" as a task to perform in the last two days, but what if someone were to find grave errors in that small window of time? Scott Barber says something about this, specifically related to performance tests, but it could be extrapolated to testing in general:

"Performing the tests at the end of a project is like asking for a blood test when the patient is already dead."

Testing and coding should be considered two tasks of the same activity, and should start together. In fact, testing should begin even before anything else, so when the time comes to start coding, it's already been decided how to go about testing. This is especially useful for preventing errors, not just looking for them.

Speaking of test management, it's closely related to project management because testing is highly connected to several areas, therefore everything must be planned together in a well-coordinated manner.

Some typical questions to ask to know if a team is managing testing proactively (or if at all) could be:

- What are the acceptance criteria? When will we give the nod to deliver the new product feature to the customer?
- How well are we testing the software? To answer this, are we using some kind of metric for coverage or do we really have no idea?
- Who tests what?
- How much testing do we still need? How's it all going?
- What risks are there and which one is the most important to mitigate next?
- Is the product we are building of quality? Does it meet the customer's needs?
- Are we considering all aspects of product quality that really matter to the user and the business (performance, reliability, usability, security, etc.)?

The list of issues related to risk management and a team's knowledge of the product quality could go on and on.

In continuous testing, testing activities must be considered from the first day of the project and thereafter. Testing shouldn't be regarded as something done in case there's spare time, or in isolation. It should be planned and executed deliberately, in order to meet business objectives.

FUNCTIONAL TESTING

Functional testing is focused on the functional aspect of the software (Surprise!).

Mainly, it asks the question, "Is the system buggy?" In other words, it focuses on how well the software works according to the design specifications and having all related risks under control. Functional testing checks the core functions, test input, menu functions, installation and setup on localized machines, etc.

As mentioned earlier, especially in functional testing, it is important that teams know if they know what it is they're testing. Which aspects have already been tested and which are missing? How thoroughly are they tested?

From our point of view, there are three major ways of doing functional testing:

- **Ad-hoc:** Ad-hoc is equivalent to asking anyone to "test the system for a while". This is typical of those who see testing as something like having the software in front of someone who makes random clicks, observing "how it is." There's no element of control, it's impossible to trace, and there's no way to know how well or badly it is going. Testing is for obtaining information about quality and this does not give us said information, so we'd never recommend this "testing" approach.

- **Scripted Testing:** In scripted testing, first one wonders, "What is it that I want to test?" Then one documents it and runs the tests according to the document, recording what went right and wrong. **The star player in this game plan is the "test case."** The two stages, the design stage and the execution stage, are so removed from each other that they could be completed by two people with differing skills. A business expert who is knowledgeable in designing test cases could create a spreadsheet (or tool) with all the cases that he or she is interested in testing. Then another person who may not have as much experience could take that documentation and execute it step by step, indicating the results of each test. In doing this, it is ideal to plan, leave a record of everything that was done, organize the team and analyze how deeply everything was tested. This method is suitable for regression testing, for passing on the knowledge of what things were tested to another person, and for documenting the expected behavior of the system.

- **Exploratory Testing:** With the advent of Agile methodologies and the focus on adaptation to change, **test cases become less relevant.** It's not always ideal to wait around to have documents handed over that indicate what to test and what is the expected result in order to convey an idea of how well or poorly the system works. That's why the exploratory testing approach exists, where the focus is on designing and running tests simultaneously, and in doing so, becoming familiar with and learning more about the system. **The focus is on finding errors and risks as soon as possible.** Furthermore, one can more easily adapt to change either within the application or the context. Because exploratory testing eliminates the need for keeping test case documents, it gives way for increased flexibility. Every action we take is based on the result we got, deciding what to do (and designing the test) on the fly. But beware, this is not ad-hoc because there is a well defined methodology to follow this scheme that allows us to organize, track, control, and manage the whole process. It aims to improve from cycle to cycle rather than run the same tests repeatedly in each cycle (as this is tedious, error-prone, and is better than just running an automated check).

Test cases are often reported in spreadsheets, or better, in specific tools meant for this purpose. The more Agile alternatives to test cases are test sessions, revision checklists and mind-maps to record test ideas rather than having a step-by-step list of actions to test. In any of the key strategies, it is fundamental to have some idea of what the whole is (total test cases, total functionality to be tested, user stories to be covered, etc.), and how to advance. This is what signifies the coverage that is being obtained.

Of course, it is necessary to prioritize this according to the levels of importance and risk to the business and users of each aspect.

To achieve testing maturity and especially continuous testing, it must be clear what is being tested, how, and how well. We recommend either scripted or exploratory testing but never ad-hoc testing.

