United Certifications - Testing Fundamentals for Software Engineers (Developers and Administrators) Syllabus

Released
Version 1.1 30-10-2023

Copyright

Word of thanks

The authors would like to thank several people in particular in the creation of this work. It has long been a desire to be able to compile or record experiences in this narrative. This could not have accomplished without the help of the following people. A special thanks to all of them for making it possible, in one way or another, for this story to come about: Kyle Siemens, Rogier Ammerlaan, Daniel van der Zwan, Annelies van Rijn, Valentijn Duijser, John Wittmaekers, Anne Laura Drost-Douma, David de Roo, Sjoerd Walinga, Jose Correia, Khadidja Zegrir. And of course all other colleagues at Concept7 and the SSTQB and ITB for supporting the merit of this document.

Version

Version	Date	Comments
1.0	01-04-2023	First Publication
1.1	30-10-2023	Processed reviewcomments

Table of Contents

Introduction

Purpose of this document

This syllabus forms the basis of the Testing Fundamentals for Software Engineers (TFSE) certification. This document describes what you need to know to pass your exam. This document and the training program, including exam, are copyrighted. The exam will only include questions on concepts and knowledge explained in this document.

The different components of the training are also available on the official Certified Testing Fundamentals website for software engineers. These components consist of:

- A complete list of training providers and available course dates. A course is recommended but not required to take the exam.
- The syllabus (this document) to download.
- A complete practice exam of 40 questions and a document with answers to use in preparation for the real exam.
- We aim to have the documents available in multiple languages. Stay tuned to the website for further developments

Purpose of this syllabus

Testing is not an exact science. There are multiple, countless ways to achieve the goal: generally, an application free of serious and potentially costly errors. This course does not describe all facets of testing, nor does it teach you all the aspects of testing that exist.

There are plenty of other courses that zero in on aspects of testing in detail. More specific courses may be required if you want to become a test engineer or learn more about testing in depth.

The Certified Testing Fundamentals for Software Engineers course provides the essentials. It outlines a practical and pragmatic approach, providing methods that can be directly applied in most everyday projects.

The goal of this course is to teach some specific test skills to developers and administrators. This course is suitable for people with a range of experience, regardless of their level of senority.

Outcome after attending this training (Business Objectives)

BO 1	Learn practical issues about testing and quality as a developer or administrator.
BO 2	As a developer or administrator, understand the basics of testing and quality.
BO 3	Understand the prerequisites and principles for performing a good test.
BO 4	As a developer or administrator, learn a variety of tools for improving quality.
BO 5	As a developer or administrator, learn a number of different ways to create a test script.

Learning Objectives

Learning objectives are short descriptions of what you need to remember after reading the text in question. There are 3 levels **[I]**:

- K1: Remembering
- K2: Understand
- K3: Apply

The following table summarizes all the learning objectives (LOs) of this course.

LO1	Remember what is meant by testing. (K1)
LO2	Remember what is meant by quality. (K1)
LO3	Remember that you can look at quality in different ways using the quality attributes. (K1)
LO4	Understand the need for testing. (K2)
LO5	Understand the difference between the focus of developers and the focus of testers. (K2)
LO6	Apply some key testing practices. (K3)
LO7	Memorize some specific test terms. (K1)
LO8	Understand the test practice of make no assumptions. (K2)
LO9	Understand the practice that testing is logical thinking. (K2)
LO10	Understand the importance of regression testing. (K2)
LO11	Recall the pros and cons of test automation (K1)
LO12	Understand the practice that testing everything is impossible. (K2)
LO13	Understand the importance of having a test strategy (K2)
LO14	Apply risk analysis in your testing (K3)
LO15	Understand what Pareto's analysis involves (K2)
LO16	Understand how capturing the cause of incidents can help you improve your SDLC. (K2)
LO17	Apply the practice of being specific. (K3)
LO18	Understand the practice of testing as early as possible. (K2)
LO19	Types of environment including testing (K2)
LO20	Understand the importance of a good testing environment (K2)
LO21	Understand the importance of test data (K2)
LO22	Understand the practice of starting small and gradually expanding your testing scope. (K2)
LO23	Recall test levels and test types (K1)
LO24	Understand how test levels and test types apply to a test strategy (K2)
LO25	Recall a basic testing process (K1)
LO26	Understand the value of documenting your testing. (K2)
LO27	Learn to create a test script by drawing out the process/program. (K3)
LO28	Learn to create a test script using the semantic test. (K3)
LO29	Learn to test functionality using a decision table. (K3)

LO30	Learn to deepen your testing by using boundary value analysis. (K3)
LO31	Learn to deepen your testing by using equivalence classes. (K3)
LO32	Learn how to test using a checklist. (K3)
LO33	Learn the test technique of pairwise testing. (K3)
LO34	Understand the importance of good communication in all your activities as a tester. (K3)
LO35	Learn the basics of testing for security. (K2)
LO36	Learn the basics of usability testing. (K2)
LO37	Learn the basics of performance testing. (K1)

Prerequisites

No specific prior knowledge is required; however, it is helpful if you have some experience with one or more of the following areas: software development, project management, managing software, accepting software, and/or testing software.

General Notes

To maintain the flow in the text of this document and courseware, the authors may refer to:

- "**Software**," in some places where "Product, Service and/or System" is intended
- "**Software engineers**," in some places where developer or administrator is intended
- "**SDLC**," is the abbreviation for Software Development Life Cycle.

Standards for testing techniques

In this syllabus, various testing techniques are explained. A conscious decision has been made to keep this limited and straightforward. The basics of some techniques are taught because this syllabus only describes the fundamentals of testing. For additional testing techniques and more advanced usage, I refer to the ISO standard [XX], and/or other sources.

Applicability in Agile and DevOps

This syllabus makes less reference to various system development methodologies (Waterfall, Agile, DevOps). This is a deliberate choice. We want to establish the essence of testing here, and, in our opinion, this essence does not differ with another system development methodology. There will be differences in how you execute the activities described in the syllabus in a project, in time, or in the environment where you perform them. You will also notice that in other types of organizations or with different types of software, the emphasis may vary. Sometimes because organizations have already evolved to a higher maturity level. But the basics of software testing and the importance of these basic skills will not change based on the chosen system development methodology. Since this syllabus is aimed at developers, administrators, and others starting with testing, it has been chosen to represent the essence and not go into too much detail on matters that are further from the basics.

Note from the main author Mattijs Kemmink:

Over 20 years prior to initiating the creation of this syllabus, I began my career as a software tester. I actually wanted to be a software developer, but my employer followed a policy that software developers must start as testers. At the time, I had to ask what testing was because I had not heard of the concept as a role. However, I was happy to get a permanent job with a large employer. And so began the first steps on my software testing path. In the years that followed, I had opportunities to switch from tester back to developer several times, but I kept returning to my testing roots.

Throughout this syllabus, I hope to take you on a journey through some of my experiences in the world of testing, relating them to the theory of well-known testing methodologies. I will present the concepts as straightforward as possible, trying to do so as I experienced them throughout my career without unnecessary baggage. However, realize that testing is never simple as it is always customized and is dependent on the relevant situation and environment. Please note that this course is not a replacement for standard theoretical courses (e.g., the ISTQB), as the QA field is simply too large for that. My goal in this course is to provide participants with a relevant experience to better understand the role of the tester and the importance of the field. I aim to help people improve the overall quality of products that they collaborate to produce and provide the knowledge that I wish I had had at the start of my career. I hope to teach you practical aspects that are immediately applicable in your daily practices. Throughout my career, I have come across many developers and administrators who were involved in and very concerned about quality and wanted to learn more about how to get involved, but did not know where to start. These are the people that I'm hoping to help with this certification course.

Chapter 1: Getting acquainted with Testing

LO1	Remember what is meant by testing. (K1)
LO2	Remember what is meant by quality. (K1)
LO3	Remember that you can look at quality in different ways using the quality attributes. (K1)
LO4	Understand the need for testing. (K2)

What exactly is testing?

"Testing" has many definitions. In this course, "testing" will be defined as follows: "a process that provides insight into and advises on quality and related risks." **[II]**

There are two valuable components: "quality" and "risk." Quality is defined and detailed in the section that follows, but it should first be noted that one might assume testing is the only means of providing insight into quality. However, there are other ways to achieve quality. Testing is in itself a reactive measure because it takes place when the product already has been created. But, something that can be more useful and efficient than testing, in many cases, is reviewing. This is because you can often carry out a review even before the product is developed. By reviewing, you can increase quality early on by improving the requirements that have been created, creating requirements that are missing, or removing redundant requirements.

The second part of the definition of "testing" that needs highlighting is "risks." Risks already exist because you provide insight into them with testing. If a risk analysis has been done for a specific project, then you can check to what extent the risks are still current. Testing can also reveal new risks. Through testing, organizations aim to find and eliminate or mitigate risks with respect to their operations. Testing might also be carried out to meet regulatory requirements, or to ensure that business processes, products, and IT solutions deliver quality.

What is Quality?

"Quality" is defined in this course as follows: "Quality is the set of attributes and characteristics of a product or service that is important for meeting established or obvious needs" **[III]**. In simpler terms, we might say that quality consists of attributes and characteristics, and quality corresponds to expectations that may or may not be established.

This definition of "quality" references one of the trickier aspects of testing: "unestablished needs." When dealing with a list of requirements there will be some unestablished needs that might be obvious to certain stakeholders and not so obvious to others. For example, consider closing an application with the X button (exit) in a Windows environment vs. an Apple environment: the same action would be on a different part of the screen.

Quality Attributes

If we considering that “Quality” deals with “attributes and characteristics” (taken from the previous quote [III], then we should consider that “Quality Attributes” are different ways of looking at quality. The ISO 25010 **[IV]** standard describes a number of quality attributes that can help us in software testing. Some examples of these are: functionality, security, performance, usability, maintainability, and portability. The ISO 25010 classification and subdivision of quality into quality atributes is very useful in software testing. We will look at some of the quality attributes in more detail in a later chapter.

Dependancy on Software

Software is everywhere these days; it is nearly impossible to live without it. Just think about having to live a day without your smartphone—this would take quite a bit of getting used to for many people. You could probably survive without your phone, but life with a smartphone is a lot faster and easier.

As your smartphone makes your daily life more efficient by relying on software, if we take a look at the bigger picture, our society has processes that are also highly dependent and improved by the use of software. Just think about all the automated processes of the government, and take the example of tax collection. If we returned to filing taxes without computers, the process would take much longer, cost more, and require substantial human resources. Another example is train systems. Software controls train schedules, access controls, and season tickets; it would certainly be a shock if we returned to carrying out these processes without software. As we want society to continue to function efficiently, we must ensure that software is also reliable and continues to function.

In addition, and precisely because we rely on it, we want software to function as intended. For example, take the software that makes the airbags in a car deploy in an accident. We want this to happen exactly as intended. If the airbags are deployed too late, there will be damage; if they are deployed too early or unexpectedly, there will be damage as well. In this example, the most crucial damage that airbags are implemented to prevent is physical injury; however, there could also be material damage to the vehicle, immaterial damage such as psychological consequences, or significant financial damage. This example illustrates how software testing is imperative in preventing damage, demonstrating quality, exposing and mitigating risk, and ensuring continuity.

Definitions

Testing	A process that provides insight into and advises on quality and related risks.
Quality	The set of attributes and characteristics of a product or service that are important for meeting established or obvious needs.
Quality Attribute	A characteristic of an information system.

Risk	A factor composed of probability and impact that can have consequences.
Functionality	A thing's usefulness, or how well it does the job it is meant to do.
Maintainability	The degree to which a product or system can be changed effectively and efficiently by designated administrators.
Portability	The degree to which a system, product or component can be effectively and efficiently transferred from one hardware, software or other operational or usage environment to another.
Review	The activity of evaluating a product or process with the goal of finding errors or making improvements.

Chapter 2: Having the Right Focus for Testing

LO5	Understand the difference between the focus of developers and the focus of testers. (K2)

As a developer, it can be difficult to test well. A developer's main goal is often to create a working product based on given specifications. The developer's focus is on delivering a customer-defined and appropriate product. As developers are mainly focused on the end results or solutions, they are less likely to notice peripheral issues.

A tester, on the other hand, has a completely different goal: to look at the application in every possible way and see if it functions correctly. A tester's focus could be considered the opposite of a developer's focus, which is why there is a natural job differentiation between testers and developers. This is why it can be tricky to combine these two very different perspectives at the same time.

However, if it is your responsibility to do both, there are some tips you can use to make it easier. For example, you could agree with a colleague that you test each other's work in order to enable a more independent view and detach yourself from the development focus. You could also designate a separate test day for yourself, or set aside a separate moment in the day when you consciously focus on testing instead of development. Having the right focus is crucial; some concrete tips will be discussed in the next chapter.

As an administrator, you can also use an awareness of these challenges to better understand why testing is sometimes excellent and sometimes lacking. One of the things you can do is question developers about their testing process, e.g., whether they employ testers, whether their developers also test, etc. If you are still in the contract negotiation phase, you can even propose explicit requirements for this. Of course, if you are an administrator who does software development or configuration yourself, you can apply the above tips for development in your own work.

Chapter 3: The Eight Test Practices

LO6	Apply some key testing practices. (K3)
LO7	Memorize some specific test terms. (K1)

To make the subject matter of testing as accessible as possible, we have formulated a number of practices based on experiences in the world of testing. These practices are linked to a number of concrete applicable activities, which should provide insight to help you maximize software quality. Adherence to these practices can reduce the chance of errors significantly. And even as a non-tester, you will still be able to improve quality of software, processes, and requirements. The principles we would like to explain are:

1. Make no Assumptions.
2. Testing is Logical Thinking.
3. Testing Everything is Impossible.
4. Be Specific.
5. Test as Early as Possible.
6. Start Small and Gradually Expand your Testing Scope.
7. Documenting your Testing.
8. Understand the Importance of Good Communication

Each of these will be explained in the sections below. In addition to these practices, there is another important universal principle that applies: testing always depends on context. In this case, “context” means environmental factors and the project conditions. Environmental factors include the specific industry or sector in which the test is taking place; the nature of the software you are creating, as well as the organization or project; and the legal requirements, regulations, guidelines, and industry standards that apply. Project conditions include time, money, and quality.

Also note that you can both apply these practices yourself and know that others building or testing software for your organization should apply these practices. You can always apply this to a vendor by asking for it or giving it as a condition.

Applicability in Agile Testing and DevOps

In modern, mature IT organizations, you may sometimes see that testing and quality have evolved, and it seems like some of the things in this syllabus are not applied or applied to a lesser extent. In some cases, this may indeed be the case. And it is a fact that there are other tools and resources to apply in these environments. However, the goal of this syllabus is clearly to establish the basics, the essence of testing and software quality. We also aim to make testing accessible to a group that is inexperienced in the field. Even in organizations with a higher maturity level or with an Agile or DevOps process, you could apply this foundation based on the practices. The difference will be that you'll need to review your test strategy per sprint or per user story and possibly adjust it specifically for that sprint or user story. If you are working on

stories with high risk in your sprint, you might choose a more robust testing technique that achieves broader coverage, for example.

In the case of DevOps, as an example, you could perform the various test levels in perhaps the same environment. But, at the very least, you perform them to ensure that forgetting them does not lead to reduced quality. For the application of other tools and resources in an Agile or DevOps environment, we gladly refer you to more comprehensive training tailored to these methodologies. For example, TMap, ISTQB, Agile United, or DevOps United, which can complement this syllabus.

Practice 1: Make No Assumptions

LO8	Understand the test practice of make no assumptions. (K2)

The basic rule for all testers is “do not make assumptions”—this is the answer you will get from anyone in testing. As a tester, you need to have an extremely critical attitude toward what you are testing, as well as possess a healthy dose of distrust. As a tester, you should check basically everything, i.e., whether the system works as described. You should also check whether what was described is what the customer needs. The former is called “verification” in the testing world: the process of checking that the software does what was specified. The latter is called “validation”: checking that the specification does what the customer needs and/or expects. **[V]**

As a tester, you will sometimes have to assume that what is described is what the customer wants. At other times, you will have to assume that the supplier’s specifications are correct. In terms of best practices:

- assume that everything described or said about the software should also be verified;
- it should also be validated that what the customer want has been built.

