Certified Automation Practitioner in Agile Testing (CAPT) Syllabus

Version 1.0

Released 15th of August, 2024

Copyright Notice

Thank you to the primary authors

- Patty de Vries van Riemsdijk, Randall Wessel

Thank you to the co-authors and editors

- Bas Kruip, Henri Budde, Bram van den Berg, Dineke van Vliet, Siegmond Waterfort, Kyle Siemens, Shanna Nguyen, and Khadidja Zegrir

Thank you to the review committee

- A special thanks to Wim Decoutere for his detailed comments and suggestions.

Revision History

Version	Date	Remarks
0.01	10-06-2022	Initial version
0.1	08-04-2023	Version for first review
0.2	13-04-2023	Minor adjustments spelling and grammar
0.3	18-04-2023	Hands-on objectives grouped and relabeled
0.4	17-05-2023	Ready for the Review Committee
0.5	13-09-2023	Subset of review comments applied
0.9	01-01-2024	Adjustments based on comments review committee
1.0	15-08-2024	First public release

Table of Contents

Business Objectives

Business objectives (BOs) are a brief statement of what you are expected to have learned after training based on this syllabus.

BO-1	Understand the value of test automation
BO-2	Understand the concepts of The Six Thinking Hats of de Bono for approaching a subject from different perspectives
BO-3	Understand and be able to apply the concepts of Toulmin's reasoning diagram for making conclusions based on good reasoning
BO-4	Understand the use of Testing Quadrants in differentiating tests
BO-5	Understand the concept of the Test Pyramid
BO-6	To be able to differentiate between test frameworks
BO-7	Understand the concept of BDD and TDD
BO-8	To be able to plan a test automation approach and strategy
BO-9	To be able to select a tool evaluating multiple aspects
BO-10	To be able to decide on what will be tested using automated tests
BO-11	Understand the best practices for creating automated tests
BO-12	Understand the elements of embedding test automation in an organization

This syllabus focuses on test automation concepts. Its general approach to test automation should enable your organization to generate value and guide you in setting up strong test automation that is maintainable and reusable in an agile context.

Different test automation framework models are explained to determine which types of automated testing can be used for a project or organization's product. Many tips are shared on getting started with test automation, choosing test tools, deciding what to automate, and maintaining test automation with specific strategies.

This syllabus does not require prior knowledge, but some basic knowledge of test automation is favorable. Therefore, it is highly suitable for beginners, interested parties, and knowledge holders who want to improve their test automation.

Learning Objectives / Cognitive Levels of Knowledge

Learning objectives (LOs) are brief statements that describe what you are expected to have learned after studying each chapter. The LOs are defined based on Bloom's modified taxonomy as follows:

Definitions	K1 Remembering	K2 Understanding	K3 Applying
Bloom's definition	Exhibit memory of previously learned material by recalling facts, terms, basic concepts, and answers.	Demonstrate understanding of facts and ideas by organizing, comparing, translating, interpreting, describing, and stating main ideas.	Solve problems in new situations by applying acquired knowledge, facts, techniques, and rules differently.
Verbs (examples)	Remember Recall Choose Define Find Match Relate Select	Summary Generalize Classify Understand Compare Contrast Demonstrate Interpret Rephrase	Implement Execute Use Apply Plan Select

For more details on Bloom's taxonomy, please refer to [BT1] and [BT2].

Hands-on Objectives

Hands-on Objectives (HOs) are brief statements that describe what you are expected to demonstrate or execute to understand the practical aspect of learning. The HOs are defined as follows:

- HO-0: Live view of an exercise or recorded video.
- HO-1: Guided exercise. The trainees follow the sequence of steps performed by the trainer.
- HO-2: Exercise with hints. Exercise to be solved by the trainee, utilizing hints provided by the trainer.
- HO-3: Unguided exercises without hints.

Level	Objective	Chapter/ Paragraph
HO-2	Describe the value of test automation for a (demo) SUT	1.2 - 1.3
HO-2	Applying the six De Bono Thinking Hats	1.4
HO-2	Applying Toulmin's reasoning diagram in distinguishing facts and opinions	1.5
HO-2	Applying the test pyramid on what to test automate in an Agile environment	2.2
HO-2	Apply the knowledge of the frameworks in selecting a framework	3
HO-0	Demo of linear framework	3.1
HO-0	Demo of keyword-driven and/or hybrid framework	3.4 / 3.7
HO-2	Write a (high-level) test suite	3
HO-2	Select the tool that fits the needs for a user interface test	5.3
HO-2	Select the right tool for a specific performance test	5.4
HO-2	Use models and techniques when creating a test approach	6
HO-2	Define as a team an automation strategy addressing ownership and the SDLC	6.4
HO-2	Select (an) appropriate test automation tool(s) to test the SUT	7.2
HO-2	Do a (mini) spike on selecting a tool appropriate for the SUT	7.3
HO-2	Follow the steps Collect and investigate, Value, Order, Analyze and Decide	8
HO-2	Learn how principles and patterns help reduce costs	9.1
HO-2	Apply the concept of DRY to test automation and SUT	9.2
HO-2	Apply DSL on test automation and SUT	9.3
HO-2	Use storytelling to present results and success	10.5

Prerequisites

Mandatory

- No specific prior knowledge is required for this course.

Recommended

- Some knowledge of Agile or Scrum or reading the Scrum guide.
- Basic knowledge of testing in general.

Reading instructions

This syllabus should be approached in the following manner:

- Please be advised that this syllabus describes the business outcomes and learning objectives for AU - Certified Automation Practitioner in Agile Testing (CAPT). Due to its conceptual nature, the practical knowledge and experience gained throughout AU official training, in addition to studying this syllabus, will greatly support the preparation for the AU-CAPT certification exam.
- This syllabus covers test automation topics in the order the authors consider the best way to set up test automation. It is recommended to read this syllabus in sequential order.

1 Expectations and Value of Test Automation

Testing is an essential activity in the software development process. Testing reveals defects that can be solved and provides insight into the quality and risks of a product. Automated checking of new or changed code provides the insight that it (still) functions as expected. When tests become (too) large, labor-intensive, or complex it will most likely take too much time or be too error-prone to test manually. Test automation allows us to tackle these issues. Automation will free up the hands of testers to do things that are better done by human beings, like critical thinking, exploring, etc.

Keywords

Test automation, automated checks, automation-supported testing, regression testing, business value, return on investment, Toulmin's reasoning diagram, value, discussion, brainstorming, The Six Thinking Hats of De Bono

LO-1.1	K2	Explain the difference between test automation and automation-supported testing
LO-1.2	K2	Explain the ROI when regression tests are executed frequently
LO-1.3	K2	Explain the value of test automation
LO-1.4	K2	Understand the value of test automation within an organization
LO-1.5	K1	Recall the advantages of test automation
LO-1.6	K1	Recall the disadvantage of manual testing
LO-1.7	K2	Explain the technique of the six Thinking Hats of De Bono
LO-1.8	K2	Explain the concept of the Toulmin's reasoning diagram

1.1 Understanding the Basics of Test Automation

LO-1.1	K2	Explain the difference between test automation and automation-supported testing

Test automation is a way of testing software in which test tools are set up to perform automated tests. During an automated test, the tool compares the test results with the expected test results. This all happens automatically with little or no manual help. An **automated test** is a check with a binary result, either the result is ‘OK’ or ‘not OK’. These automated tests are considered to be **checks**. So, test automation can also be considered automated checking [TC1].

Automation-supported tests do not perform checks that result in binary results but measure things like response times or mean times between failures, or they crawl for problems like broken links or security holes.

This differentiation is somewhat arbitrary. When a performance test tool compares a result with a target or threshold value, it may also output an ‘OK’ (green) or ‘not OK’ (red).

Test automation is of great use for tests that have to be repeated regularly, like regression tests.

By automating regression tests, testers have more time to focus on testing the software changes, so the testers receive regular feedback on whether the existing parts of the software are not broken due to these changes.

Another use of automation is in processes like the creation and cleanup of setting up and tearing down environments or creating and cleaning test data.

1.2 Value of Test Automation

LO-1.2	K2	Explain the ROI when regression tests are executed frequently
LO-1.3	K2	Explain the value of test automation
LO-1.4	K2	Understand the value of test automation within an organization
LO-1.5	K1	Recall the advantages of test automation
LO-1.6	K1	Recall the problems with manual testing

Test automation is almost indispensable in software development. Test automation contributes to discovering any defect at the earliest possible stage in software development. An automated test can be performed several times a day without human intervention. Tests can be performed faster with automation and therefore contribute to a shorter turnaround time and lower costs than manual testing.

Manual testing will reveal defects, which is of great value. However, it takes a lot of time to manually test the same test scenarios repeatedly. Automated testing can offer a solution and perform the same steps as manual testing. Depending on how test automation is set up, automated tests can be much faster.

Test automation requires thorough planning to avoid problems. Starting with test automation without thinking about the why, how, and what may lead to issues. For example, a chosen test automation tool may not fully comply with the test scenarios that must be automated. Thus, high-quality test coverage cannot be achieved, nor will it support the tests that have to be performed manually; therefore, essential tests and associated risks might be overlooked. The same problem can arise when one does not consider which test scenarios should be automated.

Although it takes some time to write the scenarios into scripts in test automation, once completed, scripts can be run repeatedly without much additional cost. The scripts and test automation will be maintained, but the plus is that it will save time in the long run. In addition, if the same steps are repeated automatically, the chance of human error in basic and/or regression testing is reduced, giving the manual tester time to create more depth in other tests. As a result, test automation can have a high return on investment (ROI). It is also vital to note that it should be determined whether the estimated ROI can be achieved. If scenarios need to be updated quite often, the ROI might never be reached. Another risk is that test automation itself is also software and can, therefore, contain errors.

Test automation can take over the routine testing activities of testers so that these testers can focus more on the more complex test topics. It can also contribute to the faster delivery of a product when, for example, a CI/CD pipeline is used within a DevOps team. The automated tests give confidence that the tested application and the associated system (still) work as expected.

By carefully considering which manual tests can be replaced by test automation, the testing will become more effective and efficient.

Benefits of well-implemented test automation are [AT1]:

- Test executions can become more efficient
- Tests can be executed more accurately
- The coverage ratio of the object to be tested can be increased
- It is possible to test different test scenarios in parallel or in different environments
- Test automation can help to discover defects faster

- It can save time
- It can save money

More about regression, performance, and other test types in Chapter 3.

1.3 The Six "De Bono Thinking Hats"

LO-1.7	K2	Explain the technique of the six thinking hats of de Bono

But how do you determine the value of test automation? Have you thought about the advantages and disadvantages of test automation?

An exercise that can help to bring forward an interesting discussion is the use of The six "De Bono Thinking Hats" [TH1]. The six De Bono Thinking Hats helps to look at something from different perspectives and collect different arguments. The Six Thinking Hats of de Bono is not about classifying someone in a group, but about creating a shared understanding. There are six different roles/perspectives with differently colored hats that represent a different way of thinking:

- Blue hat = the organizer determines the agenda, decides who has the floor, monitors the conversation.
- White hat = the information expert always wonders whether we have the right information and what the source is.
- Red hat = the emotional: wonders what we think, can indicate doubts like "I doubt if …." but is also intuitive "I have the feeling that….".
- Yellow hat = the positive: what are the benefits and feasibility? Even for a real problem, always sees the benefits.
- Black hat = the critical: what are the risks, what are the (potential) problems? Always consider obstacles on the road.
- Green hat = the creative: always has alternatives and ideas available.

