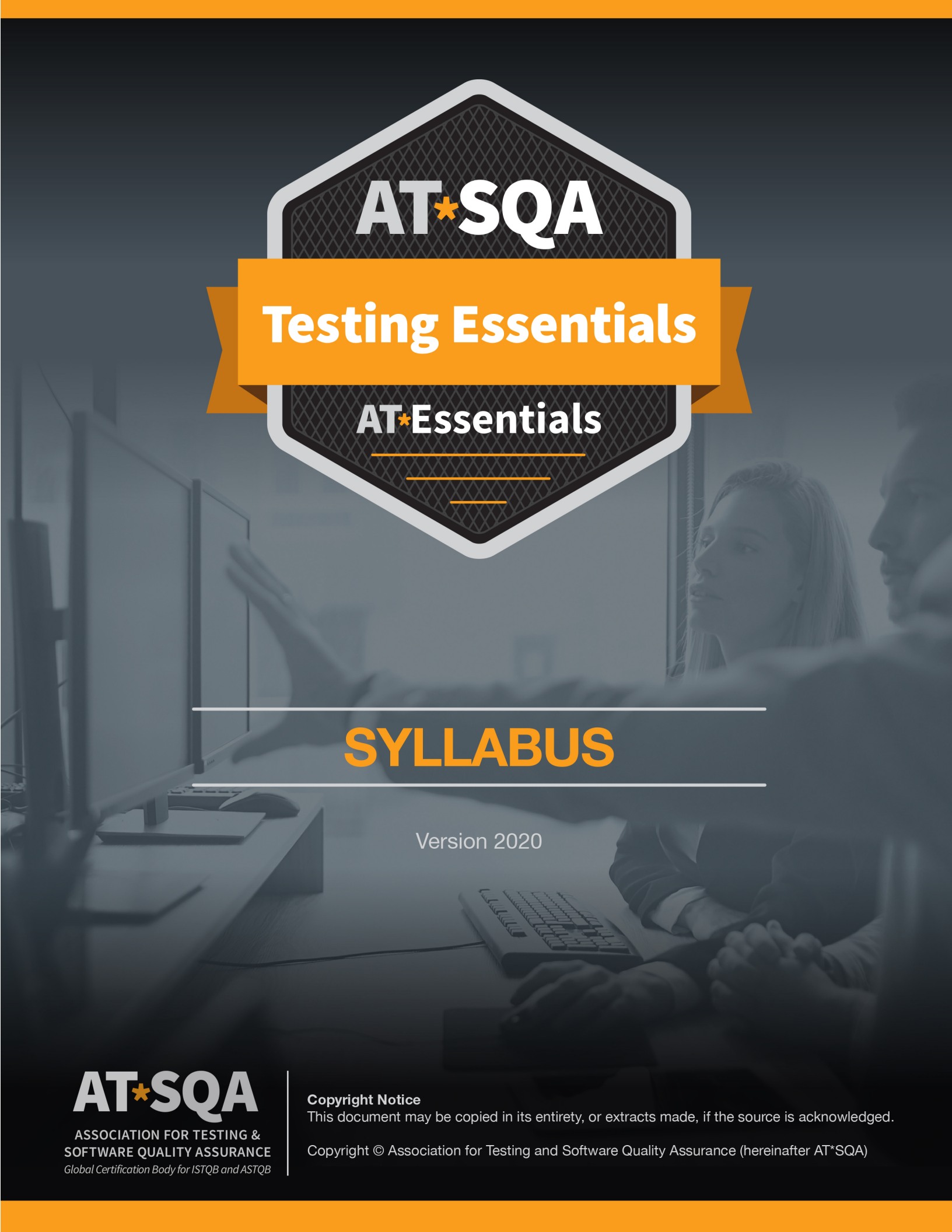

SYLLABUS

Version 2020

AT*SQA
ASSOCIATION FOR TESTING & SOFTWARE QUALITY ASSURANCE
Global Certification Body for ISTQB and ASTQB

Table of Contents

0. Introduction to this Syllabus

0.1. Purpose of this Document

This syllabus forms the basis of the AT*SQA certification for Testing Essentials. AT*SQA is an International Standards Organization (ISO) compliant certification body for software testers. AT*SQA provides this syllabus as follows:

1. To training providers - to produce courseware and determine appropriate teaching methods.
2. To certification candidates - to prepare for the exam (as part of a training course or independently).
3. To the international software and systems engineering community - to advance the profession of software and systems testing and as a basis for books and articles.