AUTOMATED CHECKS

Automated checks consist of a machine that executes checks or test cases automatically, by reading its specification in some way which could be scripts in a general-purpose programming language or one that's tool-specific, from spreadsheets, models, etc. The goal of automating is to increase testers' "bandwidth"; by automating certain repetitive processes, testers can devote themselves to other activities.

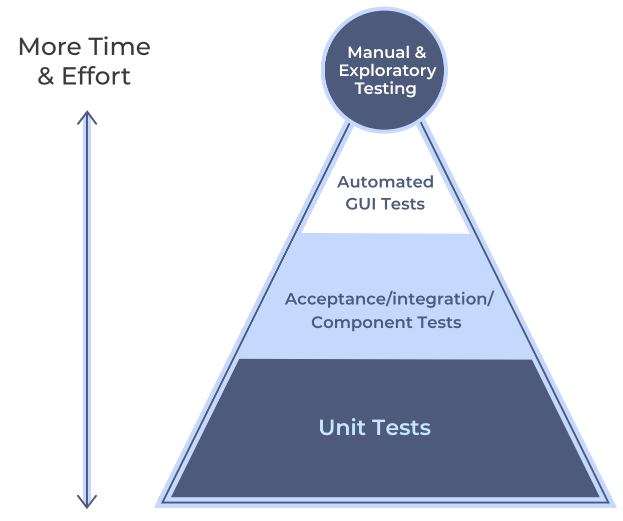
Here are just some of the benefits of automation:

- Run more tests in less time, speeding up time to market and increasing coverage
- Image enhancement, increased user confidence
- Capability of multi-platform execution
- Evaluation of application performance in different versions and over time
- Systematic execution, always test the same thing and in the same way, without losing any verification step
- Earlier detection of errors leads to lower correction costs
- Enhance tester motivation, by freeing up time for more challenging pursuits
- **Facilitation of continuous integration**

One of the desired objectives of automation is **to receive feedback on the status of the software's quality as soon as possible,** and reduce costs not only associated with testing but also development.

It is well known that **automating chaos brings faster chaos.** For the success of this activity, it's essential to properly select which cases to automate at each level and pick the ones that promise a higher return on investment.

The typical problem is that what comes to most people's mind when they think of automation is automating the actions of the user at the graphical interface level, but it's neither the only nor the best option. To understand this, take a look at the automation pyramid by Mike Cohn.


More Time & Effort
Higher ROI
Manual & Exploratory Testing
Automated GUI Tests
Acceptance/integration/ Component Tests
Unit Tests

Ideal Test Automation Pyramid

Cohn's pyramid establishes that there are various levels of checks, indicating to which degree they should be automated. The ideal situation would be to have:

- Many automated unit tests/checks during development since it's a primary point for detecting failures. If a feature fails at this point, tests/checks could fail at the subsequent levels: integration, API, etc.
- Some tests/checks at the API level and integration of components and services, which are the most stable candidates for automation
- Less automated GUI tests/checks as they are harder to maintain, slower than others in execution, and dependent on many other components

Performing GUI tests/checks lends to a greater degree of tranquility since they check the functionality end-to-end, but it's not advisable to aim for just having this one kind of automated checks, nor for it to be the majority of the test-set.

For reference, Google claims to have 70% of its automated checks at the unit level, 20% at the API level and only 10% at the GUI level.

The objective of this scheme is for there to come a time when greater test coverage is increasingly achieved, while investing the same amount of resources.

There is a very interesting problem that occurs at this level. The design and programming of unit test cases has always been a thorn in the software developer's side. Unit testing is a fundamental step for adding a piece of code to the system, but there isn't always enough time, resources, nor the will to do it. While this level of testing is recognized as a good practice to improve code quality (and avoid technical debt), it is also true that often when designing, preparing and planning for completing a programming task, things that are not considered absolutely fundamental are left out, and so, unit tests may fall by the wayside. At Abstracta, we strongly recommend not leaving them out.

Maybe this is the deeper problem: unit testing is not considered to be part of development and ends up being regarded as an optional, support activity.

Clearly, unit automated checks are extremely helpful for a continuous integration scheme in which errors are identified as soon as possible.

Furthermore, it is essential to define a good strategy for automated checks following Mike Cohn's pyramid: a strong base in unit testing, some tests at the service level and only the most critical at the graphical interface level. It's important to always consider the maintainability of the tests in order to sustain a good cost-benefit ratio.

Automated checks are the most important tests at the functional level. Truth be told, it is not possible to achieve continuous integration without them, so teams must seek to execute them frequently.