A typical example of an assumption is thinking that if something works in one environment, it will also work in another environment. In fact, this is very often not true. Notice that when it comes to configuration management, for example, there are different versions of web components for different environments, and some components may not be considered to exist in particular environments. For these reasons, different components might not even be noticed in certain environments.

Take as an example a relatively simple system (see below Figure: Overview of Environments) in which we have an information system that consists of a database, code, functions, and an external software component, that are all published on a web page. In the figure, the green check marks in the images depict what is going well and what has been tested. The red circles indicate defects.

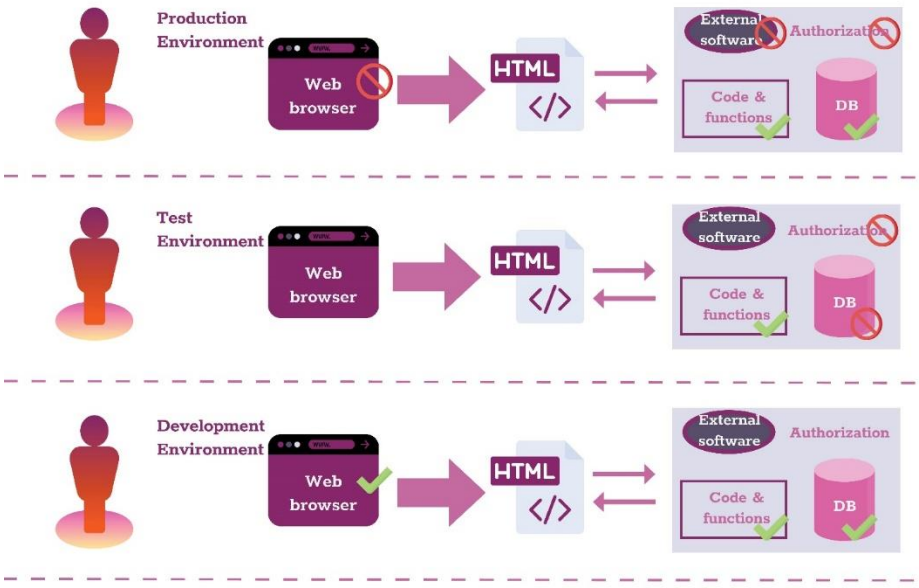


Figure: Overview of Environments

If you are building this system, you can test it thoroughly in your development environment—how it should be.

However, after you have carried out the test in the development environment, you need to deploy it to the test environment. As you have tested that the functionality works, you are likely confident that it will work in the test environment as well. But a number of questions arise: what if you forget to deliver a component? What if you need to use a different address for third-party software (think of certain plugins)? What if you need a different certificate in a different environment? Authorizations will also have to be reassigned. Are these right the first time? If so, how do you know that the different roles you had defined also work? Sometimes when tests done with the admin role work fine, they suddenly do not work for a regular user. On top of all this, the client may have a severely outdated or untested web browser that may create errors that you did not encounter with your test in the development environment.

This does not mean that you have to repeat the entire test in the test environment and then again on production: you have already demonstrated that the functionality works. However, while you might have tested everything extensively in the development environment, you might now additionally test a few instances that demonstrate that delivery is complete and that all components are there. The important point of this is that if you dynamically verify that it works, you have the greatest chance of getting it right. If dynamic verification cannot be done, you can also check or validate statically. It is also important to communicate. When in doubt about whether something is intended, contact the tester or the compiler of the requirement. Static and dynamic verification/testing will be elaborated in practice 5 of this syllabus.

Later in this syllabus, we will also discuss communication in more detail. For now, it is important to remember not to make assumptions. Additionally, if you are a developer or administrator, you must do away with assumptions such as “testing is difficult,” “testing is for testers,” “there is no time for testing,” “I have done all the unit tests,” “there is good logging,” “the testers should also have something to do,” “testing is boring,” and “it works on my environment.” Without training, it is possible to feign ignorance, but after attending this training, you will know better.

In summary: if possible, try to verify everything. If possible, do it dynamically by carrying it out. Be careful not to duplicate tests. For example, you will no longer need to test the code of your component very extensively in the test or production environment. If dynamic verification is not possible, try validation. In any case, the most important thing is that you communicate what you choose to do.

Definitions

Assumption	An assumption, premise, or hypothesis which has not been proven.
Configuration management	Configuration management focuses on the versioning of items throughout the SDLC. It enables the tracking of which specific instance of a design belongs to which specific instance of code.
Dynamic testing	Testing by executing code (from a component or system) **[V]**.
Static testing	Testing a part (of a product or workitem) of the SDLC without executing code **[V]**. For example review of code, requirements or static analysis.
Test Environment	An environment containing hardware, instrumentation, simulators, software programs, and other supporting elements necessary to perform a test.
Validation	The process of checking that the specification does what the customer needs/expects.
Verification	The process of checking that the software does what is specified.

Practice 2: Testing is Logical Thinking

LO9	Understand the practice that testing is logical thinking. (K2)
LO10	Understand the importance of regression testing. (K2)
LO11	Recall the pros and cons of test automation (K1)

Software testing is not very complicated, and it often comes down to some basic logic. Take the statement below:

'IF it rains, THEN you get wet.'

If you wanted to develop this requirement, you could turn this into code fairly quickly. Then you probably would run a unit test for this, as you usually do when you develop software. The latter is an assumption, and, of course, it never hurts to ask if and how someone tested. Keep in mind, however, that in development, the focus is on realizing things and turning requirements into code. The primary focus is on the solution. If you want to put yourself in the shoes of a tester, however, you will have to approach the requirement very differently. For example, what if it doesn't rain? Will you get wet? And what if it does rain, but you don't get wet? What should happen in that situation?

As a tester, you will want to check the following things:

Scenario	It rains	You get wet
1.	True	True
2.	True	False
3.	Untrue	True
4.	Untrue	False

Table: Possible Outcomes of an Assertion

You would need to validate that the 2nd scenario cannot occur and that the other scenarios can occur to confirm that the statement is true.

The example above is not about IT. Now take, as another example, the following requirement:

*'IF I am an **administrator** of this system, THEN I want to be able to change **users**'*

Scenario	I am an administrator	I am able to change users

1.	True	True
2.	True	False
3.	False	True
4.	False	False

Table: Possible Outcomes of an Assertion

Seeing this example, we immediately understand that being able to add, modify and delete users as a non-administrator is not desirable.
This would definitely need to be tested. In addition to your focus on realizing the requirement, try to focus on the “What-If” or “False” situations as well.

If we take the example of uploading a .pdf file on a website, the importance of logical testing becomes even clearer. Take the following requirement:

*'IF I am an **administrator**, THEN I want to be able to add a .PDF file to a website.'*

If you look at what you need to accomplish as a developer, testing is limited. However, as a tester, this case should involve much more testing. Uploading files with a different extension can have undesirable consequences. You also want to check that users without administrator access cannot upload files.

The following table illustrates what is meant by this:

	1	***2***	***3***	***4***
I am administrator	Y	Y	N	N
File is a .pdf	Y	N	Y	N
File can be uploaded	X			
No action		X	X	X

Table: decision table for uploading a file

It should now be clear that at least 4 test cases should be carried out. But a quick inventory shows that there are a number of roles besides “administrator” and “user,” so path “**3**” should be tested with all those roles. Also, there are several other file extensions that you would want to exclude for legal or security reasons, e.g., .doc, docx, .ppt, .exe, .zip, .rar., etc. This leads to more situations in path “**2**” than one would think at first sight.

In addition, just checking for an extension may be insufficient. A .pdf file can also contain malicious content. So you should suggest that the requirement be expanded to include an

additional requirement for the integrity of the file. For simplicity sake, let us park this as a side note for now.

One might argue that there is no reason to do an extensive check in this case because this is an administrator. After all, the permissions of an administrator are reserved only for a few people who generally know what to do. However, the functionality here could be reused elsewhere, or the password for an administrator could be discovered, so an extensive check may be beneficial.

Regression

We can already see that for relatively simple functionality, quite a lot of reasons can arise to test something. It also becomes quite clear that the testing process can become quite large. Now, imagine that the requirement in this example was modified after the testing process for the original requirement was completed. Let's say that another file type is now allowed. The new functionality would look like this:

	1	***2***	***3***	***4***	***5***	***6***	***7***	***8***
I am administrator	Y	Y	Y	Y	N	N	N	N
File is a .pdf	Y	Y	N	N	Y	Y	N	N
File is a .doc	Y	N	Y	N	Y	N	Y	N
File can be uploaded	X	X	X					
No action				X	X	X	X	X

Table: Decision Table for Uploading a File with a New Requirement

You thought you already had many test cases in the original situation, but now you see that the number is increasing. One reason that should be noted here is that you cannot assume that the changes in the code did not have unintended side effects on the existing functionality and therefore the original test cases must be tested as well. Remember to consider the functionality you want to test for regression with each new change. The tests for verifying that existing functionality remains unaltered are called regression tests. Regression testing is generally suitable for test automation due to its repetitive nature.

Test Automation

Before tests can be automated, manual testing is always required to gain an understanding of the system and to know what is required to create automated test scripts. **Test Automation** usually refers to the automated execution of tests: a relatively narrow definition. The definition could perhaps be expanded to state "automating parts of the test process" because it is often desirable to automate time-consuming tasks throughout the test process in an easy and cost-effective manner. For example deployement of software from one environment to another environment.Although there are many benefits to automating testing tasks, there are arguments to consider that can be made against test automation. For example, to automate

something in the SDLC a mature process as well as an investment of time, money, and resources are required. This may mean that it may take some time to receive a return on your initial investment.

Test automation can be valuable, for example, when large numbers of (regression) tests in mature complex software environments are being performed over long periods of time. Other examples of situations that lend themselves to test automation are large projects in which the same areas need to be tested over and over again (projects with many iterations). Test automation can also be useful in projects that have already undergone an initial manual testing process. Implementing test automation for a mature software product can improve the cost efficiency of testing and increase the overall test coverage.

In general, test automation lends itself well if basic testing processes are mature, or in other words, if there is a well-established testing strategy in place, including different test types of tests that cover the different test objectives.

Some typical pitfalls of test automation to be aware of:

- Automation is a means; it is still sometimes seen as an end.
- It involves building yet another application that needs to be designed, learned, managed, and supported.
- With test automation, you often build an application that distracts from the application you wanted to test.
- Automated test scripts require maintenance, and overlooking maintenance may render the scripts unmanageable.
- Keep in mind that before you can automate a test case, it must first be designed manually.
- Thinking about and developing a test strategy also gives you a better understanding of which parts of the system are well-suited for test automation. In addition to regression testing, these are often tests where the application needs to be tested on different operating systems, web browsers, and devices.

There are many benefits to the automation of certain tests: however, the key is finding the balance in what is most appropriate for automation. Sometimes **Automation in Testing** is even more logical and even appropriate than test automation itself. Test automation is a vast topic and if you are interested there are further full courses on this topic including certain tools like Selenium.

Test Tooling

There are a lot of tools available that can be of great support in different parts of the testing process or various parts of the SDLC. When capturing test cases and defects, and also for configuration management, suitable tools can be very beneficial. For performance testing, it can even be debated that tooling is a necessity. Please consider that although tools offer

incredible solutions, they often service a wide spectrum of tasks of the SDLC, so they may not offer the exact functionality to suit your needs.

If you are going to use tools, do a pilot and a good risk analysis of the tool, and carefully consider the (license) costs and the maintance costs of tooling. Tools are not always what they seem and the costs are not always clearly specified. There are also risks associated with open-source tools, which may not have a dedicated support team. Try to avoid creating extra extensive projects simply to satify the implementation of tools.

Coming back to the idea of “testing is logical thinking” follows the concepts of test automation and test tooling, where appropriate actions must be taken prior to implementation to make them successful. As the ultimate goal of every project is quality software, it is important to maintain focus on this goal and begin to focus more in the implementation of extra (potentially unnecessary) tools and automation.

Definitions

Action	The event that is taking place.
Automation in testing	Automating steps of the test or SDLC process.
Condition	A state that may or may not be met.
Decision	The outcome of a condition.
Test automation	The use of software to execute or support testing activities.
Test tooling	Software or hardware that supports one or more testing activities.
Regression testing	Re-testing pre-existing functionality to determine if it functions as it did before a given change.
Requirement	Description of what is necessary.

Practice 3: Testing Everything is Impossible

LO12	Understand the practice that testing everything is impossible. (K2)
LO13	Understand the importance of having a test strategy (K2)
LO14	Apply risk analysis in your testing (K3)
LO15	Understand what Pareto's analysis involves (K2)
LO16	Understand how capturing the cause of incidents can help you improve your SDLC. (K2)

In the software and applications that are built, there are countless choices and possibilities. In addition, you can view applications on various devices. For example, even a relatively simple **application programming interface (API)** quickly has several dozen options for entering data. It is impossible to run all potential test cases, so you will have to make choices and differentiate between them. These decisions may seem difficult at first, but some tricks will help.

Let us consider that we are changing the functionality of a phone number field. On a given website, the collection of a phone number via a phone number field is an integral part of the customer journey, used to verify customer identity when placing an order in the webshop. Guidelines for the correct collection of phone numbers in a phone number field must be considered, for the functionality of customer identity validation to be possible. This does not always happen correctly—not by users who enter the phone number by hand, nor by application developers who program the fields. Consider the following telephone number formats as examples:

Country	**Notation examples**	
Netherlands	+31 059 660 0233	00(31)059 660 0233
	+(31) 059 660 0233	00(31)59 660 0233
	+31 59 660 0233	00 31 059 660 0233
	+31 (0)59 660 0233	00 31 59 660 0233
United States	+(1)(425) 555-0100	+14255550100
United Kingdom	+(44)(20) 1234 5678	+442912345678
UK mobile prefix	+(44) 07412 123 456	
China	+(86)(10) 1234 5678	+861012345678
Singapore	+(65) 1234 5678	+6512345678

Table: Telephone Number Formats

In this example, an API could be addressed in several ways, allowing many different SOAP (Simple Object Access Protocol) messages and many different REST (REpresentational State Transfer) messages to distinguish between landline phone numbers and mobile phone numbers.

Depending on the SOAP or REST message (which have very different functionalities), there were previously two or four phone number fields in these types of messages. The field for mobile phone numbers always had to start with a mobile prefix. There was also a strict requirement in

the mobile phone number field: a maximum of 10 positions, always starting with a "+"-sign followed by a two-digit country code, which in this case also had to have the fixed value of the country. All those requirements have now been abandoned, and the field is now allowed a maximum of 20 positions. The country code and mobile prefix requirements were also abandoned, which means that you are now allowed to enter a landline number in the former field and vice versa. If you look at the way you could write down a telephone number you arrive at an infinite number of situations.

Note here that what is shown above table (*Examples test cases phone numbers*) is only part of the set of possible landline number notations, and mobile is still largely out of the equation. A developer can solve this fairly easily by building a check in the field that excludes all special characters and adds a "+"-sign automatically. However, as a tester, you have to put quite a lot of effort into checking all the valid input. You would have to test different inputs in all the different fields as well since you cannot assume that all fields indeed work with the same mechanism. As a developer, you can validate this in the code, but from a testing perspective, the code is a black box. In the end, this can result in tens of thousands of test situations.

It is impossible to test all those (functional) situations, as this will generally take too long or simply cost too much. You will have to make choices about what is required to test. It should be clear what the test object (phone number field) technically consists of, as this will help to gain a clear picture of what needs to be tested, which should be documented in a **test strategy**.

What is a Test Strategy?

A test strategy could be considered the overall tactical guidelines, which should explain all the ins and outs surrounding everything that needs to be tested throughout the SDLC. An important aspect to remember when creating a test strategy is to make sure that it provides helpful insights into the **test object**.