AT*SQA may allow other entities to use this syllabus for other purposes, provided they seek and obtain prior written permission.

0.2 What is Essential?

The Information Technology (IT) world changes almost continuously as new technologies and techniques are adopted. Software testers (whether by title or in practice) must adapt quickly and be able to leverage their skills to meet new challenges. However, the essential skills and knowledge remain the same, serving as core understanding to which new information can be added. For the sake of readability, the term “software tester” will be used to refer to anyone who is testing software, regardless of their formal role.

This syllabus focuses on the essential areas of software testing that are required, regardless of the technology, lifecycle or tools in use. Some projects may use more or less of these skill areas, but all software testers need to understand and master this core skill set.

As the name indicates, this syllabus covers the “essentials”. This syllabus should be considered a springboard for additional certifications and knowledge areas. As a part of AT*SQA’s ISO compliant offerings, the certification must be kept current with additional learning completed within the defined timespan. For more details, see AT*SQA’s website. This helps software testers to continue to expand their knowledge and marketability and acknowledges the very real need for continuing education in the software testing industry.

0.3 Syllabus Structure

This syllabus has been constructed to be tool and methodology agnostic. In places where different approaches are needed based on different lifecycles, those areas are highlighted with appropriate recommendations for tailoring the approach.

The intended target audience for this syllabus is anyone conducting software testing, whether or not they have the title of software tester. This includes Scrum team members, developers, Business Analysts (BAs), software specialists and anyone interested in learning the important aspects of software testing.

This syllabus is intended to be read in full, but if the reader is interested only in a specific area, each area can be read independently. It is recommended that the Test Approach and Testing Techniques sections (Sections 2 and 3, respectively) are considered compulsory reading, as these are generally applicable to any of the specialist areas of testing and provide a good background to general testing practices.

0.4 Examinable Learning Objectives

Each chapter notes the time that should be invested in learning and practicing the concepts discussed in that chapter. This information should be used as a guideline when creating training materials or for an individual conducting self-study.

All identified key terms are examinable, either individually or by use within an exam question. Full definitions for the key terms can be found in the AT*SQA glossary (see www.atsqa.org).

The Learning Objectives for each chapter are shown at the beginning of the chapter and are used to create the examination for achieving the Testing Essentials Certification. Learning objectives are allocated to a Cognitive level of knowledge (K-Level). A K-level, or Cognitive level, is used to classify learning objectives according to the revised taxonomy from Bloom [Anderson00]. AT*SQA uses this taxonomy to design all examinations.

This syllabus considers four different K-levels (K1 to K4) as noted for each Learning Objective (LO):

K-Level	Keyword	Description
1	Remember	The candidate should remember or recognize a term or a concept.
2	Understand	The candidate should select an explanation for a statement related to the question topic.
3	Apply	The candidate should select the correct application of a concept or technique and apply it to a given context.

4	Analyze	The candidate can separate information related to a procedure or technique into its constituent parts for better understanding and can distinguish between facts and inferences.

In general, all parts of this syllabus are examinable at a K1 level. That is, the candidate will recognize, remember and recall a term or concept. Other specific learning objectives are shown at the beginning of the pertinent chapter.

1. Introduction to Software Testing – 60 mins.

Keywords

requirements, test case, test condition, test plan, test strategy

Learning Objectives for Introduction to Software Testing

1.1 What is Software Testing

LO-1.1.a (K2) Summarize the various forms of requirements
LO-1.1.b (K1) Recall the meaning of “fit for purpose”

1.2 A Brief History

LO-1.2.a (K1) Recall the difference between a test engineer and a test analyst

1.3 Structured Testing

LO-1.3.a (K2) Explain the purpose of the documents used in a structured testing environment

1.4 The Role of a Tester

LO-1.4.a (K1) Recall who can be a software tester

1.1. What is Software Testing

Software testing has variable meanings. The term has evolved as new software development lifecycle (SDLC) models have been introduced. Regardless of the changes to the exact definition, software testing is an activity, or set of activities, that are conducted to evaluate software to determine the following:

- Have the requirements been met?
- Is the software “fit for purpose”?
- Has the risk been reduced enough?
- Have important defects been identified and addressed?

Each of these questions tends to elicit more questions.