PERFORMANCE TESTING

Performance tests are for simulating the load on the system under test to analyze its performance (response times and insights usage) to find bottlenecks and opportunities for improvement.

There are specific tools for simulation which automate actions that generate this load, for example, interactions between the user and the server. In order to simulate many users with little testing infrastructure, interactions are automated at the protocol level, which makes automation more complex (as for the necessary prep work) than automated scripts at the graphical interface level.

Two approaches to performance tests can be distinguished: testing early in development (testing the performance of units, components or services) and testing before going into production (in acceptance testing mode). The most important takeaway here is that **both approaches are essential.** It is necessary to simulate the expected load before going live, but not everything should be left until the last minute, since, if there are problems, they will surely be more complex to solve. Moreover, testing each component frequently reduces the cost of corrections, but there's no guarantee that everything will work properly when integrated and installed on a server and under the expected load.

Here DevOps are needed, as they are the ones that are able to analyze the different components of the application, operating system, databases, etc., and can handle different monitoring tools to analyze where there might be a bottleneck, being able to adjust any settings as necessary. It is also imperative to involve developers with these tasks since automation, a programming task, is needed and often the improvements that must be made are at the SQLs level, data schema, or at a logic level, algorithms, code, etc.

To execute automated performance tests frequently, an important problem is figuring out how to make sure the test scripts are maintainable. This happens because automation is done at the protocol level, and in the case of web systems, for example, it's at the HTTP level, making the scripts highly susceptible to changes in the application. A couple of the tools that have emerged to overcome this problem that we use at Abstracta are Taurus and Gatling. Although they have different approaches, both handle simple scripting language and seek to reduce their complexity. For instance, Gatling applies test design patterns like Page Object, which can reduce the impact of changes, increasing maintainability.

It goes to show that before selecting a tool, it is extremely important to define the objectives of the performance tests, in order to choose the one that best addresses your needs and challenges. Each tool comes with its own advantages and disadvantages and features that compensate for different things.

For more on performance testing tools and approaches, visit our performance engineering blog.

Consider performance testing for the acceptance of a product, simulating the expected production load, as well as accompanying the whole development process with unit performance tests to facilitate CI.

SECURITY TESTING

Security issues are usually the most infamous of all, since they commonly involve economic losses, credit card theft, the release of private data, etc., bringing about negative press and devastating business consequences.

One recent real world example that no one can forget is the massive Equifax data breach in July 2017 in which 99% of its customers' (146 million people) social security numbers were exposed. The company revealed that it had known about the security hole since March of the same year, yet failed to protect its customers' highly sensitive personal information. As a consequence, by September 2017, the company lost $4 billion.

And, breaches don't only occur within giant corporations like Equifax or the financial sector, but also in healthcare, retail, education, and government, among others. The number of U.S. data breach incidents tracked in 2017 hit a new record high of 1,579 breaches, according to the 2017 Data Breach Year-End Review released by the Identity Theft Resource Center® (ITRC) and CyberScout®.

Hence, why it is so important to keep security testing in mind!

The OWASP group provides many good guides as well as tools that allow checks to verify the typical security problems, such as cross site scripting, injection, known vulnerabilities, etc.

Each organization's security risk will be different. It is important to determine the potential impact of a security breach on your organization in order to assess how much time and resources should be devoted to this area of quality. The more critical the security of your application, the more mature your testing will be if you take the proper measures to prepare for a breach.

Having at least some basic security checks running periodically allows teams to consider this aspect of quality and over time, improve their set of controls.

USABILITY TESTING

If a product is being built according to the specifications of its customers, but without considering the context, how they are going to use it, and other necessities related to usability and UX (user experience), the users still might not fall in love with it. (Gasp!)

There are specific experiments (tests) that put the focus on finding certain user interaction problems with the system, so that in the end, the team can make it easier to use and more intuitive. An indicator that usability is poor is when a user asks for a manual to learn how to use it.

Perhaps the most popular material in this respect is everything provided by Jakob Nielsen with his heuristics for analyzing different typical usability characteristics and problems. As a part of ongoing testing, teams should consider conducting such tests frequently, either by applying these heuristics manually and individually, in groups, surveys, or using tools to run some tests even in production, or in a beta version aimed at a small group of users.

Moreover, don't forget an associated quality factor known as **accessibility.**

In this case, accessibility refers to how easily someone with a physical disability can interact with a software product. It is highly beneficial to design it in a way that it is compatible with other tools that the physically impaired use such as Voice Over, Switch Access or Switch Control, etc.

Not only does accessibility impact the physically impaired, but everyone. Whether someone is trying to use an application in low lighting or with just one hand, they will benefit greatly if the app is designed with accessibility in mind.