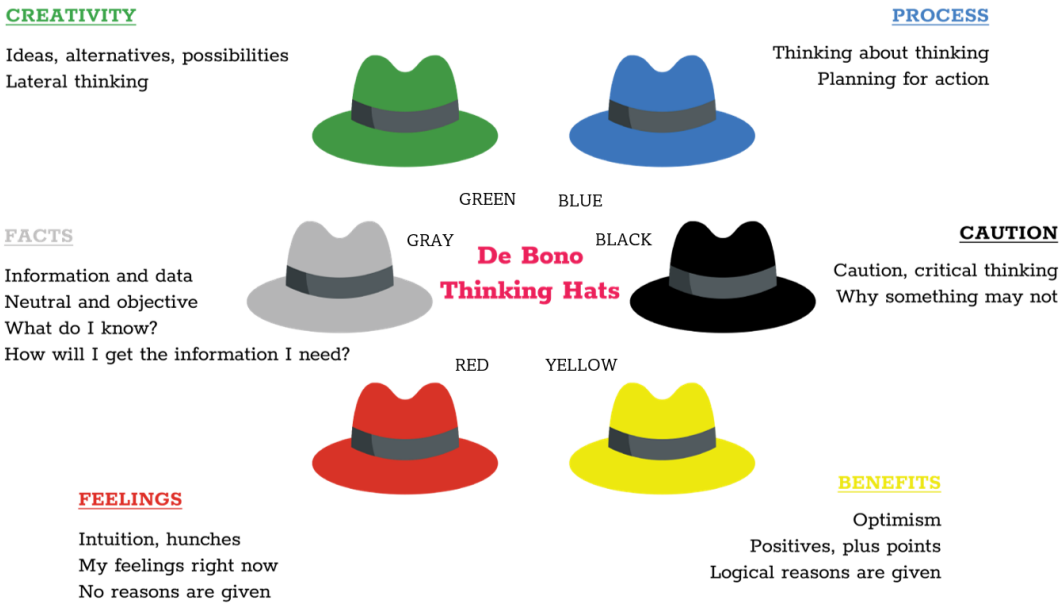

Figure 1.1: De Bono Thinking Hats

The method is simple. A team consists of six people, each wearing one of the thinking hats. From that point on, you are only allowed to think and argue from the perspective linked to the color of the hat you are wearing. Typically, the blue hat will facilitate the discussion. These hats represent acting roles and should not be confused with one's personal character.

Each participant can put on a hat, or you can all think from one hat. On a canvas with the six colored thinking hats, you can indicate how you view the position per color.

Testers usually prefer the black hat. This technique forces them to consider things from a different perspective.

1.4 Toulmin's Reasoning Diagram

LO-1.8	K2	Explain the concept of the Toulmin's reasoning diagram

When talking about the use and need of test automation a lot of claims, arguments and assumptions can be collected. Toulmin's reasoning scheme is a useful means for analyzing arguments [TA1]. An argument is the entire construction of arguments, assumptions, and any evidence that ultimately leads to a particular position or conclusion. It concerns the question based on whether a specific statement should be accepted or not accepted.

The steps of Toulmin's reasoning diagram with examples and pros and cons to the examples:

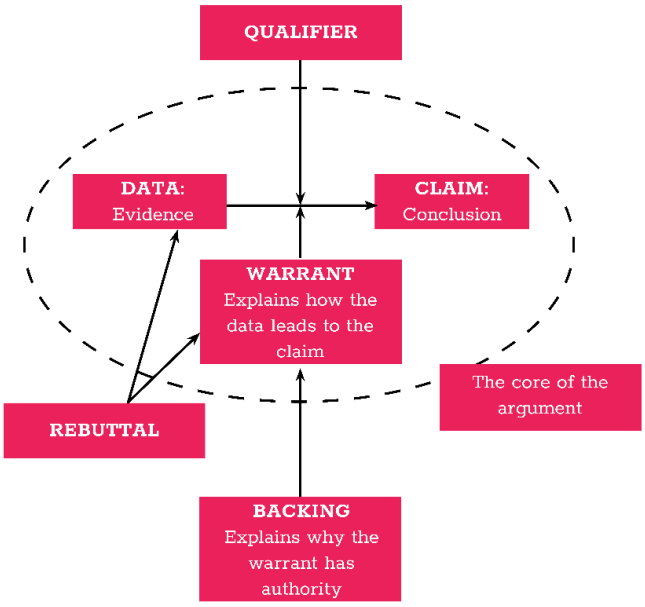

Figure 2.2: Toulmin's Reasoning Diagram

1. Identify the Claim: The claim is the main point or statement that the argument is making.

Example: "Regular exercise improves overall health."

Pro: Regular exercise has been scientifically linked to various health benefits, including improved cardiovascular health, weight management, and mental well-being.

Con: The claim may not consider individual differences, and some people may have medical conditions or physical limitations that prevent them from exercising regularly.

2. Find the Data: The data is the evidence or information used to support the claim.

Example: "Studies have shown that people who engage in regular physical activity have lower risks of heart disease, obesity, and diabetes."

Pro: The data is based on scientific studies, which lends credibility to the argument.

Con: Some studies may have limitations or conflicting results, and not all studies account for other factors that could influence health outcomes.

3. Locate the Warrant: The warrant is the underlying assumption that links the data to the claim.

Example (implied warrant): "Engaging in physical activity has a positive impact on health, and what benefits one's health can be considered improving overall health."

Pro: The implied warrant connects the data to the claim in a logical way, suggesting that health improvements through exercise contribute to overall health.

Con: The warrant might not hold true in all cases, as some health improvements from exercise may not directly translate to overall health improvements.

4. Identify Backing (Optional): Backing provides further support or reasoning for the warrant.

Example (optional backing): "Medical experts and health organizations recommend at least 150 minutes of moderate exercise per week for adults, indicating its positive impact on health."

Pro: Expert recommendations add credibility and demonstrate a consensus within the medical community.

Con: Different health organizations may have varying guidelines, and relying solely on expert recommendations might overlook individual experiences and preferences.

5. Identify Qualifiers (Optional): Qualifiers introduce conditions or limitations to the claim.

Example (optional qualifier): "Regular exercise can improve overall health, but individual results may vary based on factors such as age, existing health conditions, and the type of exercise."

Pro: Qualifiers acknowledge the complexity of health outcomes and avoid making sweeping generalizations.

Con: Overemphasizing qualifiers might weaken the claim's impact or lead to the perception that the claim lacks certainty.

6. Recognize Rebuttals (Optional): Rebuttals acknowledge potential counterarguments or opposing viewpoints.

Example (optional rebuttal): "While regular exercise has numerous health benefits, it may not be the sole factor influencing overall health, as genetics and other lifestyle factors also play a role."

Pro: Addressing potential counterarguments demonstrates a willingness to engage with opposing viewpoints and strengthens the argument's overall persuasiveness.

Con: Overemphasizing rebuttals could undermine the confidence in the original claim or shift the focus away from the primary argument.

By discussing the pros and cons at each step, you can critically evaluate the strengths and weaknesses of the argument more comprehensively. It also allows for a balanced and nuanced examination of the topic.

There are also some additional points to consider when applying the Toulmin Model and discussing arguments:

Strong vs. Weak Warrants: Some arguments may have strong warrants establishing a clear connection between the data and the claim. Others may have weak warrants that leave room for doubt or interpretation. Assessing the strength of the warrant is crucial in determining the overall effectiveness of the argument.

Counterarguments and Rebuttals: Encourage participants to think critically about potential counterarguments that opponents might raise. Exploring possible rebuttals demonstrates a well-rounded analysis and prepares them to address opposing viewpoints effectively.

Evidence Quality: Not all data presented in an argument is equal. Discuss the importance of using reliable, credible, and up-to-date sources of information. Evaluating the quality of evidence enhances the argument's credibility.

Logical Fallacies: Help participants recognize common logical fallacies that may weaken an argument, such as ad hominem attacks, false analogies, or hasty generalizations. Understanding logical fallacies can improve the clarity and validity of their own arguments.

Audience Analysis: Encourage participants to consider their target audience when constructing and presenting arguments. Audiences may respond differently to certain types of evidence, warrants, or qualifiers.

Balance and Fairness: Emphasize the importance of presenting a balanced and fair argument that considers both the pros and cons. Acknowledging potential weaknesses or limitations shows intellectual honesty and increases the argument's persuasiveness.

Real-Life Examples: Utilize real-life examples or current events to demonstrate the application of the Toulmin Model in analyzing arguments. This can make the process more engaging and relevant for participants.

Practice and Feedback: Provide participants with opportunities to create their own arguments and receive feedback from peers or the teacher. Practicing the Toulmin Model with different topics and contexts helps reinforce their understanding.

Ethical Considerations: Discuss the ethical implications of making arguments, especially when it comes to controversial topics. Encourage participants to be mindful of the potential impact of their arguments on others.

Importance of Context: Remind participants that the effectiveness of an argument depends on the context in which it is presented. Cultural, social, and historical factors can influence how an argument is received and interpreted.

By incorporating these additional points into the discussion of the Toulmin Model, participants can develop a deeper understanding of argumentation, critical thinking, and effective communication.

The Toulmin Model is a valuable tool for breaking down complex ideas and constructing well-supported arguments in various academic and real-world contexts.

2 Test Types for Test Automation

Many different tests could be automated. Some examples are:

- Unit tests
- Integration tests
- Functional tests
- Security tests
- Etc...

It is challenging to decide which tests are best captured in test automation. When doing a search on the internet, there are many tests and even more tools to choose from. In this chapter, the Testing Quadrants and the Test Pyramid structure the tests and the tools used for those tests.

Keywords

Agile Testing Quadrants, Test Pyramid, Unit tests, Integration Tests, exploratory tests, A/B testing, end-to-end, business facing, technology facing, development, critique, support

LO-2.1	K2	Understand the difference between the agile testing quadrants
LO-2.2	K2	Understand the use of the Agile Testing Quadrants to decide on what to automate
LO-2.3	K2	Understand the concept of the test pyramid on what to automate
LO-2.4	K2	Understand the importance of unit tests
LO-2.5	K2	Understand the importance of integration tests

2.1 Agile Testing Quadrants

LO-2.1	K2	Understand the difference between the agile testing quadrants
LO-2.2	K2	Understand the use of the Agile Testing Quadrants to decide on what to automate

The Agile Testing Quadrants are used to classify tests. The quadrants have been helpful over the years for teams as they plan the types of tests they want to implement. It also helps teams in the early identification of required resources identify resources necessary. This model was developed by Brian Marick in 2003 [BM1].

There are four agile testing quadrants. The bottom quadrants (Q1 and Q4) contain tests facing technology. The top quadrants (Q2 and Q3) handle tests facing the business. On the left-hand side (Q2 and Q1) are tests that support the development team, allowing them to confirm if they are going in the right direction (solutions are appropriate, designs are correct, etc.). On the right-hand side (Q3 and Q4) are tests that critique the product to confirm if the test team is on the right path to fulfill the requirements and ensure that the product will ultimately be fit for purpose.