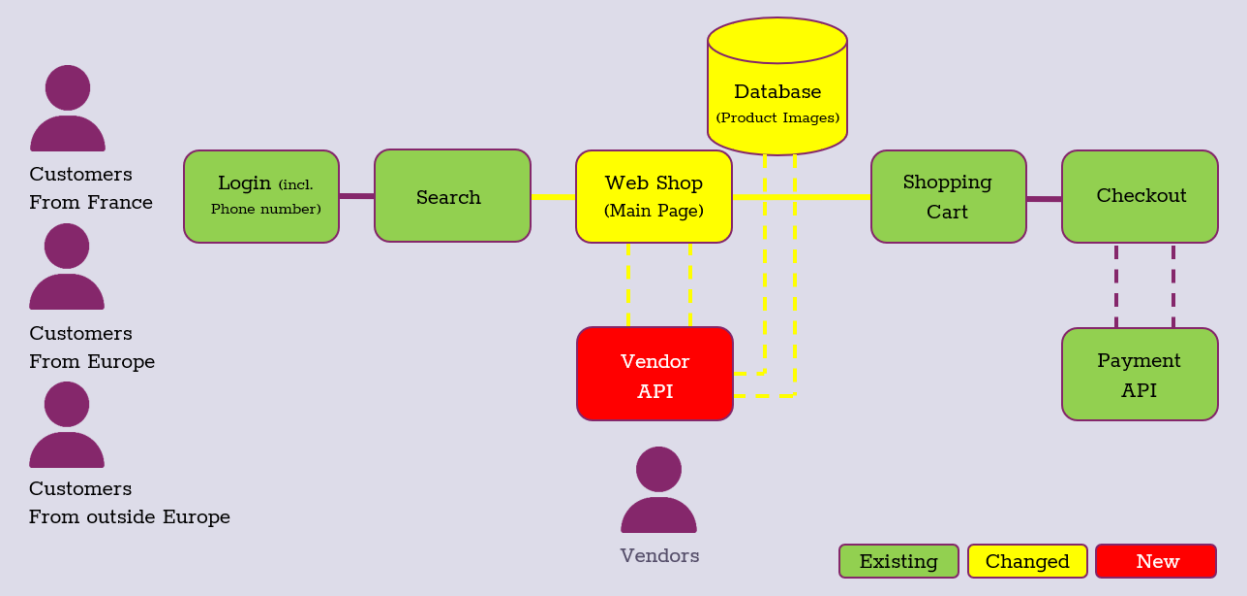
Creating a Test Strategy: Gathering Information

Useful information when creating a test strategy (and deciding on test objectives):

- Requirements: to serve as a test basis for our tests, for example, (high-level) designs, epics, user stories, use cases, technical documentation, user manuals… etc.
- Risks: to have a stronger understanding of where potential weaknesses of the system could exist. These can be sourced from existing risk assessments.
- Quality attributes: to better pinpoint the areas that may require extra focus, for example, if financial and personal information of customers is being processed (Security), many users using the system at one time purchasing tickets (Performance), etc.
- Existing high-level test cases: to serve as examples, these can be taken from an existing regression set.
- Incident reports (service desk tickets): to depict common problems in the process or system that should be considered.

Creating a Test Strategy: Sketch the System

It can be useful to include (or draw) a sketch that describes how the test object works (including the parts that exist and their surroundings). Including a sketch of the system can help to clarify which parts of the system are new, and which parts already exist. It can also be beneficial to color-code the sketch as seen here in this example:



This sketch can be used for assigning test levels and test types, which we will discuss in "Practice 6: "Start small and gradually expand your testing scope".

Once we have a sketch that explains how the system works and have gained more insight into the application itself, it could be beneficial to deepen our analysis of the risks to sharpen the focus of our test strategy accordingly.

Developing a Basic Test Strategy through Risk Analysis

When deciding what to test, we have to develop a basic risk-based test strategy. There are several ways to create this type of strategy, which can vary in length and extensiveness. We will elaborate on some of the more basic approaches that help you construct your risk-based test strategy.

To be able to create a risk-based test strategy, it is important for us to first define what we see as a risk. According to TMap **[XII]**, a risk is composed of the two elements "probability of failure" and "damage". The probability of failure consists of the frequency at which the process is being used and the complexity of the process itself. Consider the following formula:

Risk = Probability of failure (Frequency + Complexity) * Damage

Let us look at how we can best identify risks through risk assessment.

Risk Assessment to Construct a Test Strategy

Risk assessment **[XIX]** is a basic process that consists of three required steps to construct a basic risk-based test strategy: identification, analysis, and mitigation. A good source for **identifying risks** is the client (the application owner). In being the main stakeholder, the client usually understands the business for which we are building (or changing) a system the best. Therefore the client should be aware of the risks and particularities of the given business and can help decide what the most important risks are. Throughout the risk assessment process, it is also important to include other stakeholders from the client side, for example, system administrators, developers, and key-users. These other perspectives can drastically improve the validity of the **risk analysis** as a whole. They tend to have relevant insights for maintaining, using, and developing the functionality allowing them to elaborate on the frequency of use, complexity, or common issues with the functionality. Once risks are analyzed, we can start to focus on the **mitigation of risks**. Testing in itself is a risk mitigation measure. There are further measures to mitigate the risks, for example, the implementation of monitoring after deployment to the production environment.

Using the previous example of changing the functionality of a phone number field, our client might be able to confirm that 90% of their incoming orders are from Europe. Through this valuable information, we know that good working functionality is very important for European telephone numbers. Therefore, we can use this information to divide our functionality and thus our risks. Let us consider the risks that we have now identified:

Nr.	Risk
1.	Telephone number functionality within Europe is not working.
2.	Telephone number functionality outside Europe is not working.

The testing of all phone number formats from each country in Europe is already a lot of ground to cover, so it would be beneficial to narrow the scope even further. By asking more specific questions to the client about the most important markets within Europe, we can discover which telephone number formats require the most testing. For example, if our client says 70% of the revenue comes from France, we can create further subdivisions of the risk:

Nr.	Risk
1.	The telephone number functionality from France is not working.
2.	The telephone number functionality from other countries in Europe is not working.
3.	The telephone number functionality outside Europe is not working.

To keep our example simple we will stop at these identified (in reality there could be many more).

Now we must take these identified risks and properly analyze them. Risk analysis involves learning about the probability of failure and damage.

The probability of failure exists out of the factors of frequency and complexity. With frequency, we refer to the interval in which the program (or process) is exercised. When determining the frequency of use, it is a good rule of thumb to obtain the usage figures of the specific functionality from the application in the production environment. If this data is available it will provide factual insight into the usage of different functionality. Often those figures can be obtained by looking at logging or business intelligence reports for the mentioned functionality in production. By adding this data, you gain a more factual understanding of the frequency with that a particular functionality is used.

With complexity, we are referring to the difficulty to exercise a certain program (or process), or the complexity of the program itself. If we combine the results of frequency and complexity, we will arrive at a final score for the probability of failure. It is important to remember that risk analysis is always a subjective way of estimating something, which is why we should involve the client as much as possible. This subjectivity is required because we are trying to put a value on the priority that should be taken for the given risks.

In risk analysis, it is common practice to use a three-point scale with the values “high” (H), “medium” (M), and “low” (L) for all components. It is important to note that everything cannot be listed as “high”, as everything is not equally important. Consider the following **risk matrix** for our telephone field functionality example:

Risk table			**Function**	Login 2FA email	Login 2FA sms	View insurance details	Apply for insurance	Change insurance
			Frequency	L	H	H	L	M
			Complexity	H	H	L	M	H
			Total	**M**	**H**	**M**	**L**	**H**
Process	**System**	**Damage**						
Car insurance	Webportal, backend	**M**		B	A	B	C	A
Healthcare insurance	Webportal, backend	**H**		A	A	A	B	A
Boat insurance	Webportal, backend	**L**		C	B	C	C	B

Risk Matrix: Telephone Field Functionality

In this risk matrix, the bottom portion “Process”, “System”, and “Damage” refer to our subdivided risks. “France” is listed under Damage as “H”, because 70% of the orders come from here. Through mathematical deduction, if 10% of the orders were from “Outside Europe”, we could estimate that 20% of orders come from the rest of Europe (“Other Europe”). Hence, “Other Europe” is listed under “Damage” as “M” with 20%, and “Outside Europe” is listed under “Damage” as “L” with 10%. While discussing the process with the stakeholders we understood that the “Frequency” is “H” because the field is used for every order to validate the customer identity. As the changing of a telephone number field is a relatively common and straight-forward procedure, one could (remembering that it is subjective) consider that the

"Complexity" is "M". The "Total" estimate for the probability of failure for this risk is "M". When looking at the bottom righthand corner of the risk matrix, the "A", "B" and "C" refers to the level of importance of the functionality when mitigating risk. "A" refers to the risks that should receive the most coverage to mitigate the risk that would damage the company the most, "C" refers to the risks that require the least coverage (as the overall risk to the company is lower), and "B" is used for the risks that are in between.

Test Objectives

Test objectives, often called "test goals", could be a given functionality, part of a software system, or even a quality attribute that should receive special attention during testing; therefore, there is no exact science or rule for deducting them.

As we have seen in our telephone field functionality example, during risk analysis we can see that the division of the risk into different parts gave us some valuable viewpoints on the functionality. These are "Orders from France", "Orders from other parts of Europe", and "Orders from outside Europe". Here we have listed only these, but considering a webshop where you can order items, it is clear that we could name some other parts of the webshop as potential (even valuable) test objectives as well. Some examples could be: "the user-friendliness of the interface", "the shopping cart", "the checkout", "the payment options", "the search engine", "the performance", "the API for external vendors", "the API documentation". These are all examples that could become important if you were to dive into this example more closely.

The Risk-based Test Strategy

Once we know where the various levels of coverage are required, we can split up the system into different areas of focus (test objectives) and use them to create a test strategy. Creating a test strategy is about creating a structured approach to your testing through the use of different test techniques that can apply different levels of coverage to the various test objectives.

In our telephone field functionality example, we have limited our test objectives to keep the example as simple as possible, but if creating a real test strategy for this example, we would have to consider that there could be much more to take into account.

While creating your test strategy and for example looking at incident reports, if it becomes apparent that the list of test objectives is very extensive, or it becomes very challenging or seemingly impossible to include them in the test strategy, "Pareto's analysis" is an example of a method that could be helpful.

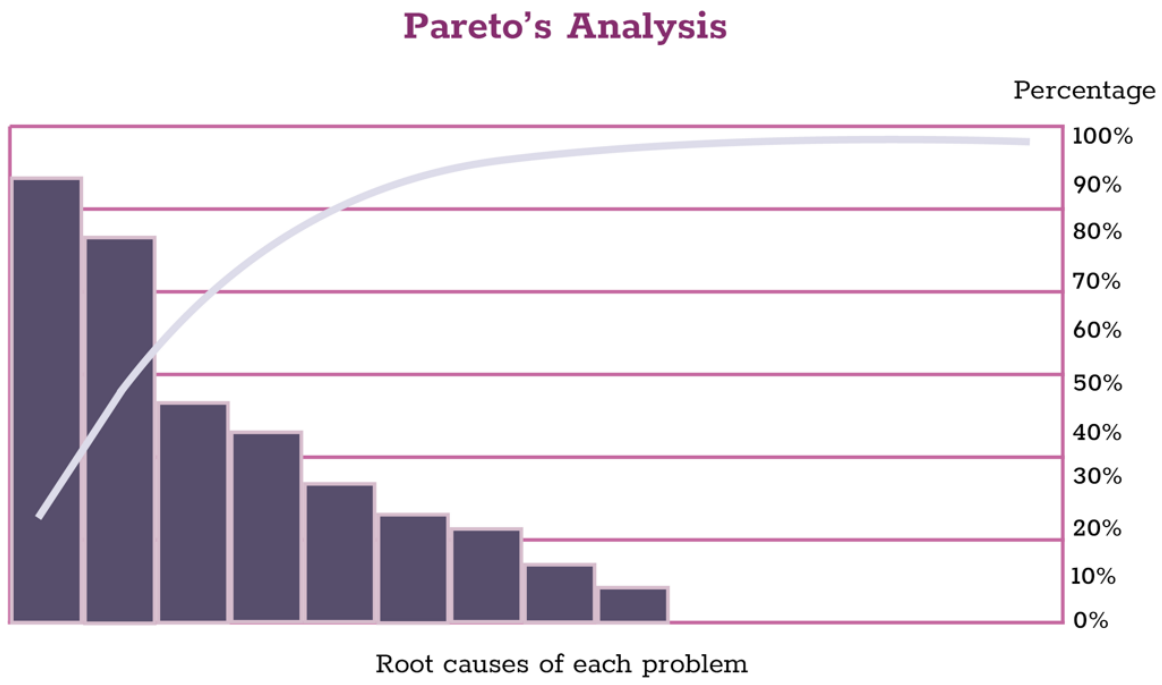
Pareto's Analysis (Example Risk Assessment Method)

Pareto's analysis can make the assessment of risk more efficient **[VI]**. Pareto's analysis is based on the idea that 80% of the benefits of a project can be achieved by doing 20% of the work. Conversely, 80% of the defects can be related to 20% of the causes.

Steps when implementing Pareto's analysis:

1. Identify and describe problems that have occurred
2. Identify the root cause of each problem
3. Assign a relative score to each problem
4. Group similar causes and add up the score for those groups
5. Having identified the most common types of problems, take targeted action on them

A Pareto's analysis will lead to this type of line diagram:



When looking at the example above "Pareto's Analysis", we see with this method, the consideration of testing everything is impossible becomes visually clear. It's important to consider "when the software is fit for its purpose".

Root Cause Analysis

Root cause analysis is the identification of the initial cause of defects, incidents, or problems. By recording the root causes and reviewing them periodically, you can take a look at the types of errors that occur most often and how you could avoid them in the future, which is a powerful tool that can greatly improve your software development process. Applying Pareto's Analysis to the root causes that are grouped, could be beneficial when performing root cause analysis as it helps to solve 80% of the common issues in generally 20% of the effort.

Below is an overview of common root causes that could be considered when applying root cause analysis:

Category cause	Main cause	Description
Code	Error in code	The functionality is not working because of an error in the code.
	Built differently than designed	The built functionality differs from the description in the design.
	Built more than designed	The built functionality was not described in the design.
	Text & Layout	Errors in spelling, punctuation, design, alignment and usability.
	Forgot to execute	Some of the functionality was forgotten to be developed.
	Unintended modification	Modified code unintentionally changed functionality.
	Merge/integrate error	Due to an integration of different systems, an error occurred.
Design	Design not updated	The draft was not updated in a timely manner.
	Interpretation error	The description of functionality in the design was misinterpreted by the developer.
	Design gaps	The design did not describe a necessary component of functionality.
	Unclear specification	The description of functionality is not clear enough in the design.
	Incorrect specification	There is an error in the specification.
Environment	Hardware	The finding was caused by hardware problems.
	Software	The finding was caused by software problems.
	Configuration	The finding was caused by misconfiguration of (parts of) the test environment.
Test	Incorrect test data	The test data used to run the test case was incorrect.
	Incorrect test case	The test case performed is incorrect.
Unknown		Cause not yet clear.

Regardless of how you (or your team) decide to develop your risk analysis and test strategy, it is most important to remember that testing everything is impossible. This is why we need to be sure that we are focussing most of our testing efforts on the most important and valuable parts of our software.

Definitions

API	Application Programming Interface (API) is a set of rules and tools that allows different software programs to communicate and collaborate with each other.
Complexity	The degree to which a component or system has a design and/or internal structure that is difficult to understand, maintain and verify [V].
Damage	The adverse consequence of an event.
Failure Probability	The statistical likelihood of something going wrong.
Frequency	How often something happens or occurs within a certain period of time.
REST	Representational State Transfer (REST) is a form of an API where computers can communicate with each other in a simple and standardized way.
Risk	A factor that could result in future negative consequences.
Risk Assessment	The process of risk identification, analysis, and mitigation.
Risk Matrix	The graphical way to evaluate the potential risks that is used to decide on the required level of coverage.
Root Cause Analysis	An analysis technique aimed at identifying the initial cause

	of errors (defects).
SOAP	Simple Object Access Protocol (SOAP) is a standardized method through which computers can communicate with each other.
Test Basis	The foundation on which tests are based, including documentation, an existing system, requirements, or the knowledge of a person who knows how it works.
Test Object	The product to be tested.
Test Objective(s)	The part(s) of the product to be tested.
Test Strategy	The overall tactical guidelines that explains all the ins and outs of everything that needs to be tested throughout the SDLC.