Plain and simple, accessible design is good design.

There are many tools that facilitate this type accessibility analysis, from the W3C to some simpler and newer ones like Pa11y. Read more about how to set up tests with Pa11y in CI here.

To ensure user loyalty and satisfaction, it's important to consider how easy and enjoyable it is for users to interact with your software. Mature testing organizations that use CI will have tests in place to account for this aspect of their software.

TESTING MATURITY LEVELS

After reading about the different areas of quality and how to manage them in a continuous integration environment, you may be wondering how to assess how well your team is currently doing and what steps to take to reach the highest level of testing maturity.

We have defined a total of three testing maturity levels (to keep the model as simple as possible, more than anything). Once you know what level your organization is at, it will be all the easier to create a plan to continuously improve your testing.

The three levels are determined by three aspects: risk, quality, and costs.

Level 1: Basic Testing

- Risks are known
- Quality is measured
- Costs are measured

Level 2: Efficient Testing

- Risks are controlled
- Quality is controlled
- Costs are controlled

Level 3: Continuous Testing

- Risks are reduced
- Quality is optimized
- Costs are optimized

THE ULTIMATE GUIDE TO CONTINUOUS TESTING

TESTING MATURITY LEVELS

abstracta

The chart below shows a cross of the characteristics of each maturity level with the different areas that we raised:

	BASIC TESTING Aware of risks, quality and costs are measured	EFFICIENT TESTING Risks, quality, and costs are controlled	CONTINUOUS TESTING Risks are reduced, quality and costs are optimized
SOURCE CODE	● Source code versioning	● Code quality control	● **CONTINUOUS INTEGRATION / CONTINUOUS DELIVERY**
ENVIRONMENT/ INFRASTRUCTURE	● - Separate dev/testing environments - Few devices/browsers are covered	● - Test data management - Full device/browser coverage - Virtual servers	◍ - Test environment management - Containers - Service virtualization
INCIDENTS/ BUGS	● Bug tracking	◍ Defect causal analysis	◍ Traceability between features, issues, and code
TEST MANAGEMENT	● - Basic test management - Test planning with development	◍ - Testing before development - Impact analysis	○ - Agile management - Testing and devs in one unified team
FUNCTIONAL TESTS	● - Functionality inventory (backlog) - Traceability between test cases and features - Test cases, checklists, exploratory testing	◍ - Test design techniques - Tests are prioritized	○ - Long-term code coverage strategy
AUTOMATED TESTS	● - Unit testing - Automated API tests	● - UI automation - Unit testing with minimal coverage	● Unit, API, and UI automated tests running continuously
PERFORMANCE TESTS	● - Client-side performance tests - System performance testing before go-live - Reactive production monitoring	◍ - Unit performance tests (during dev) - Proactive monitoring in production	◍ Performance tests running continuously
SECURITY TESTS	● Basic tests of access control policies	◍ OWASP Top 10 pen testing	◍ Automated security checks
USABILITY TESTS	◍ Usability testing	○ User testing	○ Accessibility testing

● MANDATORY ◍ RECOMMENDED ○ OPTIONAL

The model distinguishes three levels for each group of tasks, some being mandatory, recommended or optional. The lines demonstrate the relationship of dependency between them.

Thus, you can see that to have continuous integration, it is necessary to have code quality control, which raises the need to manage versions. Moreover, you should also have a set of automated checks at the unit, API, and GUI levels. But first, In order to have those tests, it is necessary to have separate test environments and to manage them properly.

That is what we, at Abstracta, consider minimal in order to have a good continuous integration strategy, and to reach the highest level of testing maturity.

CONCLUSION

Once you complete the necessary steps to shift left testing and go from "quality assuring" to quality engineering, your organization will reap the highly sought after benefits of being able to efficiently deliver smaller, more frequent releases, keeping up with customer demands, competitors, and market conditions.

The best part is that as more clients and users become satisfied with your software because it consistently delivers on every aspect of quality that matters the most to them, so too will your business feel satisfied, with its increased profits and production capacity.

Additional Resources

- Browse our blog for even more in-depth information related to each section of this guide
- Watch the webinar recording: Learn How Shutterfly Employs Continuous Performance Tests for Winning
- Customer Experiences Build After Build
- Download our ebook: A Complete Introduction to Functional Test Automation
- Read the white paper: 10 Mistakes Companies Make When Outsourcing Software Testing
- Follow Abstracta on Twitter, Linkedin, and Facebook

Have questions or are looking for some assistance in your continuous testing efforts?

Contact us here or at hello@abstracta.us