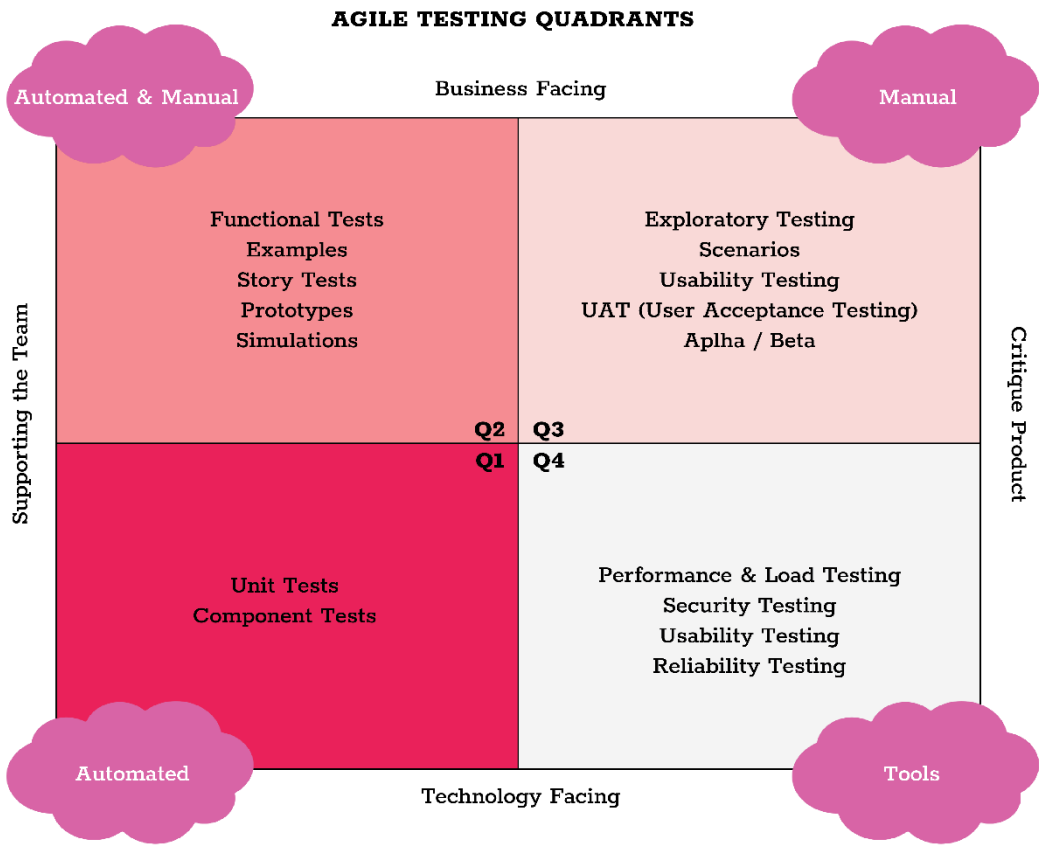

Figure 3.1: Agile Testing Quadrants

Quadrant 1: Technology-facing to Support the Team

Quadrant 1 contains tests that help the team perform technical tests. These tests are preferably automated, enabling the development team to ensure the software is (still) working as expected while also providing a solid foundation for the quality of the code written.

Unit and component tests are examples of tests that fall into Q1. All of these confirm that the code is working as intended. Tests in Q1 are written alongside development. It is advised to do as many tests as possible in this quadrant because these tests are written and executed at almost the same time as the actual software code, providing feedback in the shortest time possible.

Although not displayed in the image above, integration tests belong to Quadrant 1.

Quadrant 2: Business-facing to Support the Team

Quadrant 2 contains tests focusing mainly on the business itself while also supporting the development team. The tests can be automated or done manually, providing information about the application. The results of these tests support the validation of features within an application. Examples of tests that fall into Q2 are functional tests and user-interface tests. Manual testing in Q2 uses models like prototypes or mockups. Tests in Q2 are likely performed during and after development.

Many teams make the creation of automated tests an element of the definition of done. This ensures that testing, maintaining test automation, and quality assurance will not be forgotten. These

automation tests will also help uncover defects and issues quickly in the application before releasing the software to the public.

Quadrant 3: Business-facing to Critique the Product

Quadrant 3 contains business-facing tests that critique the product. It provides feedback about an application's current state and whether or not it works as expected. The tests in this quadrant are mainly manual, but some can also benefit from automation.

The tests here provide feedback about an application's current state and whether or not it works as expected. Usability testing, UAT, and Alpha/Beta testing are user-oriented tests that help to understand the user's experience. This quadrant involves critical thinking and in-depth observation of the applications' workflows.

As this quadrant focuses on manual testing, most tools in this quadrant support test execution. Some tools simply support registering and reporting on executed tests. Quite often, standard office word processors or basic text editors are used to create test scenarios and note results. Complex and expensive setups for eye and mouse tracking are available for usability tests.

Exploratory tests, usability tests, and Alpha/Beta tests are examples of tests that fall into Q3. These tests can be performed either before, during, or after development. The purpose is to collect information on what can be improved in the application, such as defects, but also less concrete feedback like remarks of the users or observations of the testers on the users.

Quadrant 4: Technology-facing to Critique the Product

This quadrant includes technology-facing tests that critique the product. These tests require automation and tools and provide specific information about an aspect of the application.

Examples of tests that belong to Q4 are:

- Performance tests
- Reliability tests
- Resilience tests
- Security tests

Security tests, performance tests, and related tests are discussed more in-depth in chapters 5.4 and 5.5.

Quadrant 4 tests are planned when there is a sense of urgency or a (legal) requirement that has to be met. If, for example, a fast page load time is essential, implementing some performance testing is probably beneficial.

There are no strict rules about what kind of test belongs in a specific quadrant or what tests are necessary for a software project. The testing quadrants only provide a way of ordering.

The fact that each quadrant has a number does not imply that there is some specific order, nor that tests should be implemented for each quadrant. There can be a focus on implementing a Q2 test first and then a Q1 test or vice versa. The goal is to understand that many test types are either automated or manual and to identify the most crucial types of tests that need to be implemented. The test quadrants can guide the team in determining what type of tests should be implemented. Using the quadrants during planning, development, and release will result in a better understanding of the importance of testing by the whole team.

2.2 The Test Pyramid

LO-2.3	K2	Understand the concept of the test pyramid on what to automate
LO-2.4	K2	Understand the importance of unit tests
LO-2.5	K2	Understand the importance of integration tests

The Test Pyramid is a model developed by Mike Cohn [MC1] that assists in determining a strategy for test automation. The pyramid consists of three levels: Unit (Test), Integration (Test), and End-to-end (Test). End-to-end tests are often called UI tests. This model emphasizes that you need a solid base of small unit tests. Those tests are cheap to create and fast to execute. The middle layer of integration tests requires more effort to create and execute. The top layer contains those that are relatively expensive to create and take longer to execute.

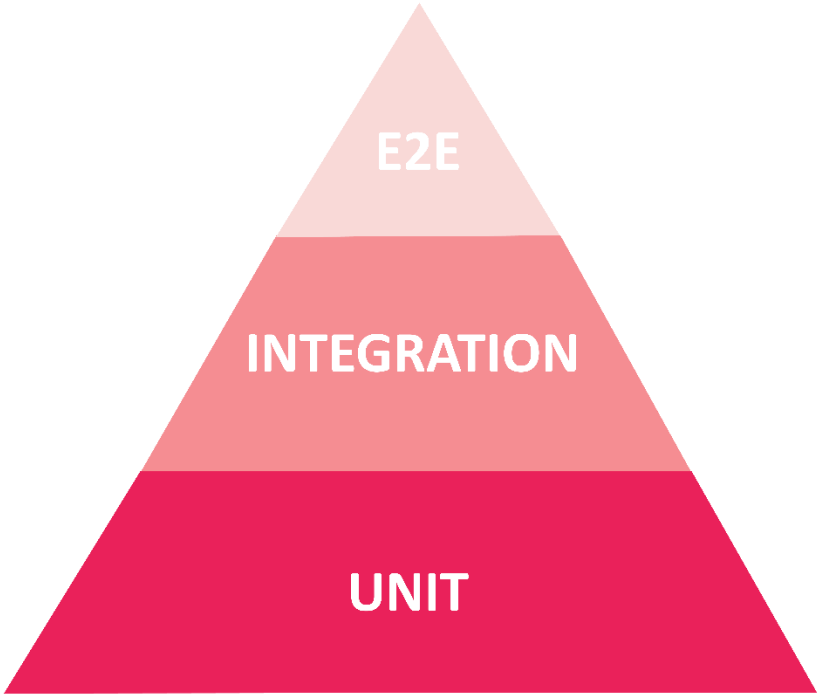

Figure 2.2 The Test Pyramid (Source: Google [GB1])

Bottom Layer: Unit Tests

The bottom layer is the unit test layer. Unit tests are considered the most important. The success of an application starts with unit tests. They help to ensure that the application's functionality works as it should. Unit tests are fast and test only one thing. Unit tests run for a few seconds and often several times a day. They provide quick insight into whether new or changed development code in software development gives the desired result. Unit tests are suitable for testing different aspects of a software component. Unit tests are closest to the development code. It is therefore recommended to cover sufficient development code within the test automation with unit tests because then it can be quickly demonstrated that adding new code or modifying existing code does not break any functionalities. The most significant number of tests is, therefore, located on this layer of the test pyramid.

Middle Layer: Integration Tests

The middle layer is the integration test layer. Integration testing focuses on the point where multiple parts of a software application come together and must integrate and function together. While unit tests focus on small parts, integration tests focus on the broad picture. Integration tests detect unexpected failures that happen when one part of the application is fixed and another is broken. Not all functions are tested through unit testing. Some parts can only be tested as part of a more extensive process. Integration tests cover critical cross-module processes.

For example, the collaboration between databases and APIs. These tests are tested against various authorities and software components, but not yet in conjunction with the user interface. Integration tests take longer than unit tests, but they don't take longer than user interface tests. Hence the middle layer of the test pyramid.

Top Layer: End-to-End Tests

The top layer is usually considered to be the user interface layer. User interface tests are in the top pyramid layer and focus on the software application's visual functional flow. These tests check whether a software application's user interface and front-end work as expected. UI tests simulate human behavior. UI tests are also called end-to-end tests or functional tests because they test the entire application from the frontend UI to the backend database systems. This full integration test on the user interface will ensure that machines, networks, and components work together as expected. It is important to remember that these UI tests are the hardest to set up.

These tests are the slowest compared to unit tests and integration tests. They are also vulnerable because single changes in a part of a software application could lead to failing tests. This requires a lot of maintenance of automated tests.

Regarding the test pyramid, creating tests for the bottom tier is relatively cheap, and the top layer will cost the most. The pyramid concept is to write many unit tests, which are relatively cheap and provide quick feedback. When a lot of functionality is tested by unit tests, less will need to be tested through the more expensive UI tests in the middle layer.

However, in many organizations, an inverted pyramid can be observed, where the focus is on user interface testing and not on unit testing. This anti-pattern of the testing pyramid can occur when unit testing is neglected, and Testing/QA compensates by creating more tests at the higher levels. GUI testing is appealing because testing from the user's perspective gives a greater sense of accomplishment as it covers real-world behavior and seems to leave fewer defects [GB1].