Practice 4: Be Specific

LO17	Apply the practice of being specific. (K3)

When designing test cases, requirements, and functional designs, it is very important to be precise. Where programming could be considered an exact science (breaking down everything into 1s and 0s), this is not always the case for testing, as we will see in the following chapters. It is important to invest this time, to have a strong basis for development and testing going forward.

Being specific ensures that you (as a tester) make insightful choices. If done correctly, the practice of "being specific" also ensures that you are describing the correct things accurately with the appropriate amount of detail required. If the requirements you received were not specific, you can make them specific by translating them into test cases. If there are any questions or ambiguities that are raised while analyzing the requirements and translating them into test cases, should be clarified with the author of the given requirements. This kind of communication can lead to a better understanding and thereby more accurate implementation of requirements.

One trick you can use when being specific is SMART, which stands for "specific," "measurable," "acceptable," "realistic," and "time-bound". The table in the defintions section of this chapter explains the components of SMART **[VII]**.

In some cases, it might be difficult or nearly impossible to further clarify "vague" requirements or test cases. If this is the case, one thing you can do is describe the risk of the requirement that you cannot properly test or the test case that you cannot properly describe. In other words, you should specify exactly what you cannot test or measure because your inability to do so increases the risk. For example, if you are required to determine that a screen background remains "purple" when a given action occurs and you have not been provided with the specific RGB color parameters, how can you determine that the screen background is in the correct shade of purple? If you were to accept all shades of purple (because you did not request the exact color parameters from the client), you may indeed fulfill the initial requirement, but have missed the point of it, as you would accept the entire understood spectrum of purple colors, and the particular purple is probably important to the corporate identity of the client.

Example Business Requirement:

Requirement:
"Workflows will be created for the various letters and emails. These workflows can be executed based on selections made using the advanced search."

There is a lot of information that is unclear in the above requirement. The "vague" information could be clarified with the following questions (among potential others):

- *How many workflows are created?*

- *How many letters are there?*
- *How many emails are there?*
- *How many selections are there?*
- *What is a workflow?*
- *Which workflows lead to which letters?*
- *Which workflows lead to which emails?*
- *Do all workflows lead only to letters?*
- *Do all workflows lead only to emails?*
- *Where can I start the workflows?*
- *How does starting the workflows work?*
- *What is the relationship between selections and advanced search?*

By asking these questions, you will gain a much better understanding of what the requirement is asking, so you can better understand what you should test. This would help the given requirement meet the SMART standards as explained above.

The following text is an improvement of this particular requirement:

- *A number of workflows can be created in the application. A workflow is part of the standard package and can be configured for specific use. There are three workflows: workflow A, workflow B, and workflow C. Workflow A is for the acceptance letter or email (if the email is known), workflow B is for the rejection letter, and workflow C is for sending the invoice.*
- *These workflows can be executed by the "Customer Contact Handler" role from screen "CER012". This screen shows all workflows. By pressing the button "Start workflow" the workflow is activated. There will be three buttons, 1 button for each workflow. The workflows may be active for a maximum of one minute.*
- *By pressing the button "Start workflows" a pre-defined selection in the advanced search is activated. These selections are as follows: for workflow A: NAME DATA or EMAIL and NAME, LETTER DATE18, SIGNATURE4; for workflow B: NAME DATA, LETTER DATE14, SIGNATURE2; for workflow C: NAME DATA, LETTER DATE11, FINANCIAL DATA, SIGNATURE1.*
- *If a workflow cannot be executed or if a workflow is active for more than one minute, an error message with the following text: "workflow X could not be executed" should be shown to the user who started the workflow. At the position of the X, the name of the workflow in question should be shown.*

The above text may take more time to generate, but provides specific information that is much easier to accurately understand and develop. This will also make it much easier to know what should be tested. This will most likely save project development time overall, as the product is more likely to fit the precise demands of the client.

Being Specific: While Reporting a Defect

When writing a defect report, it is important to be as specific as possible so the defects can be reproduced or retested by other people.

A well-specified defect report should contain:

- A clear title
- A SMART description of the steps to reproduce the defect
- Expected result
- Actual generated result
- Proof of the defect: logging, screenshots, etc.
- Relevant information: a unique number that identifies the associated test case(s), the name of the screen, traceability (or reference) to the given requirement, which environment the defect occurs in, the version number of the software used.
- Severity of defect: “Blocking”, “Severe”, “Not Severe”, “Cosmetic”
- Priority of the defect: “Low”, “Medium”, “High”

As multiple parties are usually involved in the creation and testing of new software throughout the SDLC, it is important to maintain strong communication standards, such as being specific. This helps to allow team members to save valuable time (and resources), and accomplish tasks quicker all the while maintaining motivation throughout the project.

Definitions

Specific	Be as detailed as possible and avoid generalities. Avoid reference words like this, these, that, and those because if a text is copied or modified, the meaning may not be clear enough. Always formulate positively and singularly. Consider formulating error paths.
Measurable	The requirement or test case must be measurable. Avoid vague clauses like “fast,” “instant” “many” “easy” or “similar”. Instead, link a measurable unit to each case so you can determine if it is correct. For example, “the data in screen *CER012* must be fully loaded within 1 second after opening the screen or requesting a record” or “the process should be able to compute 100,000 transactions daily basis between the hours of 23:00 and 5:00.”
Acceptable	A requirement or test case must always be accepted by the client. Also, requirements must be ethically and morally acceptable and meet legal requirements and regulations.
Realistic	A requirement or test case must always be realistic. In other words, achievable goals must be formulated for the range of time, finances, and resources available.
Time-bound	Many requirements lack a time indication. By making time parameters concrete, you can better measure and thus better test whether a requirement is satisfactory. Include clearly defined units of time such as days, hours, minutes, seconds, milliseconds, etc.

Practice 5: Test as Early as Possible

LO18	Understand the practice of testing as early as possible. (K2)
LO19	Types of environment including testing (K2)
LO20	Understand the importance of a good testing environment (K2)
LO21	Understand the importance of test data (K2)

The practice of testing as early as possible allows for significant time and cost benefits. This practice has its roots in research done by Barry Boehm in 1977 **[IX]**. Boehm's study shows that the cost of a defect increases exponentially the later the defect is found. Ergo, the earlier defects are found, the cheaper they are to fix. Consider the following graph which shows the costs of correcting defects in different software development lifecycle (SDLC) phases:

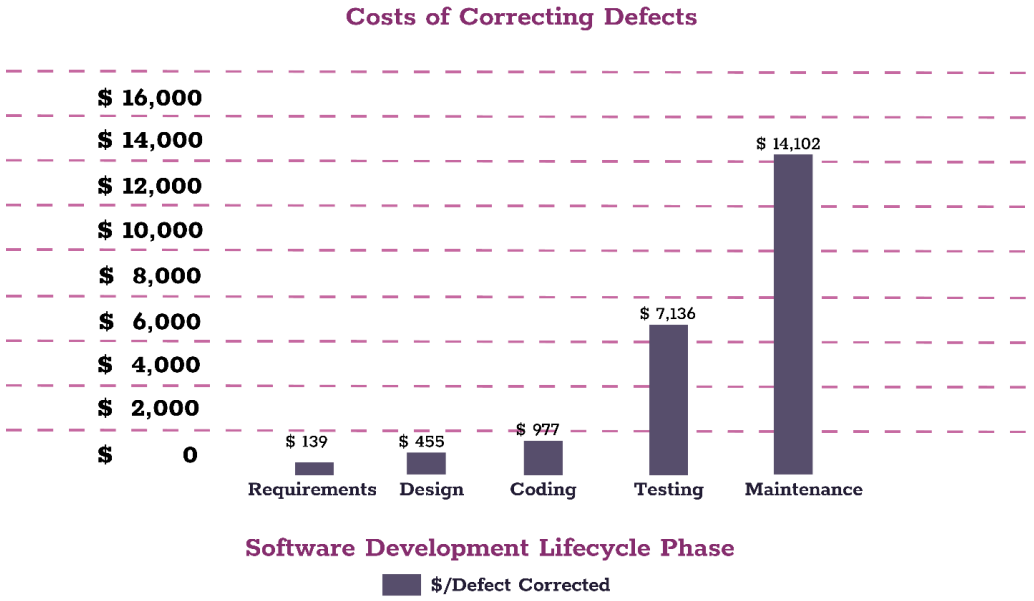


Figure: Cost of Correcting Defects

As we can see above, the correction costs grow exponentially throughout the SDLC. Therefore, if a defect is found and corrected in the early phases of the SDLC, one can imagine the positive consequences it can have on the overall time and budget of the project.

Knowing this, efforts to prevent defects should be started as early as possible. Here are some examples of how you can implement this practise of testing as early as possible:

- If you (or your team) can not test dynamically, then you can try it statically by reviewing (by yourself or with several people in a review meeting), in a walkthrough, or with an early demo of your product.
- A system test in the development environment is always cheaper and faster than testing something at a later stage, for example in an acceptance environment. It is in your best interest to test dynamically as soon as you can, for example, to avoid having the defects even deployed to the acceptance environment in the first place.

- Do not postpone any testing activities with the intention “we will come back to test this later”. Instead, by testing early, even if it means you may have to repeat some of of the tests later, this can avoid the building upon defective code which would could remain undetected for a longer period of time.

It may be tempting and even at time “more fun” to start developing early; however, remembering to review requirements can be incredibly cost-effective.

Newer research than Boehm's also confirms this. So, as a developer or administrator, be aware of this. Other measures you could apply here include having users review requirements, conducting automated static code reviews, and performing unit integration testing with fellow developers. As an example, consider the integration between the front and back end of a web application, which can often be done in the development environment but may only take place later in practice

Another important point to consider, which is often underestimated throughout the SDLC, is making sure that the test environment is well-functioning and well-managed.

Test Environments

Usually, there are four main types of testing environments (or four main types of environments in which testing is carried out). The environment that always comes first is the development environment.

The **development environment** is the first level and is used by the developers to develop the software. In the development environment, typically, technical testing like unit testing takes place. The development environment is usually only available to the developers and often has little or no integration with external software components or parties.

The second level is usually the **test environment**. The test environment is intended to be used for technical testing, for example, system testing and integration testing. Therefore, this environment needs more external components so integration testing can be executed. This environment is mainly used by dedicated testers, however, developers and maintainers could have access as well.

The third level is usually the **acceptance environment**. The acceptance environment is usually more extensive than the development and/or test environment, where almost all relevant components from the landscape are present. The main goal of the acceptance environment is to have the newly built or changed software accepted, meaning that testing is usually done by end users. Since it is often the most complete environment before the production stage, some administrators tend to use it for installation testing and other maintenance-related testing.

The final level is the **production environment**. Generally, it is not recommended to test in the production environment as this could impact the real users in real-time, however, there might

be one or two exceptions to this rule. For example, building a completely new system might require performance testing. The difficulty with performance testing is that you need to have a technical similarity to production. Technical similarity is necessary because the calculation of the load tends to be correlated with the quantity/quantities of technical components used in the environment (hardware, software, switches, CPU, memory). The environment that includes all of these components in the right proportions is the production environment. The technical similarity being key in performance is what makes it difficult to achieve this in other environments. When it comes to testing in the production environment, it is always important to be extremely cautious of the risks involved. Whenever possible, it is always recommended to run any and all tests in the environments before production.

These four environments (Development, Test, Acceptance, and Production) are often referred to as the **DTAP principle**. The DTAP principle involved the creation of software and then ensuring that it is deployed to every environment in order (Development → Test → Acceptance → Production), only entering production once it has been in every previous environment. By following this principle we can be sure that all software versions in the given environments are the same (except for changes that are in the process of being developed or tested).

DTAP in Agile and DevOps

The above paragraph is described as a situation toward which many organizations with an initial maturity level in IT and/or testing could grow. It expands your possibilities to test extensively and comprehensively early in the software development process. However, that doesn't mean the DTAP model is mandatory. If there are enough opportunities to execute the test levels defined in your test strategy in fewer environments, as seen in Agile teams, that can also provide a good solution.

There are also organizations that follow a DevOps approach. DevOps is a methodology where Development and Operations are handled within one team, and users are also represented in the teams. This sometimes leads to situations where the DTAP model is no longer fully utilized. Consider, for example, app development where new features are rolled out to production for a limited number of users, and features can be easily rolled back in case of issues. It is not always necessary to have a complete DTAP pipeline. The choice of an DTAP pipeline will always be a good balance between costs, benefits, and risks.

Understand the Importance of a Good Testing Environment

One or more properly proportioned and well-managed test environments can greatly increase the quality of your software by enabling you to test quickly, accurately, and frequently. In good testing environments, you can eliminate many assumptions that might not otherwise be encountered until production.

A test environment is important because several practices come together. In the test environment, requirements, deployed code, test cases, test data, and other necessary software are merged. If the test environment does not exist, or if parts are missing, the software cannot

be tested and therefore assumptions will have to be made (for example, that some of the components are working), which is contrary to *Practice 1: Make No Assumptions.*

To test as early as possible, you want to make sure that your test environment is as complete as possible to avoid late-stage testing in a higher-level environment. Higher-level environments are more extensive in the SDLC. Therefore, these higher-level environments tend to have more dependencies, which can severely limit their availability (time-wise) for testing at a later stage. The limited availability of higher-level environments can result from many things, such as internal project dependencies or usage by other projects. Even the fact that the time schedule becomes tighter closer to the release could be an important factor.

There are sometimes understandable financial reasons for not creating a separate test environment. However, it is important to keep in mind that all defects that are not found or are found later also cost money. Some of these costs might not be immediately apparent (relating to extra fixes and rollouts), but may come much later when the software is in production and reputations, sales, and overall quality has been damaged. When the risks of a missing test environment are made clear, many clients are willing to invest more, even without a completely sound business case. It is highly recommended to make the risks as concrete as possible and describe this urgency well.

Using Stubs and Drivers

To solve the problem of not having all components available in lower-level environments we could utilize **stubs and drivers** to simulate those components. Simulation of the missing system parts enables you to test more extensively at an early stage and will therefore enable you to find defects earlier in the SDLC. Stubs and drivers are, depending on the situation and technology, relatively simple to create and can therefore be viable to the business case when assessing the savings on the correction costs of defects at a later stage.

Key considerations about test environments:

- The omission of a test environment will always lead to a risk. Risks can also be generated even if only a portion of the test environment is omitted.
- Risks must be made clear to the person who is ultimately responsible (the client).
- Test environments should resemble the production environment as accurately as possible.
- Remember to utilize stubs and drivers in place of missing components.
- Respect the DTAP principle, and enforce it (technically) whenever possible.
- Establish configuration management and maintain the environments appropriately. It is also important to maintain configuration management of relevant components.
- Wherever possible, always automate the transfer of software from one environment to another. This can prevent unnecessary manual actions which can harm the quality and predictability of your deployments.
- It is beneficial to automate the creation of environments. In doing so, you only need to manage infrastructure components once, which allows for the easy creation of new

environments-specific changes or releases that can be easily discarded. This is especially recommended when working with cloud environments.

When it comes to the test environments, there are endless problems and situations to be encountered. We can imagine a test environment as a laboratory setup: a substance (Software) is being made in a very precise process (SDLC). You need to have the right amount of original substances that need to be used correctly: they need to be heated at exactly the right temperature, vaporized, etc. In the process, it is not possible to take some of the original substance out because it would affect the final product.

Test Data

The quality of the test data available largely determines the quality of your testing, so having good-quality test data is essential. Whenever possible it is beneficial to use production data. However, if dealing with production data, remember to consider local legislation, such as GDPR adherence in Europe.

From a testing perspective, testing with production data is beneficial because it provides many variations of functional test cases. Using a wider variety of data can be a thorough and efficient way to test the software at hand. Whether production data is possible or not, it is important to make sure that all the functional variants are present in the test data, for example, if you are testing the software of a car insurance policy, data with all the variations in address, type of car, claim-free years, etc. is essential. To be effective and efficient, it is important to test as early as possible, so we encounter defects earlier and avoid carrying them throughout the SDLC and becoming more costly defects later.

Definitions

DTAP Principle	DTAP stands for Develop, Test, Accept, Production, a principle in which the software passes through all of these environments before being put into use.
Review	A form of static testing where a (intermediate) product from the SDLC is assessed by one or more people.
Stubs and Drivers	A temporarily simulated component that is used to test another software component. A driver is placed before a component and a stub comes after it.
Test Data	The data used for the execution of tests.
Test Environment	A validated, usable and stable environment, resembling as much as possible the final production environment, used to run test cases or reproduce bugs.

Practice 6: Start Small and Gradually Expand your Testing Scope

LO22	Understand the practice of starting small and gradually expanding your testing scope. (K2)
LO23	Recall test levels and test types (K1)
LO24	Understand how test levels and test types apply to a test strategy (K2)

The concept of “starting small and gradually expanding” is also considered a best practice of project management. This also reigns true when we consider the most advantageous way to test software. The practice assumes that you have completely checked a small part (unit) of the software to ensure it is working properly. Units confirmed to be working properly can be integrated with another unit of the software, provided that the other unit has already been checked independently, which increases your scope. This has as advantage that defects that come out of the integration of the two units must result from the integration itself. This simplifies the investigation when looking for the cause of defects found. This practice enforces the “Practice 5: Test as early as possible”, as you can test earlier and more completely if you divide the system in smaller units to test.

In general, there are four test levels:

- Test Level 1 - Unit Testing
- Test Level 2 - Integration Testing
- Test Level 3 - System Testing
- Test Level 4 - Acceptance Testing

Test Level 1 - Unit Testing

The practice of testing unit by unit is called **unit testing**, and according to international testing standards (like the ISTQB) is the first **test level**. It is common practice that unit testing is performed by developers in the development environment. Unit testing is executed dynamically and it is done towards the completion of the development of the unit, while the software is still being developed. The scope and coverage of unit testing are thus always limited to the unit itself.

Test Level 2 - Integration Testing

The next test level involves the testing of two units together, which is called **integration testing**. These tests can only generate defects that have to do with the integration of the two units, since defects in both units should have been found seperately in the previous unit tests. Following this process enable error detection to be considerably easier. Note: where we use the term “unit,” we could also use “component”. As integration testing focuses primarily on the interaction between two components or systems, the scope should cover all possible interactions (an situations thereof) between the two units.

Test Level 3 - System Testing

System testing is the test level that is generally recommended to follow after integration testing. System testing is usually carried out in the test environment and involves the testing of different integrations together. System testing may sometimes involve a chain of processes, which is referred to as **chain testing**. The scope of system testing is aimed towards the flow and/or consistency of different integrations and units of the system.

Test Level 4 - Acceptance Testing

The next level that is generally recommended is called **acceptance testing**. Acceptance testing is usually carried out by users from within the organization the system is developed for. The scope of acceptance testing is to confirm if the system supports the users and organizational processes as intended. **End-to-End testing** (E2E testing) is a form of testing that tests (the main scenarios of) the flow of a system from the beginning to the end. E2E tests are usually carried out at the acceptance level of the SDLC where a lot of dependencies exist.

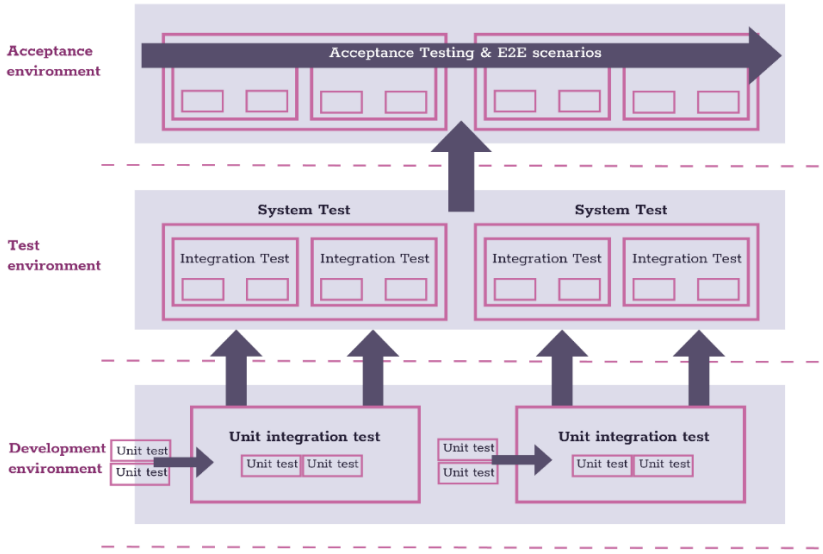


Figure: Test Levels vs. Test Environment

In the figure: Test Levels vs. Test Environment, each box (unit) represents a test. When testing has been completed on one box, then you merge the tested box with another (tested) box. This continues following the test levels (1-4) as previously explained that build upon one another.

System Integration Testing

An often-used test level that is in between system and integration testing is called **system integration testing** (SIT). Software that requires integration with the software of other teams or organizations demand special attention. When software components are developed separately, chances are higher that misunderstandings or miscommunications can occur. These communication mishaps can lead to parts of the software being interpreted differently and harm the overall quality of the software.

End-to-End Testing

We've already seen how starting small and gradually expanding your testing scope makes error detection easier, but the principle of early testing is also applicable here. An E2E test is a late and therefore relatively expensive form of testing, so this should be executed as smoothly and efficiently as possible. E2E tests should only find errors that come from the integration of all the process components. Therefore, it is in your best interest to have the earlier test types fully executed before you start carrying out E2E testing.

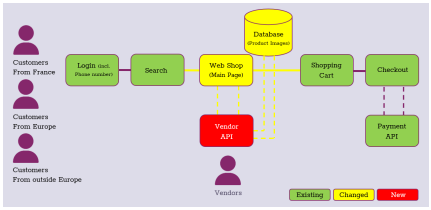
Test Types

Test types are focused on specific testing objectives as well as specific characteristics of a component or system **[V]**. They usually depict certain quality characteristics (functionality, security, performance, usability, etc.), but can also focus on concepts like regression or changes. Some of these test types will be elaborated on in the following chapter focusing on quality characteristics.

The various test types can be executed at any of the four test levels, but it is useful to consider that certain test types are more beneficial in certain environments. Test types are usually decided on after the risk analysis or the analysis of the test basis when it becomes clear that certain aspects of the system require special attention.

Implementing Test Levels and Test Types in Test Strategy

When continuing our example from practice 3 and constructing our test strategy we should analyze the different parts of the system (test objectives), so we can decide which test levels we would like to apply to each. Considering our previous example sketch, our test objectives are the various components: "Customers", "Login", "Search", "Webshop", … etc.



Reminder Sketch from Practice 3: Testing Everything is Impossible

To apply the different test levels and test types to our test objectives, it is beneficial to create a test strategy table that utilizes the information gained in our risk analysis to determine the coverage for each test objective for each test level. It is important to consider which test types are relevant when testing your software at various levels.

For our given example our test strategy could look like this:

Test strategy table		**Test levels**

Test objective/test type	Static Testing	Unit test	Integration	System	Acceptance
Orders from France	Code review	High Coverage	High Coverage	Medium Coverage	Include in all E2E scenarios
Orders from other parts of Europe		Medium Coverage	Medium Coverage	Low Coverage	Test main Scenarios
Orders from other parts of the World		Low Coverage	Low Coverage	Low Coverage	Test some scenarios
Login	..	..	..	..	..
Search	..	..	..	..	..
Usability	Prototyping	Walkthrough		Eye tracking	Test by user groups
API Documentation	Review				

Coverage

In the test strategy table, we have now determined the coverage for different processes. This coverage is divided into 'High,' 'Medium,' and 'Low.' By using various test techniques, we can achieve this desired coverage. In 'Practice 7: Document Your Tests,' we will pay attention to this.

Test strategies are a very extensive topic and your knowledge will grow throughout your testing career. If you remain focused and “start small and gradually expand your testing scope” you will already be on the right path to creating well-focused and structured test strategies in your everyday work.

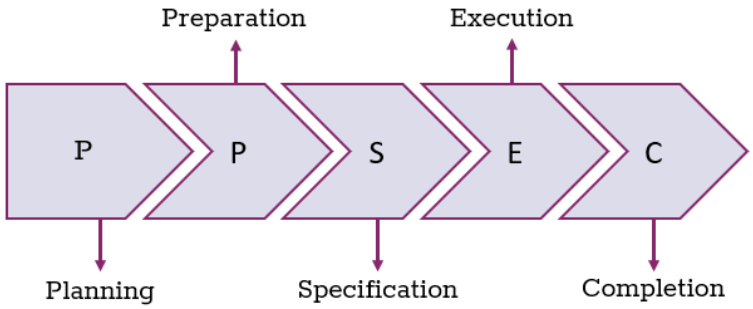
Definitions

Acceptance Test	A test level aimed at determining whether the system is accepted. **[V]**
Chain Test	A type of test that includes multiple systems.
End-to-End test (E2E Test)	A type of test in which business processes are tested from start to finish under production-like conditions. **[XXI]**
Integration Test	A test level that focuses on interactions between components or systems. **[V]**
System Test	A test level designed to verify that a system as a whole meets the specified requirements. **[V]**
Test Level	A specific representation of a testing process. **[V]**
Test Type	A collection of testing activities based on specific testing objectives and focused on specific characteristics of a component or system, e.g., a performance or security test or a user acceptance test. **[V]**
Unit/Component Test	A test level that focuses on individual hardware or software components. **[V]**

Practice 7: Documenting your Tests

LO25	Recall a basic testing process (K1)
LO26	Understand the value of documenting your testing. (K2)

The existing recognized test methodologies distinguish several different phases in the testing process, usually: planning, preparation, specification, execution, and completion.



Testing Process

Generally, 60-70% of the time spent on testing is devoted to the first three processes (planning, preparation, and specification) **[XII]**. The execution and completion phases of the tests themselves often only constitute 20-30% of the time. These percentages indicate that a considerable amount of time goes into planning, preparation, and specification.

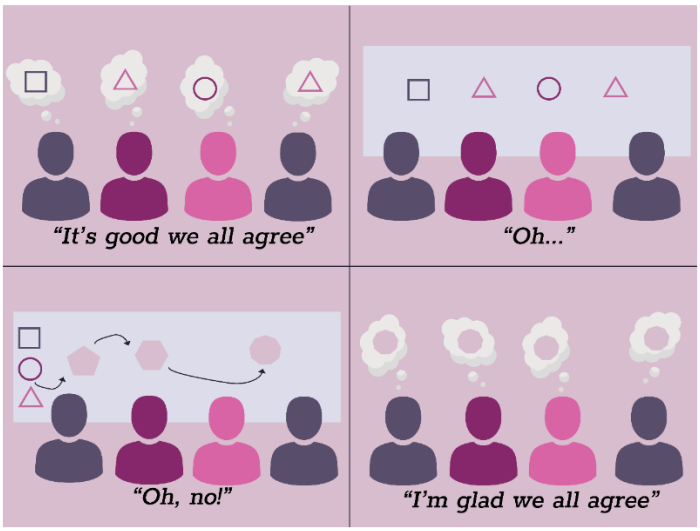
This preparation phase is crucial as it consists of selecting and proposing appropriate tests. This is where important aspects of the entire testing process are decided, which includes testing the right things and testing them at the right time.

It is sometimes required to put together a comprehensive plan with a detailed test strategy, in which agreements and choices are outlined in detail concerning the entire testing project. This can become quite an extensive activity, with a overal test plan for all test types and detailed test plans for each test type, as well as agreements about time, resources, schedules, budget, etc. Extensive plans can be valuable in certain circumstances, especially in environments like large corporations or government institutions. Comprehensive planning can also be worthwhile if you have many different testing parties and you want to agree on which party tests which items to avoid overlap or duplication of tests, or you want to prevent certain parties from testing a certain area.

However, please note that such extensive plans should not be made out of habit or because that is the way of working in a particular organization: this sort of planning should only be done

when it adds value. It should never be the priority to create an overly extensive overal test plan simply to have it, as it most likely will not be read.

In other circumstances, a less extensive test strategy of only a few pages will suffice. These briefer outlines should document your thoughts and discussions with co-workers, and they serve their purpose because they enable you to communicate or refer to it more easily. This type of documentation also helps to prevent misinterpretations of the agreements.



Hence the practice: "Documenting your testing." The meaning of this practice is twofold: first, document your test strategy. Second, document the test cases you are going to test or have tested. In order to document testing, it helps if you also have documented requirements. This might be obvious, but it is not always common practice. Documenting requirements is an excellent quality measure. By documenting requirements, you make the information transferrable from person to person, you avoid misinterpretation, and you can create a common understanding of the requirements. The requirements can then be more easily refined and used as a basis for testing. Therefore, it is always good practice to request that clients document requirements and to offer your assistance in doing so. This action is a single measure that can add considerable quality.

A good way to formulate requirements is to turn them into user stories in the format: *As <role> I want <functionality> so that I <add value>*. For example:

*'**AS** an administrator **I WANT** to be able to add a .PDF file to a website*
***SO THAT** I can show this .PDF file to vistors of the website'* **[X]**

By formulating requirements in this way, the practice of "documenting your tests", along with the practice of "being specific", will allow us to make understandable requirements. These requirements subsequently provide the guidelines to develop, test, and maintain software.

The amount of detail in which requirements are documented will vary in different organizations and projects. The amount of detail provided in our requirements will determine how rigorous and detailed our tests could be, something also to consider when constructing a test strategy and selecting test techniques to test certain parts of the system.

When documenting tests that have been derived from the test basis, this offers a structured approach, which can ensure that we do not forget relevant test cases. Documenting exactly what you have tested makes testing reproducible and repeatable. Replicability is a part of testing that is very important. Often, someone else (e.g., the developer or the test coordinator) has to reproduce your test to witness or experience a given defect. By documenting exactly what you did, you make this easier and faster for them to analyze the system, which can save time and money.

By documenting what you have tested, you make it easier to repeat those tests yourself. Remembering what you did is sometimes virtually impossible given the complexity of the situation or the number of test cases and test rounds you did. Making screen recordings as you run your tests is also a good way to capture test execution and results, provided that it is a clear and relevant recording, and that you can easily search for and recover the different test cases performed.

There is yet another important reason to document your tests. Often, you want to create a clear script in preparation for testing that will be used during test execution. For example, test execution can only start when the software is delivered to the test environment. This could be days or weeks after the creation of the test cases. So, by creating the test script earlier, before execution, the tests can be run independently by team members (or anyone) other than the author.

A final reason for documenting is as follows: To have a precise and accure outcome prediction. We should clearly write our prediction upfront. That way it will not be biased by the actual outcome of the test. And it would be more likely to notice deviations from the outcome prediction. By documenting, you are likely more likely to avoid putting the behaviour of the product during testing into the script. Consider that the behavior of the product during testing is not necessarily the correct behavior. If the expected behavior is established beforehand, the chances of including an undesired behavior in the test script are lower, and defects are more likely to be detected.

For example, think of constructing an item from IKEA. When you carefully follow the instructions, the right outcome (the correct construction of the item) results. If you just start putting bits and pieces together without following the instructions, you stand a significant chance of ending up with some items attached upside down or inside out.

Using Test Techniques

There are various ways of creating test designs and test scripts, called “test techniques.” Various test techniques apply to different situations, tests, and software components, some of which are discussed below. Several important things to consider when applying test techniques are listed below. These apply when using a test technique and creating a test script.

- Apply the test practices.
- Design your test cases before you execute them, rather than doing this at the same time.
- Use a technique to create a high-level test design.
- From the high-level test design, add more detail when creating the test case(s).
- Linguistically, test cases should be formulated singularly.
- Separate a logical test case from a physical test case.
- Consider initial conditions (pre-conditions).
- Document an output prediction.
- Document the result.

The items listed above will be explained in more detail in each of the sections below, which focus on various test techniques.