Another variant of the anti-pattern is the Hourglass, as described by Google [GB1]. The hourglass has only a few integration tests in the middle layer. This happens when end-to-end tests are used where integration tests should be used.

You can gain valuable insight into the functional flow of the user interface, but underlying defects within the code and possible integration are often discovered too late. A discovered flaw in the top layer of the pyramid will take much longer to be resolved. This not only blocks a release or go-live because of an extension of the lead time but also costs a lot more money. If no structured plan has been drawn up for automated testing, there is a risk that a software application will be tested incompletely.

In variants of the pyramid, the middle level is split into more levels, such as component, integration, and API tests, and the pyramid is topped with manual testing [JW1].

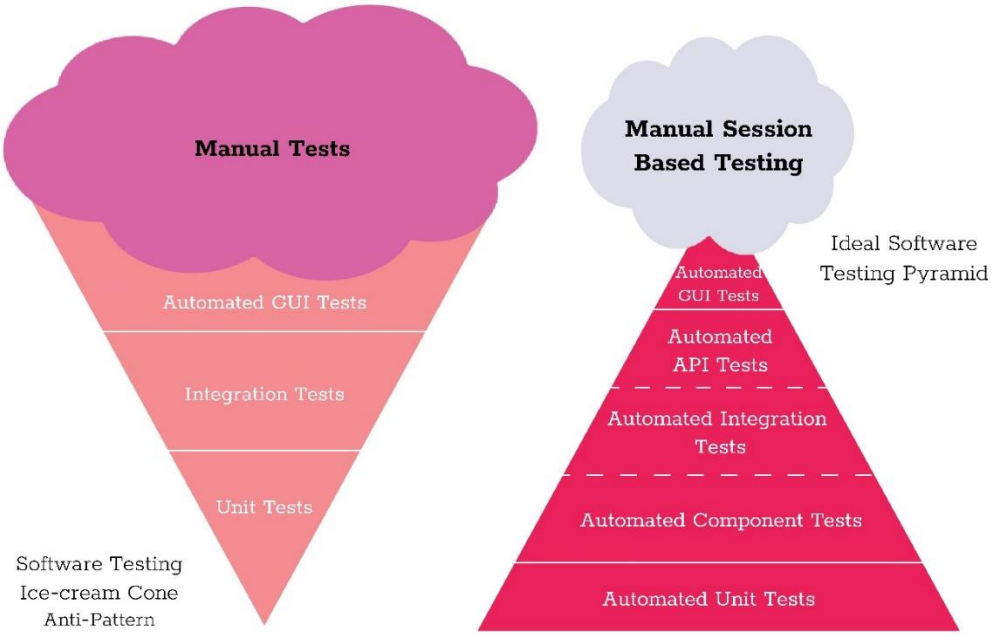

Figure 2.3 Variants of the Testing Pyramid (Source: Vishal Naik - Thoughtworks [AC1])

Google presented a simplified pyramid that has end-to-end tests as the top layer.

3 Test Automation Frameworks

Once the decision has been made on which tests to automate, the choice of test automation framework is essential. There are different test automation frameworks, each framework with its own characteristics. This chapter describes several frameworks, the relations between them, and their advantages and disadvantages.

Keywords

Linear framework, Modular framework, Data-driven framework, Keyword-driven framework, Hybrid framework, Model-based testing framework, Code-Driven testing framework

LO-3.1	K2	Compare and differentiate the test automation frameworks
LO-3.2	K1	Recall the characteristics of a linear framework
LO-3.3	K1	Recall the characteristics of a modular framework
LO-3.4	K1	Recall the characteristics of a data-driven framework
LO-3.5	K1	Recall the characteristics of a keyword-driven framework
LO-3.6	K1	Recall the characteristics of a model-based testing framework
LO-3.7	K1	Recall the characteristics of a code driven testing framework
LO-3.8	K1	Recall the characteristics of a hybrid framework

Below is a list of the most used types of test automation frameworks. These frameworks will be discussed in the following paragraphs with their associated advantages and disadvantages:

- Linear framework
- Modular framework
- Data-driven framework
- Keyword-driven framework
- Model-based testing framework
- Code-driven testing framework
- Hybrid framework

3.1 Linear Framework

LO-3.1	K2	Compare and differentiate the test automation frameworks
LO-3.2	K1	Recall the characteristics of a linear framework

A linear test automation framework is the most basic type of framework.

A test script is created for each test case, and the test scripts are run individually. This makes it possible to convert a (limited) number of manual test scripts into automated test scripts. This often happens with the record & playback functionality of the test tool. The scripts won't have reusable or shared functionality. The tools which are based on a linear framework are, in general, focused on ease of use and lack more complex features like reuse of code, data-driven testing, or scaling.

Advantages:

- No code needs to be written. The testers, therefore, do not need any programming knowledge.
- Creating the test scripts is fast by using record & playback.

Disadvantages:

- Not having reusable functions will result in poor maintainability. When, for example, a login page changes, all tests using that login page need to be changed.
- All actions are recorded in the script, which does not promote readability.
- The test data is part of the test script, which means that a test script cannot be executed with different test data.
- Not (easily) scalable.

3.2 Modular Framework

LO-3.1	K2	Compare and differentiate the test automation frameworks
LO-3.3	K1	Recall the characteristics of a modular framework

A modular framework addresses the main disadvantage of a linear framework as it uses reusable modules. For testing, the application is divided into different modules and a test script is made for each module. This division into modules makes it possible that changes in a module do not affect the other modules. The test scripts of the modules are then called from an overarching test script, which ultimately allows a complete workflow or test case to be tested.

Advantages:

- In case of changes in the application, only the test script for the affected module needs to be adjusted.
- Creating test scripts for test cases takes less time because (parts of) test scripts can be reused.

Disadvantages:

- The test data are still part of the test script, so scripts cannot be run with different test data.
- Programming knowledge is required to set up the framework.

3.3 Data-driven Framework

LO-3.1	K2	Compare and differentiate the test automation frameworks

It often happens that test cases must be run several times with different test data. Using a linear or modular framework in such cases is not efficient. In a data-driven framework, the test data are separated from the test script.

With a data-driven framework, it is possible to retrieve the input and expected output data from an external file or table. This can for example be an Excel sheet, TXT file, CSV file, or database.

Often this is a table-driven test. Each line in a table is tested sequentially. For each provided input the result of the tested system/function is checked against the predicted output.

Data-driven frameworks can be combined with Keyword driven or BDD frameworks [see 4.1].

Simple examples:

Input x	Output y (=x^2)
1	1
2	4
3	9
-4	16

Input a	b	Output c (= a AND b)
False	False	False
False	True	False
True	False	False
True	True	True

Advantages:

- Test scripts can be run multiple times with different test data.
- Maintainability is improved because test data can be developed independently and is separated from the test script.

Disadvantages:

- Requires extensive programming knowledge to link the external data files to the test scripts.
- The initial setup of a data-driven framework can be very time-consuming.

3.4 Keyword-driven Framework

LO-3.1	K2	Compare and differentiate the test automation frameworks
LO-3.5	K1	Recall the characteristics of a keyword-driven framework

A keyword-driven framework is a variant of a data-driven framework. Not only is the test data separated from the test script, but specific actions (code or keywords) are also stored separately from the test script in a file.

A keyword indicates action(s) that must be performed on the application. Each keyword represents one or more instructions. The keywords create some abstraction from the application as they can be written in natural language and are, therefore, easier to read.

Creating the code or actions used within keywords requires programming knowledge

Subsequently, minimum technical knowledge is required for the preparation of test scripts.

The framework can be written in the same language as the application itself. This eases learning and adoption of the framework.

Advantages:

- Minimum technical knowledge is required to draft test scripts with a keyword-driven framework.
- A keyword and the associated execution code can be used in multiple scripts. This ensures reusability and better maintainability.
- The test scripts can be created independently of the system to be tested.

Disadvantages:

- The initial costs to create this framework are relatively high compared to linear or modular frameworks.
- Development of the software used in this framework requires programming knowledge.
- When the number of keywords increases, the maintainability of this framework decreases.
- During the project, new keywords will be added continuously for which code must be developed.

3.5 Model-based Testing Framework

LO-3.1	K2	Compare and differentiate the test automation frameworks
LO-3.6	K1	Recall the characteristics of a model-based testing framework

These are two definitions of Model-based testing:

Definition 1:

"Model-based testing (MBT) is the automatic generation of test cases, using abstract models based on the requirements and behavior. Although this method of testing requires an extra effort at the start to set up a model, it offers significant advantages over other methods."

Definition 2:

"Model-based testing is the preparation of test cases based on models based on requirements and behavior. After which these test cases can be run manually or automatically."

Source: https://en.wikipedia.org/wiki/Model-based_testing

The main difference between these two definitions is that "Definition 2" does not assume that the test cases are always automatically derived from abstract models.

When you start from Definition 2, many testers already use MBT "unnoticed". They often create flows or graphs to visualize the functionality to be tested and then manually derive the logical test cases.

To understand what model-based testing is, you need to understand what a model is. A model describes the behavior of a system. The behavior can be described in terms of input sequences, actions, conditions, output, and the data flow from input to output. Read the syllabus Certified Practitioner in Agile Testing for more info on models.

There are numerous models available. The models describe various aspects of system behavior. Some examples of models:

- Data flow
- Process flow Count
- State transition diagrams

Advantages:

- Effectiveness
- Modeling promotes communication between the parties involved. For example, stakeholders, designers, developers, and testers
- Better communication helps create a shared vision and understanding of the requirements and detect possible misunderstandings.
- Graphical models make it easier to involve stakeholders.
- Drawing up (automated or not) and analyzing test cases is already possible before any code has been created.
- Efficiency
- Drawing up models early can prevent mistakes in the development process.
- Models can be verified by the stakeholders (business analysts, designers, developers) so gaps can be detected early.
- MBT supports the automatic derivation of test cases and thus ensures that mistakes made when preparing the test cases manually can be avoided and makes the maintenance of test cases more manageable.
- Different sets of test cases can be derived from a model.
- MBT helps reduce maintenance costs as requirements change.
- Only the model needs to be modified.

Disadvantages:

- The test is 'as good as the model', as a model is a schematic view of reality and never an exact representation of the reality of what a product looks like, so it has a particular abstraction. As a result, details can be overlooked.
- Drawing up a model can be very labor-intensive and contain errors.
- It may be possible to capture parts in a complete model at a low technical level. At a higher level, the number of possible paths and combinations becomes so large that a model *must* be simplified.

3.6 Code-driven Testing Framework

LO-3.1	K2	Compare and differentiate the test automation frameworks
LO-3.7	K1	Recall the characteristics of a code-driven testing framework

Code-driven test automation is mainly promoted in the TDD (Test Driven Development) method. Before writing the development code, unit tests are developed. Tests are developed to specify the code functionality. Once the tests pass, the code is refactored and retested before being considered complete or good. The process of TDD is initially slow as it takes time to get accustomed to. More on TDD in paragraph 4.2.

To avoid duplication of code, the development team will sometimes perform code-driven testing together with a tester. The application needs a modular design. As code-driven testing influences the planning of code, it's recommended that all team members use it.

Advantages:

- This framework covers the bottom layer of the test pyramid.
- This method is also useful to test the software of user interfaces.
- Code coverage is high and creates a higher reliability of the product.
- The results of unit tests will show if the behavior is the code works as expected under different kinds of sections and numerous circumstances.
- Good way to early detect defects in components or modules of the software.
- Features a straightforward interface.
- Easier to maintain.
- Easier to refactor.
- Tests document the code.
- Less debugging.