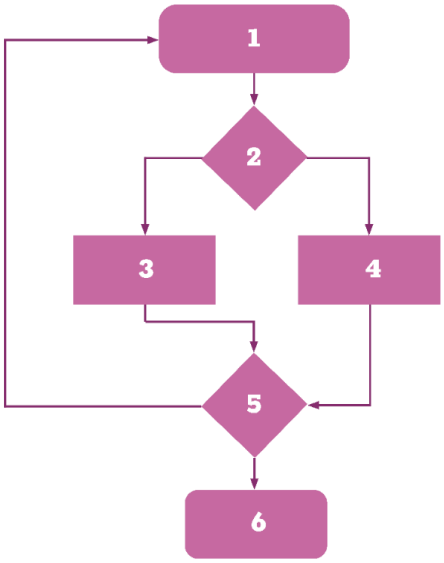
Process Flow Test

LO27	Learn to create a test script(s) by drawing up the process/program. (K3)

The first test technique we are going to discuss is called the “process flow” technique **[XI]**. This technique is useful for testing flows in code, applications, and processes or chains of processes.

Process flow tests allow you to outline the structure of a program or process, including decision moments as well as alternative and error paths. The technique provides insight into the paths within a process or program. This allows you to check whether the main flow of the most important decisions are working correctly as well as the most important error paths. It might be worth it to mention (though it might seem needless) that such a process diagram can also be drawn based on the program code.

Example Process Flow:



Description:

1. Open mobile app & show home screen.
2. Is the customer an existing customer.
3. Yes, show login screen.
4. No, show account creation screen.
5. Login(Create account & Login) succesfull?
   Yes -> 6 No -> back to 1.
6. Show continuation screen.

Steps to Construct a Process Flow Diagram

Start by analyzing the information, process, or program code. In doing so, you can draw a diagram similar to the example process flow. In the process of diagramming, you might encounter difficulties in connecting the lines, which could mean you have encountered a defect. While putting together the diagram, try to be as extensive as possible. In the sections where information is missing, or where you cannot tie the lines together, gather information by asking team members and/or the client about the intended behavior of the missing part or path.

In the process flow diagram, a rectangle signifies an action and a diamond signifies a decision. A line with an arrow is drawn from the action and decision to the follow-up action and decision. By drawing it this way, it is easy to determine what paths exist in the process or code and whether you can test them.

For this example, imagine a mobile app, and that you want to test the login process. Schematically, it is represented in this previous process flow diagram.

Drawing a diagram this way enables you to clarify which flows exist and which ones you can test. The next step, labeled “Description” in this example, includes a written version in understandable language. This makes the process even clearer and should make it possible for anyone on the team to execute the test as well.

Creating the Test Script

When creating the test script, the first step is to write out all the high-level steps of the different flows deduced from the diagram.

Test case	Schema	Description
1.	1-2-3-5-6	Successful login attempt; existing customer.
2.	1-2-3-5-1	Unsuccessful login attempt; existing customer.
3.	1-2-4-5-6	Successful login attempt; new customer.
4.	1-2-4-5-1	Unsuccessful login attempt; new customer.

Once these steps have been outlined, you have created high-level test cases. These test cases are called “logical test cases,” since they solely describe the logic used to construct them. After this has been completed, you can then create the detailed steps and write down the detailed test cases.

Test Case 1, Successful login attempt; existing customer	
Step 1. Open the screen with an existing customer. (pre-condition)	
Step 2. Show login screen. (pre-condition)	
Step 3. Log in with valid credentials. (pre-condition)	
Test: that the follow-up screen is displayed (outcome prediction).	
(Expected) Result	Ok/Not Ok

With this amount of detail, it is easy to see the different parts of the test case. These different parts are initial conditions or “pre-conditions.” In this test case, the pre-conditions are steps 1-3: they describe the conditions to fulfill that enable the execution the test case. The next part is the test itself, with the outcome prediction. The final part of the test case is the (expected) result, in which the conclusion of the outcome prediction would be true (Ok) or false (Not Ok). We have now explained a detailed test case. This test case, like its higher-level version, is still called a logical test case. To be thorough, the other detailed test cases are as follows:

Test Case 2, Unsuccessful login attempt; existing customer	
Step 1. Open the screen with an existing customer.	
Step 2. Show login screen.	
Step 3. Log in with invalid credentials.	
Test: you are returned to the home screen (outcome prediction).	
(Expected) Result	Ok/Not Ok

Test Case 3, Unsuccessful login attempt; new customer	
Step 1. Open the new client screen.	
Step 2. Show login screen.	
Step 3. Log in with invalid credentials.	
Test: you are indeed returned to the home screen (outcome prediction).	
(Expected) Result	Ok/Not Ok

Test Case 4, Successful login attempt; new customer	
Step 1. Open the new client screen.	
Step 2. Show login screen.	
Step 3. Log in with valid credentials.	
Test: that the follow-up screen is displayed (outcome prediction).	
(Expected) Result	Ok/Not Ok

Once the test cases have been described, they can be executed, but in order to do this, you have to search in the database for the right data to implement. In the example case, this is the customer data. This can be done with existing data, or data manipulated to fulfill the pre-conditions of the test cases. Test cases in which the right data has been implemented are called “physical test cases”.

Semantic Test Technique

LO28	Learn to create a test script using the semantic test. (K3)

Another test technique you can use to create test cases is the semantic test technique **[XII]**. This technique uses the words IF, THEN, and ELSE to describe test cases. Semantic tests are useful for testing simple and medium complex functionality. Semantic tests also lend themselves well to testing single requirements as well as requirements that are co-dependent of other requirements, e.g., different control requirements on screen like “customer’s age needs to be < 16” or “customer’s credit check needs to give a positive result”.

To give an example of usage, we will use the same sample description of functionality as we used for the process flow technique:

Description:

1. Open the mobile app and show the home screen.
2. Is the customer an existing customer?
3. Yes -> show login screen.
4. No -> show account creation screen.
5. Login successful? Yes -> 6; No -> back to 1.
6. Show the continuation screen.

Using this example, you would arrive at the following semantic test script:

1.	IF The app is opened THEN Show the home screen ELSE No action
2.	IF The customer is an existing customer

	THEN Show the login screen ELSE Show account creation screen.
3.	IF The customer is an existing customer with a successful login THEN Show the continuation screen. ELSE Show home screen.

With co-dependency, it would then become:

```
IF The app is opened
THEN Show the home screen
        IF The customer is an existing customer
        THEN Show the login screen
                IF There is a successful login attempt
                THEN Show the continuation screen
                ELSE Show home screen
    ELSE Show account creation screen
ELSE No action
```

Note the alignment here: IF, THEN, and ELSE belong together. This is called a nested statement and is also used in program code. Furthermore, note that an IF always has a THEN and an ELSE. In this manner, it is possible to describe all possibilities of the functionality.

The IF describes a pre-condition: a condition that a test case must satisfy in order to be executed. The THEN describes the action that should occur when the condition is met. Finally, the ELSE describes what should happen if the condition is not met, and refers to either an error path or a subsequent functionality. Now that we have described the test cases as high-level test cases, we can continue as we did with the process flow technique, by going into further detail:

Nr.	**High-Level test case**	**Nr.**	**Detailed test case**
1.	IF The app is opened THEN Show the home screen ELSE No action	1.	Verify that the home screen is displayed.
		2.	Check that no screen is displayed when no action is taken.
2.	IF The customer is an existing customer THEN Show the login screen ELSE Show account creation screen.	3.	Verify that the login screen is displayed with an existing customer.
		4.	Make sure the account creation screen is displayed with a new customer.
3.	IF The customer is an existing customer with a successful login	5.	Verify that upon a successful login attempt, the continuation screen is displayed.

	THEN Show the continuation screen. ELSE Show home screen.	6.	Check that upon unsuccessful login, the home screen is displayed.

Next, we can create the most detailed step(s) of the test case:

Detailed Test Case 1, Verify that the home screen is displayed.	
Step 1. Open the app	
Step 2. Show the home screen.	
Check: that the home screen is displayed (output prediction).	
Result	Ok/Not Ok

Detailed Test Case 2, Check that no screen is displayed when no action is taken.	
Step 1. Install the app, but do not open the app.	
Check: that nothing is opened (output prediction).	
Result	Ok/Not Ok

The second test case may seem redundant—this may be true in this example, but it often makes sense to explicitly check for no action.

Decision Tables

LO29	Learn to test functionality using a decision table (K3)

Decision tables **[VIII]** can be used to test the structure and logic of a program or process. In decision tables, you differentiate between conditions that are or are not met, and you describe the actions resulting from those decisions. In this manner, you can properly check for full functionality. Decision tables are a complex technique that allows thorough testing: they are often used for testing complicated processes or applied in high-risk environments where it is important to distinguish all possible outcomes.

Decision tables are constructed by writing out all the conditions. When formulating conditions in a decision table, it is important to keep a few things in mind. Conditions should be formulated in simple language and singular terms. Also, they should be formulated in such a way that the outcome of the question will be easily understandable if answered with 'Yes' or 'No'. This way, the table will be easy to understand and to replicate.

Example entry to an amusement park ride:
You have the following requirement: 'If age > 15 or height > 135, then you have access to the attraction.'

If you have a VIP ticket, you have direct access to the attraction. This leads to the decision table below.

Conditions								
Age > 15 year								
Height > 135 cm								
VIP Ticket								
Actions								
…								

The next step is to fill in all the actions resulting from the conditions. Once the actions are listed, you can fill in the decisions with 'Y' (Yes) and 'N' (No)". Excel is a good tool to help with this. In a decision table, the conditions are boolean (Y/N). The number of possible "different decisions" is therefore 2 to the power of the number of conditions. So with 3 conditions in our table, it will be 2 to the power of 3, $2^3 = 2 \times 2 \times 2 = 8$. At 4 conditions, $2^4 = 2 \times 2 \times 2 \times 2 = 16$.

Once the columns have been determined, you can fill the 'Y' and 'N' in the columns. An easy way to do this is to fill the first half of the first condition with 'Y' and the second half with 'N'. In our example, we have 3 conditions, resulting in 8 columns (2 x 2 x 2) where columns 1-4 would be 'Y' and columns 5-8 'N'

Conditions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Age > 15 year	Y	Y	Y	Y	N	N	N	N
Height > 135 cm								
VIP Ticket								
Actions								
Direct access								
Granting Access								
No access								

Once you have filled the first condition (row), you can move on to the next condition. A good rule to follow is to copy the decisions of the previous condition into the following row and change the second half of the consecutive 'Y' portions to 'N's and the first half of the consecutive 'N' portions to 'Y's as shown here:

Conditions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Age > 15 year	Y	Y	Y	Y	N	N	N	N
Height > 135 cm	Y	Y	N	N	Y	Y	N	N
VIP Ticket								
Actions								
Direct access								
Granting Access								

No access								

You can move on to the next conditions until you reach the last condition, as seen here:

Conditions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Age > 15 year	Y	Y	Y	Y	N	N	N	N
Height > 135 cm	Y	Y	N	N	Y	Y	N	N
VIP Ticket	Y	N	Y	N	Y	N	Y	N
Actions								
Direct access								
Granting Access								
No access								

Then, you can match the outcome of the different conditions to an action. In our example, a person must be either 15 years old or older or taller than 135cm to be granted access to the attraction. Furthermore, it is necessary to have a VIP ticket to gain direct access to the attraction Column 1 describes a person who is either 15 years old or older and is taller than 135cm and also has a VIP ticket. Therefore, direct access should be granted to the individual described in column 1. Write 'X' in the field that indicates the correct action.

Next, you can fill in the second column. Since the "VIP ticket" requirement is an 'N,' direct access will not be granted to an individual who meets these criteria. The individual has to wait in line for the usual access. All of the other actions in the table should now be filled out until there is an action for each column. You now have the 8 test cases to test the functionality thoroughly as seen here:

Conditions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Age > 15 year	Y	Y	Y	Y	N	N	N	N
Height > 135 cm	Y	Y	N	N	Y	Y	N	N
VIP Ticket	Y	N	Y	N	Y	N	Y	N
Actions								
Direct access	X		X		X			
Granting Access		X		X		X		
No access							X	X

Boundary Value Analysis

LO30	Learn to deepen your testing by using boundary value analysis. (K3)

Boundary value analysis **[XIV & XV]** is a method of covering the boundaries of values in input fields, conditions, APIs, batch processes, and plugins in your tests. In this method, you look

specifically for the boundary to determine distinctions in the difference in functionality used in the software. Carrying out boundary value analysis enables you to properly check for completeness and correctly working functionality.

To illustrate boundary value analysis, let's imagine using an on-demand service with age-appropriate distinction for violent content. One of the rules is that a person must be 18 years or older to view certain content. This could be checked by testing with an age randomly chosen above 18, say 23, but perhaps a better alternative would be to check this with an age of 17 (no access), 18 (access), and 19 (access). You would use these values specifically because many errors arise from wrong values for the boundaries: consider the usage of the symbols '<', '>', '=', '<=', and '>=', in programming code. A good way to apply boundary value analysis is to make 3 test cases per limit: a test case on the boundary, a test case just left (most likely less than) of the boundary, and a test case just right (most likely greater than) of the boundary.

Let's look at another example of senior citizens receiving a discount on public transport to illustrate how boundary value analysis can be applied. Take the following requirement: *'Citizens of 65 years of age and older receive 20% off their public transport ticket fee.'* When we apply boundary value analysis to this, it will lead to 3 test cases per boundary, yielding the following test cases:

- 64, 65, 66 (invalid, valid, valid)

Equivalence Partitioning

LO31	Learn to deepen your testing by using equivalence partitioning. (K3)

Equivalence partitioning, or equivalence classes **[XIII & XIV]**, is a test technique that distinguishes different classes of (input) values or other requirements for an application. Values from within the same class(es) often lead to the same kind of processing. We distinguish between "valid" and "invalid" equivalence classes. Input values from valid equivalence classes are processed correctly and input values from invalid equivalence classes could result in an error message. When applying this principle, at least one test case should be based on each separate equivalence class.

Let's take the example of children gaining access to a special play structure. An input control on age has the following requirement: 6 < age < 12 (the age should be greater than 6 but less than 12).

In this example, there are three equivalence classes:

- Age > 6 years
- Age in the range of 7 to 11 years
- Age < 12 years

Applying equivalence classes results in the following test cases:

- Age = 4 years (invalid)
- Age = 8 years (valid)
- Age = 15 years (invalid)

It is common in equivalence partitioning that there are fewer test cases than when carrying out boundary value analysis. In other words, the coverage of the test cases applied in boundary value analysis is much higher. In the play structure access example, we would have 5, 6, 7, 11, 12, and 13 as our boundaries in boundary value analysis, where we only have 3 in equivalence partitioning. Nonetheless, equivalence partitioning is useful if you want to test with slightly less coverage or if you need to test alphanumeric input values, for example, when considering various documentation used for identification: passport, driver's license, ID card, etc.

Checklist-based Test Technique

LO32	Learn how to test using a checklist. (K3)

The checklist-based test technique **[XVI]** involves the creation of a checklist to serve as the basis for testing. Examples of the practical application of this technique can be seen in repetitive activities such as delivering software to a test, acceptance, or production environment. Certain aspects of security are also issues that can easily be handled with a checklist. On the checklist, you list the checks you perform in a column next to columns with the options 'Ok' and 'Not ok' to be ticked off accordingly. Checklists can be created based on experience and can be expanded by adding the cause of previous incidents. Using checklists in processes provides predictability and consistency; checklists are also a good way to capture knowledge from very experienced employees and transfer that knowledge to junior employees.

Example of the checklist based test technique:

Nr,	Description of test case	Result (Ok/Not ok)	Note
1.	Verify that the login screen is displayed.	Ok	
2.	Make sure that the "login" field is displayed.	Ok	
3.	Make sure that the "password" field is displayed.	Ok	
4.	Verify that the entries in the "login" field with 35 positions are accepted.	Ok	
5.	Check that the entries in the "login" field with 36 positions are not accepted.	Not ok	The login field allows more than 36 positions, see defect 123.
6.	Verify that the entries in the "password" field with 35 positions are accepted.	Ok	
7.	Check that the entries in the "password" field with 36 positions are not accepted.	Not ok	The password field allows more than 36 positions, see defect 138.
8.	Check that if a user enters a password of fewer than 16 characters, the message "your password must be at least 16 characters" appears.	Ok	
9.	Check that if a user enters a password with no special characters, the message "your password must contain at least 1 of the following special characters -_()*&^%$#@!?/><" appears.	Ok	
10.	Check that if a user enters a password without a capital letter, the message "your password must contain at least 1 capital letter" appears.	Ok	
11.	Verify that a user can log in with the role "administrator"		
12.	...		