Disadvantages:

- Tests won't reveal defects that are introduced within both the test code and implementation code.
- All team members need to be involved and work on code-driven testing.

3.7 Hybrid Framework

LO-3.1	K2	Compare and differentiate the test automation frameworks
LO-3.8	K1	Recall the characteristics of a hybrid framework

This framework contains all the possibilities of the frameworks described above. This makes it possible to use the advantages of other frameworks and thereby reduce the various disadvantages. Developing and maintaining a hybrid framework requires knowledge acquisition and will therefore take more time than implementing one of the frameworks earlier mentioned [HT1].

Advantages:

- Improved test efficiency.
- Lower maintenance costs.
- Minimal manual intervention.
- Maximum test coverage.
- Reusability of code.

Disadvantages:

- Knowledge of the combined frameworks is required.
- Fully scripted tests will increase the automation effort.

4 Test Development Approach

Different methods are used to approach test development. These methods help create a specific structure within development and testing and for test automation. This chapter Discusses Behavior-Driven Development (BDD) and Test-Driven Development (TDD).

The syllabus AU Capturing Agile Requirements by Example (CARE) [AU1] provides more information on BDD and TDD.

Keywords

Behavior Driven Development, BDD, Test Driven Development, TDD, The Three Amigos

LO-4.1	K1	Recall Behavior Driven Development, abbreviated BDD
LO-4.2	K2	Recall The Three Amigos
LO-4.3	K1	Recall Test Driven Development, abbreviated TDD
LO-4.4	K2	Compare and differentiate BDD and TDD

4.1 Behavior Driven Development (BDD)

LO-4.1	K1	Recall Behavior Driven Development, abbreviated BDD
LO-4.2	K2	Recall The Three Amigos
LO-4.4	K2	Compare and differentiate BDD and TDD

Behavior Driven Development (BDD) enables the team to communicate and close the gap between 'technical' and 'business' people. BDD is a framework for software development activities that helps teams develop valuable software of high quality faster.

BDD uses Agile and Lean practices and is based on the following three principles:

- Less is more. Do just enough analysis and design beforehand to get started.
- Deliver value for the stakeholder. Sole effort is in what produces value or increases the ability to deliver value.
- It's all about behavior. At every level, the same language can be used to describe the system's behavior.

BDD helps to identify the behavior of a function clearly. The behavior is identified by describing realistic examples rather than being described in abstract and generic jargon. The idea is to define the behavior of the software together with all team members, representatives from the business, and possibly even customers. BDD focuses on added value for the business. By prioritizing the business based on the added value that functionality provides. This can be done through a specification workshop, during which participants are instructed to think about how the system should behave and describe it according to a fixed format. “The Three Amigos” is a great way to make these conversations happen. In essence, “The Three Amigos” approach means that three key stakeholders, a business representative (likely a PO), a developer, and a tester, work together to develop a user story or feature before starting the development of the software.

Advantages:

- BDD increases and improves collaboration. It enables everyone involved in the project to engage in system development easily.
- By using simple language, everyone involved can write scenarios.

- Developers can deliver better results because they have a good understanding of what the business needs. By focusing on value, no useless functions are built.
- Developers can deliver better results because they have a good understanding of what the business needs.
- By focusing on the value, no useless functions are built.
- Improved code quality reduces maintenance costs and minimizes project risks.
- All developed software can be traced back to business objectives.

Disadvantages:

- Requires a lot of cooperation and involvement from the business/stakeholders.
- BDD is not suitable for a waterfall approach. The analyst is no longer the only one who determines the requirements; testers don't wait to register findings until the end of a release, and developers regularly consult with the rest of the team.

4.2 Test Driven Development (TDD)

LO-4.3	K1	Recall Behavior Driven Development, abbreviated BDD
LO-4.4	K2	Compare and differentiate BDD and TDD

In Test Driven Development (TDD), you produce very small increments of tests and regular code. Unit tests are written before the implementation of the development code. At first, you have a test that fails (as there is no functional code yet), and then you write the code to make the test pass. After writing each increment, you switch between testing and coding, so you write a single test, write the code to make the test pass, write another single test, write the code to make the test pass as well, etc.

Applying TDD ensures that all functionality is covered by unit tests. This way, a solid bottom layer of the test pyramid is created (see 2.2).

5 Complementary Tools and Components

A single, out-of-the-box test automation tool will generally not be sufficient for all desired tests. Sometimes, capabilities need to be expanded by adding additional libraries or supportive tools. This chapter describes components and tools that can be complementary in a test automation setup.

Keywords

Assertions, Assertion libraries, Test harness, Stub, Driver, User Interface (UI), Graphical User Interface (GUI), assertion library, integration tests, web services, performance testing, security testing

LO-5.1	K1	Recall the definition of assertions and an assertion library
LO-5.2	K1	Know what a test harness is
LO-5.3	K2	Understand the difference between a stub and driver
LO-5.4	K2	Understand which tool fits a user interface test
LO-5.5	K2	Understand how performance tests need to be approached
LO-5.6	K2	Explain the importance of security testing
LO-5.7	K1	Remember the characteristics of security testing
LO-5.8	K2	Understand the need to prevent potential legal, financial and operational issues

5.1 Assertions and Assertion Libraries

LO-5.1	K1	Recall the definition of assertions and an assertion library

An assertion library is a set of functions or methods for checking whether a given condition is true in a software application. Assertions are used in automated tests to verify an application's behavior and ensure that it is working as expected.

Assertions make tests meaningful by verifying the results during test execution. An example of a statement might be a function that adds two numbers and confirms the correct result after running a test. To assert equality, equate the actual result with the expected result. The assertions library provides simple test sets of assertions that can be used to verify results.

Assertion libraries are often included in testing frameworks and are available in most programming languages. They provide a simple and concise way to write tests and check an application's behavior. These assertions can be explicit or implicit. Explicit assertions are checks of values and properties; see Example 1. Implicit assertions are, for example, clicking a button or link or inputting text. The click asserts that the button, link, or input is present and enabled. See Example 2.

Within test automation, different types of assertion libraries can be used in a test automation project. An assertion library works best when built into the testing framework, making downloading and adding dependencies unnecessary. In addition, many test automation tools allow you to write your statements. If the selected framework does not have an assertion library or the library is missing specific features, it is sometimes possible to add or change a library.

Example 1: Explicit assertions

```
var assert = require('chai').assert
  , foo = 'bar'
  , beverages = { tea: [ 'chai', 'matcha', 'oolong' ] };

assert.typeOf(foo, 'string'); // without optional message
```

```
assert.typeOf(foo, 'string', 'foo is a string'); // with optional message
assert.equal(foo, 'bar', 'foo equal `bar`');
assert.lengthOf(foo, 3, 'foo`s value has a length of 3');
assert.lengthOf(beverages.tea, 3, 'beverages has 3 types of tea');
```

Source: https://www.chaijs.com/guide/styles/

||||||||||||||||||||||||||||||

Example 2: Implicit and explicit assertions

Example of Gherkin style test case (Given, When, Then) written in Robot Framework with both implicit and explicit assertions.

Implicit:

- Input text
- Click

Explicit:

- Wait Until Element Is Visible
- Should be equal as strings
- Element Should (Not) Be Visible
- Page Should Contain Element

```
*** Test Cases ***
Scenario: Search on a complete name
   Given I search for an owner named Davis
   When  I click Find Owner
   Then  I should find Betty Davis and Harold Davis

*** Keywords ***
I search for an owner named ${owner}
   Open search page
   Input Text    //input[@id="lastName"]    ${owner}

I click Find Owner
   Click element    //button[@type='submit']

I should find ${owner_1} and ${owner_2}
   # Wait for the site to show results page 'Owners' after the click:
   Wait Until Element Is Visible    //h2[text()="Owners"]
   # Assert (explicit) that elements are gone that were on previous screen:
   Element Should Not Be Visible    //button[text()="Find Owner"]
   Element Should Not Be Visible    //a[text()='Add Owner']
   Element Should Not Be Visible    //input[@id="lastName"]
   # Grab all user names using Selenium:
   ${list}=    Get list of owners on current page
   # Explicit assert that the names match:
   Should be equal as strings    ${owner_1}    ${list}[0]
   Should be equal as strings    ${owner_2}    ${list}[1]

Open search page
   # Open search page:
   Click Element    //a/span[contains(text(),"Find owners")]
   # Make sure we are on the right page:
   Page Should Contain Element    //h2[text()="Find Owners"]
   Element Should Be Visible    //button[text()="Find Owner"]
```

```
  Element Should Be Visible    //a[text()='Add Owner']
  Element Should Be Visible    //button[@type='submit']

Get list of owners on current page
  @{listOfOwnersAsText} =    Create List
  @{listOfOwnersAsElements}=    Get webelements    //tr/td[1]
  FOR    ${ownerAsElement}    IN    @{listOfOwnersAsElements}
    ${owner}=    Get text    ${ownerAsElement}
    Append To List    ${listOfOwnersAsText}    ${owner}
  END
  Sort List    ${listOfOwnersAsText}
  [Return]    ${listOfOwnersAsText}
```

Example test case based on the project PetClinic created by The Spring PetClinic Community [PC1]

5.2 Test Harness, Stubs, and Drivers

LO-5.2	K1	Know what a test harness is
LO-5.3	K2	Understand the difference between a stub and a driver

A test harness is comprised of software and test data configured to support automated tests. Some components of a test harness:

- Drivers
- Stubs
- Repository with test scripts
- Report engine

Stubs and drivers are tiny programs that mimic interfaces or modules and are used to test the actual software under test. Stubs and drivers replace modules that are missing or not yet developed. The reason for missing can be a deliberate choice, for instance, because of security concerns, costs of hardware or licensing, or the strategy of making tests as small and fast as possible to get a solid set of unit tests. A stub is a passive tool that responds to requests. A driver is an active piece of code that provides input, controls, and verifies execution.

5.3 User Interface Testing

LO-5.4	K2	Understand which tool fits a user interface test

User interface (UI) tests are focused on the visible part of an application and check whether the frontend of an application works as expected (see 2.2). UI testing is performed on

- Browser
- Mobile and tablet apps
- Desktop applications
- Embedded applications

There are two basic approaches used in automated UI testing:

- Image recognition
- Browser API-based

The most basic use of image recognition is comparing (a part) of the screen with an image stored as expected. The test passes if the screen capture exactly matches the expected image. These tools can be tricky to use as the chance of failing automated checks grows when:

- A minor change is made to the UI
- A different font is used on the UI
- An element is moved by a few pixels in the UI

The required match can be eased by allowing some tolerance on, for example, size, position, brightness, or color. Some tools have additional features like optical character recognition to match an expected text or use artificial intelligence to search and match specific UI elements [MT1].

API-based tools interact with the API of a browser or the (graphics) system to control and check screen elements like buttons, checkboxes, dropdowns, application windows, etc. The tools assert (see 5.1) that the element is visible, enabled, clickable, etc.

5.4 Performance Testing

LO-5.5	K2	Understand how performance tests need to be approached

Performance is one of the aspects that can be covered by automation. Performance testing gives insights into the processing time, responsiveness, resource utilization, reliability, and scalability of a system. Performance testing is placed in Q4 of the testing quadrants (see 2.1).