With checklists, less experienced employees can check the same things as their more experienced peers. Checklist-based tests can be combined quite well with other test techniques such as the CRUD matrix, boundary value analysis, equivalence partitioning, or the semantic test technique. Once conducted, however, a checklist should be maintained regularly as some of the repetitive actions might change over time.

Practical uses of the checklist-based test technique:

- Layout
- Input values (screens and files)
- Output values
- File and field validation
- Correct error messages
- Missing fields
- Incorrect field length
- Incorrect field type (data type)
- Wrong position

Another practical application of this technique would be checking the completeness of delivery to another environment. When an application is built, it often consists of different components and files. After creation, these components have to be gathered for deployment to the next environment. Although there are tools that can be of assistance, there might still be manual actions—and every one of these actions is a quality risk to the software, as well as to testing in the next environment. Therefore, in these situations, a checklist in which all necessary actions are meticulously described can be useful. In addition to creating a checklist, you can also utilize existing checklists to test applications. For instance, there are excellent online checklists available for security and usability that you can readily incorporate into your project.

Pairwise Testing Test Technique

LO33	Learn the test technique of pairwise testing. (K3)

Pairwise testing is a method of finding defects by combining two values of a variable**.** This test technique assumes that most defects are caused by one factor or by the interaction of two different factors. Where testing all possible combinations of input values would be impossible or very tedious, pairwise testing can be an effective way to test every combination of two random factors and encourages a better variety of test data.

Let’s take an example to illustrate how pairwise testing works. Imagine a system with three defined input variables: “file format”, “role”, and “access level”. These defined variables have several values listed in the following table:

File format	Access Level	Role
.pdf	Green	Employee
.gif	Orange	Administrator
.docx	Purple	
.jpeg		

In total, we have 4 x 3 x 2 values, so a total of 24 unique combinations for the input variables.

TC	File format	Access Level	Role
1.	.pdf	Green	Employee
2.	.pdf	Orange	Administrator
3.	.pdf	Purple	Employee
4.	.pdf	Green	Administrator
5.	.pdf	Orange	Employee
6.	.pdf	Purple	Administrator
7.	.gif	Green	Employee
8.	.gif	Orange	Administrator
9.	.gif	Purple	Employee
10.	.gif	Green	Administrator
11.	.gif	Orange	Employee
12.	.gif	Purple	Administrator
13.	.docx	Green	Employee
14.	.docx	Orange	Administrator
15.	.docx	Purple	Employee
16.	.docx	Green	Administrator
17.	.docx	Orange	Employee
18.	.docx	Purple	Administrator
19.	.jpeg	Green	Employee
20.	.jpeg	Orange	Administrator
21.	.jpeg	Purple	Employee
22.	.jpeg	Green	Administrator
23.	.jpeg	Orange	Employee
24.	.jpeg	Purple	Administrator

As applying the pairwise testing test technique involve focussing on only two input values at once. Therefore, in the first step considering only the first two columns, will result in the following table of combinations:

File format	Access Level
.pdf	Green
.pdf	Orange
.pdf	Purple
.gif	Green
.gif	Orange
.gif	Purple
.docx	Green
.docx	Orange
.docx	Purple
.jpeg	Green
.jpeg	Orange
.jpeg	Purple

Here we have each input value from column “file format” with each different input value in column “Access Level” providing twelve possibilites. As there are four different “file format” input values and three different “Access Level” input values, so 4 input values x 3 input values = 12 different test cases for now.

Once all the possible combinations for the input values of the first two columns are clear, we need to apply the same technique to the second and third columns to find out how many combinations are possible. As there are two different “Access Level” input values and three different “Role” input values, we arrive at 3 input values x 2 input values = 6 test cases as seen here:

Access Level	Rol
Green	Employee
Green	Administrator
Orange	Employee
Orange	Administrator
Purple	Employee
Purple	Administrator

One might assume at this point that you have 18 test cases; however this is not the case in pairwise testing. The value in this technique is to get the most relevant coverage possible with a smaller amount of test cases.

If we consider all of the 24 possible combinations, and now implement pairwise testing, we simply need to make sure that all the above listed combinations of the previous two table comparing two input values at a time are represented within the test cases, so all other test cases can be removed.

For example, if we highlight the possible test cases in green that statisfy the first table comparing the input values of “file format” and “Access Level” we would highlight the following:

TC	File format	Access Level	Role
1.	.pdf	Green	Employee
2.	.pdf	Green	Administrator
3.	.pdf	Purple	Employee
4.	.pdf	Purple	Administrator
5.	.pdf	Orange	Employee
6.	.pdf	Orange	Administrator
7.	.gif	Green	Employee
8.	.gif	Green	Administrator
9.	.gif	Purple	Employee
10.	.gif	Purple	Administrator
11.	.gif	Orange	Employee
12.	.gif	Orange	Administrator
13.	.docx	Green	Employee
14.	.docx	Groen	Administrator
15.	.docx	Purple	Employee
16.	.docx	Purple	Administrator
17.	.docx	Orange	Employee
18.	.docx	Orange	Administrator
19.	.jpeg	Green	Employee
20.	.jpeg	Green	Administrator
21.	.jpeg	Purple	Employee
22.	.jpeg	Purple	Administrator
23.	.jpeg	Orange	Employee
24.	.jpeg	Orange	Administrator

Now we would take the next table comparing the next two input values “Access Level” and “Role” and realize that we already have highlighted three of the six combinations. Therefore, in pairwise testing, we simply need to make sure that the other three combinations are also represented. Therefore we can highlight any of the rows that fulfill this in yellow, while trying to cover as many combinations as possible. The remaining rows can be removed as seen here:

TC	File format	Access Level	Role
1.	.pdf	Green	Employee
~~2.~~	~~.pdf~~	~~Green~~	~~Employee~~
3.	.pdf	Purple	Administrator
~~4.~~	~~.pdf~~	~~Purple~~	~~Administrator~~

5.	.pdf	Orange	Employee
~~6.~~	~~.pdf~~	~~Orange~~	~~Administrator~~
~~7.~~	~~.gif~~	~~Green~~	~~Employee~~
8.	.gif	Green	Administrator
~~9.~~	~~.gif~~	~~Purple~~	~~Employee~~
10.	.gif	Purple	Administrator
~~11.~~	~~.gif~~	~~Orange~~	~~Employee~~
12.	.gif	Orange	Administrator

Although we have already fully applied the principle of pairwise testing, we can do the same with test cases 13 to 24. Here, too, we combine the unique combinations of the 'file format' and 'access level' columns with the unique combinations of the 'access level' and 'authorization' columns. This leads to the following table in which four test cases ('14', '16', '18', '19', '21', '23') can be removed:

13.	.docx	Green	Employee
~~14.~~	~~.docx~~	~~Green~~	~~Administrator~~
15.	.docx	Purple	Employee
~~16.~~	~~.docx~~	~~Purple~~	~~Administrator~~
17.	.docx	Orange	Employee
~~18.~~	~~.docx~~	~~Orange~~	~~Administrator~~
~~19.~~	~~.jpeg~~	~~Green~~	~~Employee~~
20.	.jpeg	Green	Administrator
~~21.~~	~~.jpeg~~	~~Purple~~	~~Employee~~
22.	.jpeg	Purple	Administrator
~~23.~~	~~.jpeg~~	~~Orange~~	~~Employee~~
24.	.jpeg	Orange	Administrator

This leaves the following twelve test cases:

TC	File format	Access Level	Role
1.	.pdf	Green	Employee
3.	.pdf	Purple	Employee
5.	.pdf	Orange	Employee
8.	.gif	Green	Administrator
10.	.gif	Purple	Administrator
12.	.gif	Orange	Administrator
13.	.docx	Green	Employee
15.	.docx	Purple	Employee
17.	.docx	Orange	Employee
20.	.jpeg	Green	Administrator

22.	.jpeg	Purple	Administrator
24.	.jpeg	Orange	Administrator

In this example, it becomes clear that we have test cases that cover a wide variety of test data, but we have not tested for example the combination of .pdf, orange and administrator (test case “6”).

An advantage of pairwise testing is that the coverage of your test cases might be quite high; however, all dependencies between the different variables are not tested. Also, one can imagine when applying this technique can also produce combinations that are unrealistic to the situation at hand **[XVII]**.

Relationship between coverage in the test strategy and test techniques

In Practice 6, we discussed the development of a test strategy. There is a connection between the test strategy and the application of test techniques. Where the test strategy identified different risk classes, we can now use test techniques to cover these risk classes. Some test techniques provide extensive coverage and can be well-used to cover test goals with high risk. Others provide average or low coverage and can be used for average or low risks. You can also vary within a test technique with higher, average, or lower coverage. The table below describes the characteristics of different test techniques and where you can use them. Keep in mind that the use of a test technique depends on various factors. For example, you need to have a suitable test basis for the technique. Note that the 'CRUD' test technique will be discussed in Chapter 4.

Test techniques	**Suitability for which processes and coverage**
Proces Flow Test	Suitable for processes where it is crucial to test the complete flow of events and actions, such as workflow applications and business processes, or complex code that you want to test within the flow. Can be well-applied in end-to-end, chain, or acceptance testing.
Semantic Test	Suitable for processes where the meaning and interpretation of data and messages are crucial, such as in communication protocols and data exchange systems. Often useful in system and integration testing.
Decision Table	Suitable for processes with complex business logic and decision rules, such as financial systems or insurance calculations. Can be applied at multiple test levels to test complex software. Combines well with Boundary Value Analysis or Equivalence Partitioning for higher coverage
Boundary Value Analysis	Suitable for processes where critical boundary values are, such as financial transactions, security applications, etc. The application of boundary value analysis generally provides average to high coverage.

Equivalence partitioning	Suitable for structured, well-defined processes where input variables can be divided into classes. Provides lower coverage than boundary value analysis.
Checklist Based Testing	Suitable for processes where structured lists of test items can be generated and applied, such as in user acceptance testing and security audits. Coverage depends on how specific the checklist is made.
Pairwise Testing	Suitable for processes with many combinations of input variables, such as configuration and compatibility testing in software applications. It can also be applied to generating test data. Provides average coverage, which is often representative and efficient.
CRUD matrix	Geschikt voor processen waarbij database-interacties en bewerkingen op gegevens cruciaal zijn, zoals in applicaties voor gegevensbeheer. Goed toepasbaar op authorisatiemodellen en levert hiervoor een hoge dekking op, mits volledig uitgevoerd.

In the following table, you see some examples of how to link these different test techniques to our previously established test strategy table. This allows us to deploy them to achieve 'High,' 'Medium,' and 'Low' coverage. Keep in mind that the deployment and effectiveness of a technique depend heavily on the specific context of the test project, the test basis, and the software to be tested. You should consider these factors when making a choice.

Test strategy table		**Test levels**			
Test goal/test type	**Static test**	**Unit test**	**Integration-test**	**System test**	**Acceptance-test**
Orders from France	Code review	Semantic test + Boundary Value Analysis	All posibilities and exceptions need to be tested	Decision Table + Boundary Value Analysis	Proces flow test of all E2E scenarios
Orders out off different parts of Europe		Pairwise testing	Pairwise testing	Semantic test	Proces flow test including most important scenarios
Orders from other parts of the world		Checklist with low coverage	Checklist with low coverage	Procesflow with low coverage	Procesflow test met enkele scenario's testen
Vendor API	Code review	CRUD		Decision tables + Boundary Value analysis	
Login	..	..	..	..	..
Search	..	..	..	..	..

Usability	Prototyping	Walkthrough		Eye tracking	User groups
API Documentation	Review				

In this practice, we have discussed several test techniques and explained how to apply them. We have also described the relationship between the test strategy and the deployment of test techniques. The practice "Document Your Tests" teaches you how to apply this in your daily work.

Definitions

Boolean	A result that can only have one of two possible values: true or false, yes or no.
Coverage	The extent to which your test cases cover the functionality to be tested (tested functionality vs. maximum existing functionality).
Detailed Test Case	A test case on a very detailed level of abstraction. Detailed test cases almost always consist of pre-conditions, an output prediction, and a result.
High-Level Test Case	A test case on a high level of abstraction that is usually meant to derive detailed test cases from (or to provide high-level insight into) the application. There is a relationship between test levels and the abstraction level of test cases. In higher test levels, you generally encounter less abstract test cases, meaning they are more high-level. For example, in unit testing, you typically have a very low abstraction level.
Logical Test Case	A test case that is derived from a test base consisting of preconditions (the inputs) and post-conditions (the actions or results). Logical test cases are based on logic, not concrete data.
Outcome Prediction	The expected result of a test case.
Physical Test Case	A logical test case with test data added to it.
Pre-condition	A condition that must be met to run a test case.
Result (Actual)	The actual result of a test case: ‘Ok’ or ‘Not OK.’
Test Process	The collection of interrelated activities consists of test planning, test monitoring and control, test analysis, test design, test implementation, test execution, and test completion.
Test Case	A set of preconditions, input values, actions (if any), expected results, and postconditions, developed from test conditions.
Test Technique	A structured way to derive test cases from a test base.
Test Script	A set of instructions for conducting a test.
Test Design	The activity that derives and specifies test cases from test conditions. **[V]**
Variable	An element, feature, or factor that is liable to vary or change.

Practice 8: Understand the Importance of Good Communication

LO34	Understand the importance of good communication in all your activities as a tester. (K3)

We now have learned a large number of skills for testing software in an effective and structured way. One skill serves to reinforce the previous practices: good communication.

The basic skills of communication include speaking, writing, listening, and reading. When messages are communicated via verbal or written channels, it is important to verify that the message has been delivered correctly and has been understood.

Messages always have a sender and a receiver. Good communication involves verifying, as the sender of the message, that the message has been delivered correctly and has been understood. As the receiver, verifying that your understanding of the communicated message was what was intended is also a good communication practice. In addition to verbal and written communication, there is also non-verbal communication, which can also be used to verify that messages have been received and understood.

Validating communication is necessary because everyone has personal filters that can cause messages to be processed in unexpected ways. Everyones' filters are based on their personal experience, background, cultural differences, upbringing, and beliefs. As a result, a message intended one way may come across in different ways to different receivers. When testing, it is important to verify that the messages you send have been understood as intended and that the understanding of messages you receive was the same as intended by the sender.

Communication can sometimes be difficult for several reasons: checking for understanding can make people feel the sender is controlling them, is too meticulous, or is interfering too much with their work. Also, people can come across as very confident, making you reluctant to ask further questions yourself. Therefore, good communication also requires thoughtfulness, tact, and soft skills. This practice is especially important because software development is a complex set of activities that are often not tangible.

Let's look at some examples of how you could apply this practice in conjunction with testing practices. The "no assumptions" practice also assumes a similar concept because, with this practice, you verify and/or validate everything and assume nothing.

The practice "test as early as possible" highlights the importance of reviewing. Reviewing or having products reviewed is an excellent way to increase the quality of these products at an early stage, and reviewing is all about verifying that the product is accurate, of good quality, and complete. "Products" could mean anything from functional design, requirement, test script, the result of risk analysis, etc. When sending products for review, good communication is important because confirming if the recipient understands what the product entails as well as what is expected of them can greatly improve the results received in the end. In doing so, the entire review will be more valuable.

The practice of "documenting your tests" is also relevant to the practice of good communication. By documenting your work and describing requirements or test cases clearly, not only can the documentation be referenced at a later time, but there is also a higher chance of you and others understand what is meant.

Another part of good communication is the management of expectations. It is important in our work as testers that reasonable and realistic expectations are set and communicated to the various stakeholders clearly and understandably. Managing expectations is important because it minimizes surprises and prevents expectations and final results from being too far apart. It is therefore important to communicate at the right time if, for example, you foresee that the creation of a test script or the execution of a test will take much longer than initially expected.

Using effective communication in a testing process is crucial.

Throughout a testing process, there are many instances where communication plays a vital role. We've learned in earlier practices that applying communication is important, including in the practice of 'Be Specific' and 'Document Your Tests.' In the practice 'Testing Everything is Impossible,' we learned how to establish a Risk-Based Test Strategy. Communication is inherent in all these practices. Of course, after executing the tests, we need to communicate with the client about the achieved test results. It would be peculiar to deliver the software at the end of

the process without providing stakeholders with insight into the results obtained during the testing of the software. Even during a lengthy development and testing process, keeping stakeholders informed is essential.