This syllabus uses a broad approach to performance testing. Load, volume, and stress testing are closely related to performance testing. All tests put a particular strain on the system for a certain amount of time and the response of the system is monitored. This can be a workload for a prolonged period of time or just the execution of a short task or job. Variations can be created by varying the load gradually or abruptly over time. By simulating an overload or temporary outages of components and services the resilience of the system can be tested.

The most basic way of measuring response times is by using a stopwatch. Manually starting and stopping a stopwatch is labor-intensive and error-prone or can be impossible when measuring events that take milliseconds. Automated performance tests make it possible to measure fast events and combine them with scenarios while the results are automatically collected. Examples of scenarios are increasing the number of requests by increasing the frequency or ramping up the number of simulated users, generating additional I/O, or decreasing the bandwidth.

Performance testing requires a thorough understanding of the entire test environment, the scope of the test, the workload mix, and the requirements. Do not only focus on the system's software but also discuss the proposed test parties involved, like the IT department, service providers, etc. Tests need to be well-prepared and should be approached as a multi-disciplinary (team) effort. The various areas of expertise make it possible to take the whole system into account and not just focus on a single aspect of an application, such as a database. Possible participants in setting up a performance test team:

- Product Owner and business stakeholder(s)
- Development team
- Infrastructure and network specialist(s)
- Database engineers

Executing a performance test in the test or acceptance environment might have unexpected side effects on your production environment or other test environments and lead to extra costs or even financial claims. Even when well-planned and set up by a multi-disciplinary team, a component (service, connection, etc.) might be shared and impacted by a performance test.

Just like regular functional tests, performance tests need maintenance. As applications change, the performance tests used should be regularly checked and updated.

A dedicated environment to conduct performance tests should be available, if possible. Factors that influence performance tests:

- Test environments are, in general, less well equipped when compared to production environments: for instance, fewer servers, less memory, and less allocated CPU or bandwidth.
- Absence of interfaces or stubbed interfaces.
- Test environments might have a smaller database or slower disks
- Probe effect
- Determining the most suitable operational profiles

Related to reliability testing are resilience testing and chaos testing. Both types of tests are based on creating some kind of disturbance in the system while the system has a specific load. These disturbances can be any kind of (simulated) failure or extreme load on the system. For instance, a broken disk, low bandwidth, a network interface that is not responding anymore, or a query that runs for a very long time. The system response can be examined, e.g., what errors are generated, what is logged, how many transactions fail, and if the system can recover from such a disturbance.

5.5 Security Testing

LO-5.6	K2	Explain the importance of security testing
LO-5.7	K1	Remember the characteristics of security testing
LO-5.8	K2	Understand the need to prevent potential legal, financial, and operational issues

Software security testing is a process of evaluating the security of a software application or system to identify vulnerabilities and potential threats. A vulnerability is a flaw or weakness in the system that compromises the system when it is exploited. This might result in, for example, loss of data, loss of revenue, or might have legal consequences of failing to comply with legislation. A threat can be anything: a malicious external attacker, an internal user, system instability, etc. [IO1].

Different testing techniques, such as penetration testing, code review and threat modeling can be used to uncover security issues and evaluate the security of the software. Effective software security testing requires a combination of technical expertise and a thorough understanding of the software and its architecture. Software security testing can be performed at different stages of the software development life cycle, including design, implementation, and deployment.

Automated security testing (AST) refers to the use of software tools and scripts to automate the process of security testing. These tools simulate various security threats and attack scenarios and provide quick feedback on the software's vulnerabilities and potential risks. This can be achieved by incorporating these tests in the continuous integration pipeline, periodically scanning a software repository, or running the AST test regularly on a test environment.

Many products are available for security testing, and the tools are quite diverse. Some are highly automated and perform some kind of checking, whilst other tools support automation-supported testing as they need human interaction to operate and evaluate the results.

Well-known security test techniques:

Vulnerability Scanning

Vulnerability Scanning uses an automated tool to scan for vulnerabilities and loopholes. This can be either dynamic or static. The first involves a running system, whereas the latter relies on scanning the source code [VS1].

Penetration Testing

Penetration Testing, also known as Pen Testing, involves simulating an external hacker or attacker. You can perform a penetration test using an automated system or a manually executed collection of scripts.

Access Control Testing

Access control testing allows you to check if the application under test (or parts of it) can be accessed by unauthorized people. The objective of this test is to make sure the application complies with the security policy.

The OWASP Top 10 represents the most critical security risks to web applications [OW1], and it is a starting point for approaching web application security.

Before performing security testing like vulnerability scanning or penetration testing, it is wise to think about potential legal, financial, and operational issues that might arise. In general, remote security testing is considered legal as long as it is done with the website owner's permission. When you test your own application, this should be no issue. However, applications are generally not stand-alone but part of a more extensive system of network appliances, servers, and other components. Larger organizations or (cloud) hosting providers generally have some kind of intrusion detection and network monitoring software in place. This software might get triggered, raise the alarm, and even take automated precautions like blocking the IP address or a whole IP address block used for testing. This might seriously impact the operation of an organization. Getting consent beforehand and exchanging contact details with the parties involved is advised.

6 Test Automation Approach and Strategy

How should test automation be approached? What are the next steps? This chapter discusses the steps to formulate a test approach to determine a fitting test automation strategy.

Keywords

Team involvement, ownership, responsibility, Software Development Life Cycle (SDLC)

LO-6.1	K1	Recall getting the whole team's involvement
LO-6.2	K2	Describe the frequently used responsibilities for a unit, integration, and UI tests
LO-6.3	K1	Remember Software Development Life Cycle (SDLC)
LO-6.4	K2	Summarize different kinds of SDLC

6.1 Generate Team Involvement

LO-6.1	K1	Recall getting the whole team's involvement

Test automation requires investment, commitment and continuous effort. Successful test automation needs a whole-team approach. The whole team should be involved in the planning, execution, and maintenance of test automation when starting a new software delivery project. The whole team includes developers, testers, product owners, analysts and business stakeholders and ensures that all skills and knowledge are available to accomplish the goal. Make decisions as a team about how to implement and control test automation.

Working together, the team can make decisions about:

- What types of tests to implement
- Which test scenarios to implement
- Which tools to choose
- Who will be responsible for each level of testing
- How to collaborate with the stakeholders
- How to maintain test automation

Start by creating a shared understanding of test automation to align expectations. Spend time determining what aspects a good automation test suite should have. Start by creating a shared understanding of test automation to align expectations. Each team member should be able to give their view on what is valuable in a test automation suite. Something that one person considers critical might not be important to another team member. It might be surprising when you find out that stakeholders have different focuses and knowledge. Using models like the testing quadrants and test pyramid (see 2.2), to guide discussions on the critical points and determine what types of tests will make up the automation suite. People are all different, some can express themselves easily, whereas others may have more difficulty sharing their opinions. To get a brainstorming session off to a good start, a discussion using “De Bono Thinking Hats” and Toulmin's reasoning scheme (see 1.3 and 1.4), can add a lot of value. Doing these different exercises will help ensure that all stakeholders agree on the strategy for automation testing going forward.

6.2 Thinking About Ownership and Responsibility

LO-6.2	K2	Describe the frequently used distributions of ownership for a unit, integration, and UI tests

Once implemented, it is important to regularly evaluate the test automation. This can be in the regular retrospective at team level, but also be an expert group. This will help to jointly address what is going well with the test automation and what can be improved.

Once the team has identified what types of tests to include in the project's test automation test, determine ownership of each type of automation test. A typical example of ownership is as follows:

- Developers own the unit tests because the unit tests are written alongside the development work
- Developers and testers own integration tests because both focus on the integration during testing
- Testers own the user interface tests and tests of non-functionals because testers often focus on the system as a whole.

Usually, this way of ownership is set up within projects, but it doesn't always have to be that way.

The team decides who owns which tests within the project. Naming owners for specific test levels does not mean other team members cannot help with those test levels. Often, the help of other team members contributes to expanding knowledge about the test automation and the project itself. Quality is the responsibility of the entire team.

There are many ways that the entire team can be involved in test automation. For example, business stakeholders can help developers and testers determine high-priority test scenarios, testers can help developers devise effective test cases for unit testing and integration testing, and developers can help testers write user-interface test scripts. Finally, testers and developers can report relevant test results to business stakeholders.

Test automation projects will gradually evolve. New tests need to be added, maintenance is required, and there is always room for improvement. By collaborating as a team, test automation can be kept maintainable at a high level, contributing to the value and trust of the software under test.

6.3 Thinking About the Development Processes

LO-6.3	K1	Remember Software Development Life Cycle (SDLC)
LO-6.4	K2	Summarize different kinds of SDLCs

Software Development Life Cycle (SDLC) is often regarded as a set of stages or phases in the development process. There are several SDLC models for the system development lifecycle. Examples with an agile context are the Agile Software Process (ASP), SCRUM, Rapid prototyping model and Extreme Programming. Waterfall and the V-model are more traditional approaches of a SDLC.

Each SDLC model describes approaches to a series of tasks or activities that occur repetitively during development processes. Common activities include:

- Analysis
- Design

- Development
- Testing
- Release
- Maintenance

The chosen model used for the SDLC impacts which tests should ultimately be automated (See 6.4), how to automate them, and when and where to execute them. Different types of tests are performed at different points within the software delivery processes. Automated testing can be performed both locally, such as running from a local machine like a laptop or desktop, or running remotely on a (cloud) server, for instance when it is part of a continuous integration (CI) setup. It is essential to run tests often and to ensure that they continue to pass as the software continues to evolve.

6.4 Define a Test Strategy

Before starting to write tests, it is necessary to determine the test strategy and an agreed-upon test automation approach within the team. Involve stakeholders to determine a list of the highest-priority features. Discuss for each feature which scenarios can be automated and which test scenarios are tested manually. This should be clear at the beginning of the project, although there may still be some adjustments, add-ons, or changes later on.

Test scenarios well suited for automation are:

- Scenarios of high-impact features. The impact can be determined by risks, value, ROI, and cover the system and/or business.
- Scenarios that take a lot of time to execute manually
- Scenarios that are difficult to test manually.
- Scenarios that should show the same results every time.

The whole team sits together to:

- Identify these scenarios from automation
- Determine the type of test for each scenario
- Consider the inputs, actions, and outputs expected during testing.
- Decide which test levels will be part of test automation

It is recommended to have a solid base of unit tests, a medium number of integration tests, and a smaller number of user interface tests (see 2.2). If a user interface scenario shows the same results as a unit or integration test, the latter should be preferred, and the user interface should be skipped for this scenario.

It is also important to consider additional required resources and decisions such as:

- How test data will be created, used, and maintained?
- Applying style guides, design patterns, naming conventions, and similar agreements to ease test code maintenance?
- Which tool will be used for building and running automated tests?
- Which test environments are needed for automated and/or manual testing?
- Who has responsibility for these resources?

Thinking as a team about test automation and the resources required for it, you should draw up your test strategy, for example, in a document or on a wiki page of a scrum team. This helps the current team, but also future teammates, handle test automation in the right way.

7 Test Automation Tool Selection

By using the Agile Quadrants, the Test Pyramid, and knowledge of test automation frameworks, the team can make a joint decision about what the test automation will look like. The team should also decide which tools they need for test automation.

Keywords

Life cycle, Open source, Closed source, Licensing, Support, Documentation, Pricing, Headless testing, Test results and metrics, Data governance, CI/CD tooling, Spike

LO-7.1	K2	Summarize factors that influence the selection of a tool
LO-7.2	K1	Recall feature gap
LO-7.3	K2	Interpret technical aspects that need to be considered
LO-7.4	K2	Interpret licensing and support aspects that need to be considered
LO-7.5	K2	Interpret operational aspects that need to be considered
LO-7.6	K2	Understand the importance of the life cycle of a tool
LO-7.7	K2	Understand the importance of test results and metrics
LO-7.8	K2	Understand the importance of data governance
LO-7.9	K2	Explain the concepts of a spike

7.1 Factors in Selecting a Tool

LO-7.1	K2	Summarize factors that influence the selection of a tool
LO-7.2	K1	Recall feature gap
LO-7.3	K2	Interpret technical aspects that need to be considered
LO-7.4	K2	Interpret licensing and support aspects that need to be considered
LO-7.5	K2	Interpret operational aspects that need to be considered
LO-7.6	K2	Understand the importance of the life cycle of a tool
LO-7.7	K2	Understand the importance of test results and metrics
LO-7.8	K2	Understand the importance of data governance

Before looking at the market for available tools, the team should first decide what the requirements for this tool are. Some projects require specific features that are not available in every tool like for instance support of headless testing or easy-to-use reporting. The following is a non-exhaustive list of factors that influence the selection of a tool for test automation:

Technical:

- Compatibility with the technical environment (software and hardware)
- Integration with CI/CD tooling
- Support of headless testing

Licensing, pricing, support, and costs:

- Open or closed-source software
- Licensing and pricing
- Support
- Costs

Operational:

- The life cycle of the tool
- Test results and metrics
- Data governance

Technical

Compatibility with the Technical Environment

It might seem obvious, but tools do not support all possible combinations of hardware and software. The tool can be run on the same device as the system under test (SUT), but in many cases, the tool and SUT will differ. Therefore, it is helpful to ask questions like:

- What are the requirements to run the tool? Hardware, operating system, other required software like databases, drivers, etc.?
- What hardware/platform can the tool test? Server, desktop, laptop, tablet, mobile, wearable, etc.?
- What target operating systems (and versions) does our system support, and what does the tool support? Windows, MacOS, Linux, Android, iOS, etc.?
- Which target browsers does the tool support?
- Can the tool test 'native' apps?

Integration with CI/CD Tooling

An overlooked requirement might be using a tool in a continuous integration (CI) setup. When a tool only runs on a tester's local machine (laptop, desktop), it is impossible to take advantage of the possibilities of CI, such as running tests automatically on each pull request. More on CI in paragraph 10.3 Use Continuous Integration.

Headless Testing

Headless testing means testing with a system with no output to a screen; there is 'no head' (display) attached. In most cases, headless testing refers to headless browser testing. Most modern browsers have the capability to run in a headless mode. Because the screen is not rendered, these tests are generally faster, making it easier to run tests in parallel. A headless browser is executed invisibly in the background. Headless testing is often required for integrating UI tests in a CI/CD pipeline.

Licensing, pricing, support, and costs

Licensing, pricing, and support are closely related. In general, open-source software is free and has no official support, whereas closed-source software is licensed, needs to be paid for, and is backed by official support. This is an extreme simplification of reality. Some open-source projects have a very active community that is willing to answer questions quickly and provide help. Some companies provide paid support on open-source software. Depending on the contract or subscription, some companies have different product service levels. It is, therefore, advisable to estimate the costs related to the acquisition and use of a tool. Some possible questions related to licensing:

- What is the licensing and pricing model?
    - Price per user: unique users, concurrent users, named users, and machine users
    - Price per CPU or core
    - Price per use: time, CPU time, number of API calls
    - Flat fee: site license, corporate license, etc.
    - Commercial, educational, or non-commercial use

    - Free: open-source
- Is it a subscription or a one-time license?
    - With or without updates?
    - What customer service is provided?
- How does the licensing scale when the demand changes?
- Is it possible to run tools locally and in a CI/CD environment?

Some questions related to costs:

- How much time (and thus money) do we need to get experienced with a tool?
- Is training required?
- Is additional software needed? Quite often, you need extra software to run a tool. This can be anything from (free) browser plugins to additional database instances or disk storage for test data.
- Is additional hardware needed? Some tools require dedicated hardware and dedicated instances of an operating system.

Operational

The Life Cycle of the Tool

Over time, tools' importance and popularity will change. Using a brand new 'cutting-edge' or even 'bleeding-edge' tool can be very appealing. Bleeding-edge software refers to the latest technologies, often offering new features and capabilities to improve productivity and efficiency. However, this software can also be untested and unstable, might have security vulnerabilities, or might have compatibility issues with existing systems. A feature gap can exist when some desired, or even worse, required functionalities might not yet be available in a tool.

Tools that have existed for many years can also become a problem, as software can rely on external components or libraries or use specific functions of an operating system. When these components are deprecated because of security issues, fading community interest, or simply stop working because of an incompatible new operating system, selecting that tool might not be the best idea.

Test Results and Metrics

The idea of automated testing is to run tests repeatedly (over and over again). For the tests under review, it's good to have a way to run simple tests in a repeatable manner and then view the results of the tests performed. Selecting the right tool(s) is crucial to achieving precise results.

Many tools can display test results. However, not every tool is user-friendly when it comes to test results. Test results can be shown, after which you still have to investigate where something is not right and whether it is indeed a bug or whether the automated test may need to be adjusted. Many tools are now hybrid, allowing adding plugins or your software code to see all in-depth test results quickly.

Adding your design to the test results can present the data clearly and conveniently. Some tools generate beautiful and detailed reports when the team or business requires them. To achieve this, people must have somewhat advanced knowledge of test automation. In any case, remember that it should be easy to perform tests, that results can be clearly interpreted, and that causes can be found quickly.

Data Governance

Where are data such as program code, test scripts, test data, results, and reports stored, and who has access to this data? Especially when having (contracted) third parties or when cloud services are used, it is wise to ask yourself questions like:

- What information is shared with suppliers/providers?
- Is it sensitive data (competitive, privacy, security, …)?
- Who is the owner of the data?
- Who can access the data?
- Is it allowed by local law to store data in the (unspecified) cloud?
- Do we need an escrow agreement for access to the data in case of bankruptcy or other legal issues?
- What is our organization's policy on storing data externally?
- Can we easily migrate to another tool by extracting relevant data?

7.2 Selecting the Correct Test Tool(s)

Test automation tools should support the agreed-upon testing strategy. If testing tools are selected before the team adequately discusses them, several necessary options may very likely be unavailable within the chosen test automation tool. Therefore, always select the right tool from the agreed-upon test levels after determining the test strategy as a team.

Do not choose a test tool based on someone who knows how to use it. If you choose the tools first, before you know what you want the test automation to look like, you risk excluding valuable, needed aspects by choosing the wrong tool.

As discussed earlier in Chapter 3, different test automation frameworks are available for the different test levels. It can be challenging as a team to agree on selecting the most appropriate test automation framework. It is best to discuss this together as a team and make the most informed choice. However, there is no single test automation framework that meets all the team's needs. Therefore, the aim is to find the tools most suitable for performing the chosen test types. Start with two basic requirements for choosing the right tool:

- What kind of test should be implemented in the test automation tool?
- Which programming language must be used to write the test cases for the test automation tool?

Preferably, the tool should promote cross-functional collaboration between the testers, developers, and business team members. These tools will lead to better testable code and fewer defects.

7.3 Run a Spike

LO-7.9	K2	Explain the concepts of a spike

There will be many uncertainties when selecting test automation tools. A time-boxed spike can help reduce these uncertainties. The purpose of a spike is to gather information by experimenting.

A spike is created when the team doubts the fit of a specific tool or does not know all the details and wants to fill that knowledge gap. A spike helps to understand better or collect unknown

requirements. A spike can help to estimate how much time it will take to implement the chosen test automation tool.

Though a spike does not deliver business value as a story, it helps the business with its delivery in upcoming iterations. This spike can be added to the backlog.

Run a spike on one or more selected tools by doing small experiments. These experiments help the team to learn about the use of a tool and its technical details leading to an informed judgment about a tool. List the pros and cons that emerged while running a spike on the test automation tools. The team can compare and discuss the tools to select the best-suited tool(s).

When there is still doubt about a specific test tool's usability or capabilities, one should consider conducting a pilot or a proof of concept (PoC). This will take more time and should go beyond the scope of a single user story in a single iteration.

8 Deciding What to Automate

This chapter describes the steps to make a thorough decision about which test cases to automate based on the chosen approach, strategy, and tool(s).

Keywords

Test scenarios, Value, Risk, Impact, Probability, Costs, Ordering

LO-8.1	K2	Explain the steps to take to decide what to automate

Do all test cases need to be automated? What would be the first test to automate? When many test scenarios can be chosen from, it is important to choose which scenarios to automate and subsequently which scenarios to automate first. To make a thorough decision on what cases to automate the next steps should be made:

1. Collect and investigate test scenarios
2. Value the scenarios
3. Order scenarios by risk
4. Analyze the costs of automation
5. Decide on the scenarios to automate

Step 1 – Collect and Investigate Test Scenarios

The first step is to research and write down test scenarios. The goal is to collect many different scenarios and avoid discussions on aspects like complexity, feasibility, etc.

Step 2 - Value Each Scenario

The list of test scenarios can be exhaustive, so automating everything may not be necessary. Therefore, have the team assign a value to each scenario. The list of values can help decide which tests are most valuable to automate.

A major aspect to consider is its importance/relevance. What is essential to one team member may not be necessary to another. Thinking about the scenarios together also promotes cooperation within a team.

In addition, it is also necessary to consider feasibility, depending on how much time and capacity a team has for test automation.

A scenario might not seem valuable initially, but it might be necessary to automate, as it enables other scenarios or is even required to test other system parts. For example, a scenario to log in will generally be one of the first automated scenarios, as all other user functions require a user to be authenticated and authorized.

Step 3 - Order Scenarios by Risk

To determine the order in which test scenarios from the list should be automated, first, the previously selected test scenarios need to be reviewed from a risk perspective. The riskiest test scenario is captured first in the test automation, and the least risky test scenario is captured last in the test automation. It can help a team to assign a risk score to the test scenarios.

The risk is determined based on two factors:

- What is the impact of the function? If it is broken, what is the impact on customers and/or clients?
- What is the probability of use? How often is this feature used by customers or clients?

Risk assessments allow teams to consider how often a feature will be used and what will happen if it breaks.

Step 4 - Investigate the Costs of Automation

Risk and value are critical when considering test scenarios, but it's also important to consider how much it can cost to automate a test. Think about the time, effort, and money. Here too, the team can work with scores. A score can determine whether a test script is easy to write and/or how quickly the test will be written. By attaching a price tag to it, the team sees the various possibilities for the test scenarios to be automated.

Step 5 - Select What to Automate

After examining each test scenario's value, risk, and cost, enough information should be available to select the best (if eligible) test scenarios for automation. It does not mean that the other test scenarios are not important, quite the contrary, the focus is first on the test scenarios from the top list to be recorded in the automation. Of course, focusing on a more extensive list of test scenarios is possible.