This can be achieved through simple reporting. In the report, we essentially need to address three topics: the status of test cases, defects, and risks. For test cases, you can include the statuses described in the table 'Example Test Case Statuses' below in the report.

Testcases	
Status	**Description**
Not Started	Test cases that have not been initiated yet.
In Progress	Test cases that have been initiated but are still in progress.
Not Applicable	Test cases that have been designed but are no longer applicable.
Failed	Test cases that were not successful and resulted in a defect.
Passed	Test cases that have been successfully completed.
Total	The overall count of test cases.

'Example testcase status'

For defects, you could consider the following statuses in your report. Note that sometimes more statuses may be possible than described in the table, but for a simple testing process, you can use the ones below.

Defects	
Status	**Description**
New	Defects that have just been discovered
Open	Defects that are still open
Assigned	Defects that have been assigned to someone for investigation, determining the impact
In Progress	Defects that are currently being repaired
Ready for Retest	Defects that are ready to be retested
Retesting	Defects currently undergoing retesting
Retest OK	Defects that have been resolved after retesting
Retest Not OK	Defects that have not been resolved after retesting
Parked	Defects for which the decision is made not to address immediately. For example, due to high impact on the current sprint or needing more analysis for a proper solution.

'Example defect status'

For defects, it is advisable to include the severity in your report. Below, you'll find an example of different severity levels that you could use.

Severity of Defects	
Severity	**Description**

Blocking	The defect blocks other functionality, preventing them from being tested, creating uncertainty about the functioning of these test cases.
Serious	The defect is severe and needs to be fixed but does not block any other functionality
Not serious	The defect is valid but not severe enough. There is an acceptable workaround available.
Cosmetic	The defect is cosmetic or a typo

'Example Severity of Defects'

Reporting on risks can be done by demonstrating how much effort has been invested in adequately covering the test goals identified in the risk analysis. For this, we take the table created in Practice 3 and make some adjustments. Essentially, you are reporting on your established test strategy, providing stakeholders with feedback on how and to what extent the identified risks during the risk analysis have been mitigated.

Risk report				
Risk(Testgoal)	**Consequence**	**Severity**	**Measures**	**Remarks**
Orders from France are not being processed	Significant revenue loss due to the inability to sell these products. Substantial damage to the company's reputation among customers.	High	Extensive testing	All measures as described in the testing process have been implemented. As a result, this risk has been completely mitigated.
Orders from other parts of Europe.	Gemiddels omezetverlies, Imagoschade.	Medium		This risk has been significantly reduced through the implemented quality measu
Orders from other parts of the world.	Minor revenue loss, slight damage to reputation.	Low		This risk has been nearly eliminated due to the implemented measures.
Vendors API	Sellers may not be able to present their products correctly, leading to potential revenue loss or errors in the offered products.	High	Test trajectory with sellers, monitoring availability after production release	Despite an extensive testing process, there remains a residual risk here, primarily related to the correct use of the API. To facilitate this as effectively as possible for the sellers, we want to establish monitoring and active support. This involves setting up our technical API service desk to proactively call back and assist if we detect a significant number of errors during monitoring.
…	..			

'Example risk report'

Realize that your reporting can also be approached per test level or test type. In our example, we use the entire test process. However, you could also report only on code review, unit testing, or acceptance testing. Some examples are provided below:

Code Review		
Lines of code reviewed	Defects found	Defects still open
16.268	68	5

'Example code review report'

Test Cases Unit test					
Not started	**In Progress**	**Not applicable**	**Failed**	**Passed**	**Total**
15	26	8	5	145	199

'Example Unit test report'

Defect report									
	New	**Open**	**Assigned**	**Solving**	**Ready for retest**	**Retesting**	**Retest OK**	**Retest not OK**	**Parked**
Blocking	0	1	1	0	2	1	5	0	0
Serious	0	1	2	0	0	1	5	1	1
Not serious	0	0	1	2	0	0	4	1	3
Cosmetic	1	0	0	1			2	0	1
Total	0	2	4	3	2	2	16	2	5

'Example Defect report'

The reports are meant as examples. You can use them or create your own report by combining elements from these examples. You can, of course, visualize this report by using charts or by reporting on the progress of the testprocess and status changes per day or week. By reporting at the right intervals (daily, weekly, monthly) on the progress of your test process and how it mitigates risks, you can effectively adhere to the practice of 'The importance of good communication' and provide stakeholders with insight into your testing process.

Definitions

Expectation Management	The process of clarifying or adjusting certain expected outcomes.
Non-verbal communication	The body language we use when we express ourselves.
Receiver	The recipient of a message or record, whether written or oral.
Sender	The sender of a message or record, whether written or oral.
Verbal communication	The words we choose to express ourselves.

Chapter 4: Quality Attributes, focusing on Security, Usability, and Performance

In this chapter, we further discuss some important examples of quality attributes. To gain a better understanding of what quality attributes are, we will focus on three that are relevant to nearly every project: security, usability, and performance.
These quality attributes are important to include when creating a test strategy. They are aspects of quality that you encounter in almost every software project and should be considered when formulating a test strategy. In our chosen example of an e-commerce website selling products, security, usability, and performance will also play a role. For instance, the risk associated with inadequate security measures could lead to unauthorized sellers wrongly using the API. Concerning the usability and performance of the webshop, if they are not in order, we may receive fewer user orders because they may abandon the platform due to a poor interface or slow processing speed. There are more quality attributes beyond those mentioned here. However, we choose to exclude them from this syllabus, except for functionality and the quality attributes mentioned in Chapter 1, to keep the content manageable and focus on the basics.

LO35	Learn the basics of testing for security. (K2)
LO36	Learn the basics of usability testing. (K1)
LO37	Learn the basics of performance testing. (K1)

Quality Attribute: Security

A very helpful aid when testing security in applications is The *OWASP Top Ten* **[XVIII]**. The OWASP (Open Web Application Security Project) is a non-profit foundation focused on improving software security. Every few years, the OWASP releases a list of the ten most critical security risks, called “The OWASP Top Ten”. The OWASP Top Ten is widely recognized as the first step for any company interested in improving the level of security in the software that they develop, as well as the methods by which those creating the software conduct their work. The focus is not only on the security software being created but also on the business processes and best practices in the workplace, for example, it would not be a good practice to have passwords written down near the working space. The OWASP Top Ten includes many valuable suggestions for how to include various best practices in testing (or development) processes. More specific information, including examples and detailed information on how to test for security flaws, is to be found at https://www.owasp.org.

Granting permissions is easy and enables users to do many things in a development and test trajectory. Often this is done from a “user is king” service point of view. To avoid broken access control, users should be granted permission to do what is necessary to fulfill their tasks.

Quality Attribute Security: OWASP Top Ten example: Broken Access Control

Broken access control usually means vulnerabilities in roles and permissions. Well-designed access control should be maintained and created so that users cannot act outside their

intended authorization. Failure to design access control appropriately could lead to unauthorized disclosure, updating, or deletion of data, or the use of functions that are outside users' intended authorizations. For example, take a small company in which users in the financial administration department are granted all permissions to financial administration. These permissions could lead to their being able to raise their salaries. This is why it is important to reiterate that the roles and permissions system must be properly designed, aligned, and tested. A good rule of thumb here is to grant only the permissions that are needed—no more.

The administrator role should only be used when necessary, and should not be shared with an infinite number of people. A specific account should be created to accommodate other (possibly admin) authorizations, and also set up a procedure that makes sure permissions can be revoked if necessary, e.g., when a user leaves the company.

Quality Attribute Security: CRUD Matrix

A good way to test (and also design and align) roles and permissions is to set up a simple authorization matrix. This authorization matrix is also called a CRUD (Create, Read, Update, Delete) matrix. An authorization matrix is created by placing all roles and permissions, linked to the functionality they are supposed to exercise, within a table.

	Customer Data	**Order Data**	**Product Data**
Manager	CRUD	CRUD	CRUD
Employee	CRU	CRU	RU
Customer	RU	CR	

Example simple CRUD matrix

By using a CRUD matrix, you can find missing steps or check whether data is (allowed to be) accessible. Working through the CRUD matrix and conducting tests with all different roles (one by one) we can verify all granted permissions as well as the actions that are not allowed, and then confirm again that what is not allowed is indeed not possible. If we take our example CRUD matrix, and follow the "Employee" row, for entities "Customer Data" and "Order Data" we see that we need to test for the creation, reading, and updating of data, we must not forget to confirm if the employee is unable to delete data in these entities.

Quality Attribute Security: OWASP Top Ten example: Cryptographic Failures

Cryptographic failuress focus on errors that are the result of encryption. This may be related to the process of encryption itself or the lack of necessary encryption. These failures can lead to the exposure of sensitive data in REST or transition, for example, a password database that uses a simple, cracked, or one-way encryption to store passwords.

A good starting point in resolving these issues is proper data classification: this determines the degree of confidentiality of the data. This is why passwords, medical records, credit card numbers, personal information, and confidential business information require extra protection,

specifically if that data is covered by privacy laws like the EU's General Data Protection Regulation (GDPR) or other standards.

After classification, the next step is to properly secure the data with modern, uncompromised encryption methods. These actions apply to both internal and external application links. See the OWASP website for more prevention suggestions and testing tips.

"Broken Access Control" and "Cryptographic Failures" are only two of the items from The OWASP Top Ten for making your software and working environment secure. It is nearly impossible to exclude the possibility of being hacked, but the goal should be to avoid it as much as possible. The challenge is to detect hacking quickly and ensure that you maintain access to (and control of) your data. Proper monitoring, backup procedures, fallback procedures, and restore procedures are key to maintaining secure data. If these procedures are in place, practiced, and tested regularly, you are already on a good path to better security practices.

The OWASP website mentions several more tips and tricks for secure development and security testing: https://www.owasp.org.

Quality Attribute: Usability

Usability refers to the degree to which a product or system can be used by specified users to effectively, efficiently, and successfully achieve specified goals in a specified context of use.

If we consider that we are checking a website, app, or on-premise system, the main screen is often the most important page when it comes to usability. Not only is it the most visited part of the software, it also determines whether the user can (or even will) continue to use the software to access all other parts. This is where users will form their first opinion of the software, so recognition of suitability is extremely important. Once the users' needs are met, the user will try to accomplish their goal(s) by using the system. To maximize user satisfaction, clear inputs should be provided and the most important tasks should be displayed. The main screen is also an ideal place to distinguish yourself from the competition.

For best results, clear navigation and information architecture are essential. The key is to focus on the task itself. You can orient yourself to the task if you know the need of a specific user. For example, if you are granted access to something, navigation in that space must be clear. Users want to know where they are, what they can do, and where they can go. You also want this information delivered in the most recognizable way. Elements for a website might include the "home" link, a hamburger menu, logical categorization (bread crumbs), a sitemap, etc.

It is important to match the designer's mental model with the user's mental model. To take this into account, consider what the user's main focus is: this should be the most important CTA (Call To Action). A CTA is the most important button in achieving a goal. This goal is usually what the visitor aims for, and this same goal usually benefits the business because it will contribute

to achieving their business goals. Therefore, be sure to clearly state what the most important tasks are.

In a form, it should be clear where to fill something out and what needs to be filled out. Immediate feedback for both correct and incorrect entries should be provided. Consider how you would display whether something is mandatory or in what format a phone number or postal code must be entered. If something goes wrong, feedback should be given directly to the user for improvement.
When considering the quality attribute usability, the main focus should always be on the goals of the user and the overall goals of the client itself. In the best case, these goals would overlap and the software should function as smoothly as possible for these goals to be met.

Quality Attribute: Performance

Performance testing can be divided into 2 different main topics: load testing and stress testing. Load testing involves simulating a certain load of users and looking at response times under this load. Stress testing involves finding the limit of what the system can handle in terms of peak load.

A good performance test requires specific knowledge and tools: detailed knowledge of the infrastructure and how it relates on the right scale to the infrastructure of the production environment. For a good performance test, it is crucial to have an environment that is similar or almost identical to the production environment. Infrastructural components such as switches, hardware, and firewalls and their sizing need to be considered to accurately simulate the system load. To achieve this load, often specific tools are required to simulate a large number of simultaneous users.

Another way to get a good indication of performance is by making a user load profile of the future environment. This can provide a better understanding of what load is required of the system, and at which times of the day. In addition, setting up proper monitoring, including timestamps, is a mitigating measure in performance testing.

Definitions

Call to Action	The most important action you want the visitor to perform
Load Testing	A type of performance testing performed to evaluate the behavior of a component or system under different loads, usually between expected conditions of low, normal and peak usage.
Main Screen	The most important screen of an application from which most actions can be done, usually the first screen after logging into an application.
Perfection User Interaction	The degree to which a user interface allows the user to have an enjoyable and satisfying interaction.
Performance	The speed at which the information system handles transactions.
Recognition of Suitability	The degree to which users can recognize whether a product or

	system is appropriate for their needs.
Security	Assurance that consultation or mutation of the data is possible only by those authorized to do so.
Stress Testing	A form of performance testing that aims to evaluate a component or system at or beyond the limits of the workload expected or specified for it, or with limited availability of resources such as memory or server capacity.
Usability	The degree to which a product or system can be used effectively, efficiently and satisfactorily by users.

References

Reference	Source
[I]	**Book:** L. Anderson, P. W. Airasian, and D. R. Krathwohl, A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives, 2001, Allyn & Bacon, ISBN 03-21084-05-5
[II]	**Website:** ISO 9000:2015 Standard information: https://www.iso.org/standard/45481.html
[III]	**Website:** American Society for Quality glossary: https://asq.org/quality-resources/quality-glossary/
[IV]	**Website:** ISO 25010 Standard information: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010
[V]	**Website:** ISTQB® glossary application: https://glossary.istqb.org/
[VI]	**Website:** Wikipedia page for Vilfredo Pareto: https://en.wikipedia.org/wiki/Vilfredo_Pareto
[VII]	**Article:** Doran, G. T. (1981). "There's a S.M.A.R.T. Way to Write Management's Goals and Objectives", Management Review, Vol. 70, Issue 11, pp. 35-36.
[VIII]	**Article:** Jan Vanthienen, "The History of Modeling Decisions using Tables (Part 1)" Business Rules Journal, Vol. 13, No. 2, (Feb. 2012): https://www.brcommunity.com/articles.php?id=b637
[IX]	**Book:** Boehm, B. (1981) Software Engineering Economics, Prentice-Hall Inc., ISBN 01-38221-22-7
[X]	**Website:** Agile Alliance glossary: https://www.agilealliance.org/glossary/user-story-template/
[XI]	**Website:** Wikipedia page for Flowchart: https://en.wikipedia.org/wiki/Flowchart
[XII]	**Book:** Pol. M., Teunissen. R., Veenendaal van. E., Testen volgens TMap 2e druk, p326, ISBN 90-72194-58-6
[XIII]	**Book:** Beizer, B. (1990) Software Testing Techniques. 2nd Edition, Van Nostrand Reinhold, New York. ISBN 18-50328-80-3
[XIV]	**Book:** Burnstein, Ilene (2003), Practical Software Testing, Springer-Verlag, ISBN 0-387-95131-8

[XV] **Book:** G. Myers, The Art of Software Testing; John Wiley, New York, 1979. ISBN 0-471-04328-1

[XVI] **Website:** ISTQB Foundation syllabus, 2019 version, p61: https://www.istqb.org/certifications/certified-tester-foundation-level

[XVII] **Website:** Software Testing Help regarding for Pairwise Testing: https://www.softwaretestinghelp.com/what-is-pairwise-testing/

[XVIII] **Website:** OWASP Top10: https://owasp.org/Top10/

[XIX] **Article:** The New Religion of Risk Management, Bernstein, 1996: https://hbr.org/1996/03/the-new-religion-of-risk-management

[XX] **Website:** ISO29119-4:https://www.iso.org/obp/ui/en/#iso:std:iso-iec-ieee:29119:-4:ed-2:v1:en

[XXI] **Website:** TMap Glossary Online: https://www.tmap.net/page/tmap-glossary-online