When a team scores the potential test cases for automation based on value, risk, and cost, it helps quantify this data and makes it easier to select what to automate. Feel free to adapt this model as a team to suit the team's needs best.

9 Good Practices in Test Automation

Once the decisions about what to automate have been made, you should determine how to automate.

Keywords

Test design patterns, Don't repeat yourself (DRY), Domain-specific language (DSL), Maintainable and healthy test

LO-9.1	K1	Understand the value of test design patterns
LO-9.2	K2	Understand the value of principles
LO-9.3	K2	Understand different kinds of patterns
LO-9.4	K2	Understand the concept of DRY
LO-9.5	K2	Understand the concept of DSL
LO-9.6	K1	Recall the characteristics of a maintainable and healthy test

9.1 Follow Test Design Patterns

LO-9.1	K1	Understand the value of test design patterns
LO-9.2	K2	Understand the value of principles
LO-9.3	K2	Understand different kinds of patterns

Design principles and patterns help to reduce the cost of writing and maintaining automated test scripts. Create a test design and look for ways to improve it. This helps keep maintenance costs to a minimum in the long run. And it helps get quick and helpful feedback. Section 9.4 Maintain Standards explains a few key principles that ensure maintainable and consistent test automation.

Some guidelines to keep in mind:

- Each test should have a single goal. As a result, tests have a more precise scope, are easier to debug, and are easier to update as business rules change.
- Tests should be completely self-contained. They should be able to be run in any order and should not rely on data from other tests.
- Tests should consist of steps describing the behavior. The technical details of each step must be defined in a function outside of the test. By abstracting the technical steps at a lower level in this way, the test becomes more legible.

The list of design patterns and practices is not exhaustive.

Decide which test design patterns are essential to the team. Document those patterns in detail to enable others to understand the principles and how to structure the tests.

9.2 Do Not Repeat Yourself (DRY)

LO-9.4	K2	Understand the concept of DRY

“Do not repeat yourself”, abbreviated to DRY, helps avoid duplicated tests in automation. DRY can ensure that only one test component needs to be updated when something changes in the system under a test. DRY helps with sharing and reusing test code within test automation.

For example, with DRY you try to reuse a first test in a second additional test. When you adjust the first test, it is also adjusted for the additional test, etc.

9.3 Use Domain-Specific Language (DSL)

LO-9.5	K2	Understand the concept of DSL

A domain-specific language, or DSL, describes items specific to the software under test. It is best to give each entry in the software under test a descriptive name and use that same name consistently, both in the code and in the test automation. Using the DSL during test automation makes the automated tests easier and more straightforward to understand for everyone in a team. Having a common language helps teammates communicate and collaborate better.

9.4 Maintain Standards

LO-9.6	K1	Recall the characteristics of a maintainable and healthy test

Test automation is a continuous process. Test automation works best when it starts with a strong foundation and builds over time. Below are three simple characteristics of healthy and high-quality tests in a test suite.

Valuable Tests

Automated tests must always bring value. Review the automated tests over time to determine if these tests are still needed or need improvement. Quality should prevail over quantity. Focus on what has value to automate rather than the number of tests. Treat test code like development and production code and give it the same level of nurturing and attention.

Reliable Tests

Tests must produce the same result every time. Failure will be inevitable, so it is essential to have a plan to deal with failures. Create independent tests (see the 9.2). Run tests in a special, unique environment so as not to interfere with others who use the same environment.

Fast Tests

Try to have the test running as fast as possible to have fast build times and eventually release development code faster. Running tests in parallel dramatically reduces the overall time required to run automated tests. Limit testing in the user interface to have more focus on the faster lower-level tests (see 2.2 The Test Pyramid).

10 How to Embed Test Automation

Test automation is a continuous process that should be adopted by the organization and embedded within its regular processes. This chapter contains some necessary activities to ensure durable test automation within an organization.

Keywords

Maintaining standards, Definition of Ready (DoR), Definition of Done (DoD), maintenance plan, code coverage, Succes sharing

LO-10.1	K1	Recall possible criteria of a Definition of Ready
LO-10.2	K1	Recall possible criteria of a Definition of Done
LO-10.3	K1	Recall aspects of test automation maintenance
LO-10.4	K1	Recall triggers for test automation in CI
LO-10.5	K1	Recall the benefits of measuring code coverage
LO-10.6	K2	Summarize potential benefits and achievements of test automation

10.1 Embed test automation

LO-10.1	K1	Recall possible criteria of a Definition of Ready
LO-10.2	K1	Recall possible criteria of a Definition of Done

When working in an agile team, test automation should be part of the way of working and be considered in the “Definition of Ready” (DoR) and the “Definition of Done” (DoD).

In the DoR, a team should consider each user story:

- What tests are needed?
- Should the user story be tested using automated tests?
- Are automation-supported tests needed?
- Are non-functional automated tests needed, like performance or security testing?
- Is a new automated test necessary? Or can an existing automated test be modified or extended?
- Is there a risk of failing existing tests because of implementing the new user story?
- What is needed to test the new user story using test automation? (e.g.: test data, stubs, drivers, hardware, accounts/credentials)

The DoD may also contain requirements on test automation. For example, a team can describe that a user story is Done when:

- The test code has been reviewed.
- The code has been committed to the repository.
- The automated test has been executed successfully twice (or execution is idempotent, so independent of the number of executions, the result is always the same).

The DoR and DoD described above are by no means definitive and exhaustive. Defining such criteria is a way of embedding test automation within an organization.

10.2 Make a Maintenance Plan

LO-10.3	K1	Recall aspects of test automation maintenance

Test automation is an ongoing process and test automation maintenance will be a big part of that process.

Maintaining test automation consists of:

- Adding new tests
- Updating old tests
- Fixing malfunctions

New features within a software development project require new tests to be added to the test automation. For those building the new functionality, discuss with the team how it can be tested and what types of tests will be automated. Both applications and testing can change and/or become obsolete over time. It is, therefore, necessary to keep all tests and test code up to date for smooth test automation. Sometimes tests require new test data or some other assertion to validate the results. Alternatively, a test can be deleted if this test is no longer relevant or if the tested feature has changed completely.

To decide whether the test that ended in error is caused by a defect, the error must first be further investigated. Tests fail for various reasons, including automation. Because the build must always be green when running test automation, there must be an action plan for tests that fail.

If there is a random defect, find a way to fix the defect, such as rerunning the failed tests. Tests that fail (seemingly) randomly are called flaky tests. A good practice is to isolate flaky tests from tests done consistently and fix the flaky tests.

A legitimate error may indicate a defect that needs to be fixed in the code. This would require rolling back the code that introduced the bug or fixing the bug. Tests need to be added, updated, and repaired regularly. Having a maintenance plan in place ensures these changes go smoothly, ensuring more robust test automation.

10.3 Use Continuous Integration

LO-10.4	K1	Recall triggers for test automation in CI

Test automation can be run over and over and produce the same result every time. Continuous integration helps run automated tests repeatedly across platforms and environments, including in parallel. Continuous integration can trigger tests due to changes pushed to a repository. Test automation can also run regularly, such as every hour or at night at a specific time. One advantage of test automation in CI is that defects are more likely to be discovered compared to tests run locally on an ad hoc basis.

10.4 Measure Code Coverage

LO-10.5	K1	Recall the benefits of measuring code coverage

Code coverage is one of the metrics to analyze test automation. It measures how much of the code is covered by (unit) tests. It typically follows the coverage of statements and functions. Other metrics are branch coverage (if-else, for- and while-loops, switch statements) and line coverage.

Code coverage helps visualize which parts of the application have been tested well and which have not. It also helps determine whether the risks within those parts of the application have been appropriately addressed within the test automation project.

Code coverage tools are usually free for open-source projects and easy to set up. They also provide command-line tools and reports to visualize code coverage. This report is different from a report with test automation test results, but both can be combined into a single report.

10.5 Share Successes

LO-10.6	K2	Summarize potential benefits and achievements of test automation

Share test automation stories with the rest of the organization or business.

- Discuss what problem test automation wants to solve.
- Emphasize the value of test automation. Tests can tell a compelling story about the status of an application.
- Share the benefits and the lessons learned with the rest of the organization. Share examples of things like how many defects test automation has caught, in comparison to manual testing.
- Explain the purpose of automation, which is to detect defects faster.
- Tell or show something about code coverage, This gives an impression of the value this provides.
- Share how many hours were saved by not testing a feature manually.

Graphics and images, in combination with demos and storytelling, help management understand the value of test automation in an Agile environment:

- Productivity increases because more checks are automated, and less time is spent on each release.
- Multi-functional team members work more closely together.
- Teams decide together where, for example, integration tests can be improved.
- Create insights about work processes within business and development.

Being able to share compelling test automation stories backed by data is therefore invaluable.

References

Reference	Source
BT1	A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives - By L. Anderson, P. W. Airasian, and D. R. Krathwohl (Allyn & Bacon 2001)
BT2	https://www.apu.edu/live_data/files/333/blooms_taxonomy_action_verbs.pdf
1 Expectations of Test Automation	
AT1	https://www.techopedia.com/definition/17785/automated-testing
TC1	James Bach & Michael Bolton, 'Testing and Checking Refined', https://www.satisfice.com/blog/archives/856
WT1	https://www.identify.nl/de-waarde-van-testautomatisering
2 The Value of Test Automation within Organizations	
TA1	https://owl.purdue.edu/owl/general_writing/academic_writing/historical_perspectives_on_argumentation/toulmin_argument.html
TH1	https://books.google.nl/books?hl=nl&lr=&id=Q1grDwAAQBAJ&oi=fnd&pg=PT6&dq=six+thinking+hats+of+de+bono&redir_esc=y#v=onepage&q=six%20thinking%20hats%20of%20de%20bono&f=false
2 Test Types	
AC1	https://www.thoughtworks.com/insights/blog/architecting-continuous-delivery
BM1	https://scholar.archive.org/work/pmyn7ajzkbbwhatpvcmcp4yimy/access/wayback/http://umj.metrology.kharkov.ua/article/download/193279/193570#page=40
GB1	https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html
JW1	https://www.james-willett.com/the-evolution-of-the-testing-pyramid
MB1	https://en.wikipedia.org/wiki/Model-based_testing
MC1	Mike Cohn, 'Succeeding with Agile', Addison-Wesley Educational Publishers Inc, 2009.
3 Test Automation Frameworks	
AU1	https://www.agile-united.com/au-courses
DD1	https://testsigma.com/blog/data-driven-vs-keyword-driven-frameworks-for-test-automation
HT1	https://www.brainbox.consulting/blogs-news/software-testing-blog/hybrid-test-automation-framework
4 Test Development Approach	
AU1	https://www.agile-united.com/au-courses
5 Complementary Tools and Components	
AS1	https://www.gartner.com/reviews/market/application-security-testing
FT1	https://www.softwaretestinghelp.com/functional-testing-vs-non-functional-testing/
IO1	https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/00-Introduction_and_Objectives/README

MT1	https://www.ministryoftesting.com/articles/8e1ecbd6
PC1	https://spring-petclinic.github.io/
VS1	https://owasp.org/www-community/Vulnerability_Scanning_Tools
OW1	https://owasp.org/www-project-top-ten/
7 Tool Selection	
SA1	https://www.scaledagileframework.com/spikes/
9 Good Test Automation Practices	