1.1.1. Requirements

Software requirements come in many forms including:

- Formal requirements documents prepared by Business Analysts (BAs)
- Technical requirements documents, such as functional specifications, design documents, and interface design documents

- Higher level documents, such as use cases which describe how an expected user would accomplish tasks or goals by using the software
- SDLC unique documents, such as user stories in the Agile lifecycle model
- Very informal diagrams on white boards and results from workshops
- Word-of-mouth and drawings in a highly collaborative environment (where the team is all working together, all the time)

The ability to verify that the software meets the requirements is dependent on the clarity of the requirements. If a requirement is clear and defines exactly what the software is supposed to do, the verification is straightforward. Where the requirements are vague or missing, the tester must be able to apply their own knowledge of the users and domain in order to determine if the requirements have been met.

1.1.2. Fit for Purpose

All software is designed to fulfill a purpose, but just accomplishing a task is not enough. In order for it to be "fit for purpose", the software must work for the people who will be using it, in the environment in which they will be using it. For example, a mobile application that allows people to deposit checks by taking a picture of the check may work great in the lab with specific lighting and backgrounds, but may fail when used in a user's home. In this case, the requirement may be met (it works functionally), but it is not "fit for purpose" because it is not usable in the target environment.

1.1.3. Risk

Because there is rarely enough time to perform all the testing possible, risk prioritization is used to limit the testing to what is needed to mitigate risk to an acceptable level. Determining what is acceptable may be a matter of opinion, which is why risk analysis requires cross-functional input to ensure each risk is being considered and rated accurately. With the above example of the check deposit, if the decision is that a low-lighting environment is highly unlikely, that would reduce the rating of that risk. On the other hand, if it is determined that this is highly likely to occur and that the user will be unable to deposit their check, the risk would be considered as very high and additional work would be required to adequately mitigate that risk. Risk is discussed further in Section 2.6.

1.1.4. Finding Defects

One of the purposes of testing is to find and fix defects before the software is released to the users. Defects, also called bugs, are flaws in the software that cause it to function incorrectly or cause the user to use it incorrectly. Clear requirements help in determining what is a defect and what is not. The less clear the requirements, the more discussion will be needed to determine if an anomaly is actually a defect or if it is just an undocumented feature of the software. Keeping the user's view in mind when testing the software helps the tester to better determine what a user would consider to be a defect. For example, an incorrect text prompt "enter suer name" is clearly a defect. What if the user name always has to be between 5-15 characters but the user is not told that? Is that a defect? Defect identification and proper recording is

an important task for a tester. Defects that are not recorded accurately are difficult, if not impossible, to fix.

1.2. A Brief History

Software testing has existed for as long as there has been software. The formality, emphasis, funding and respect for software testing has varied over the years, but it will always be needed. Good practices that were popular in the 1970's still have merit today, just as new practices developed since that time also have merit. It is important to remember that there is a wealth of knowledge in software testing. Environments, languages, devices and approaches may vary, but understanding the essentials of software testing will allow the tester to work in, and adapt to, any environment.

In software testing, there tends to be a differentiation between technical testers (i.e., test engineers) and non-technical testers (i.e., test analysts). Technical testers are expected to have the skills such as those needed to write test automation, conduct performance tests or participate in code/design reviews. Test analysts are generally expected to conduct the functional testing (i.e., does the software meet the requirements), as well as to consider usability (i.e., will the target user be able to use the software effectively, efficiently, and enjoy using it) and domain/environment attributes of the software. In some cases, test analysts are also expected to work with end-users for user acceptance testing (UAT) and to help validate that the software will work in the target environment for target users who are accomplishing the target tasks.

Like software development, software testing will continue to evolve. Mastering the essentials of software testing will help make a tester resilient and able to adapt to changes.

1.3. Structured Testing

Highly-structured testing, such as that required by some sequential lifecycle models (discussed in Section 2.3), generally has a higher level of documentation. Formal test strategies, well-defined test plans, explicit test cases, controlled test data and test environments, and a well-managed defect lifecycle are all artifacts of a highly-structured approach to testing.

While the documents may vary depending on the environment, the following are normally found in a structured testing environment:

- Test strategy – a test strategy is an organization-wide document that defines how testing will be conducted across all comparable types of projects in the organization.
- Test plan – a test plan is the implementation of the test strategy for a particular project and includes the approach to be used for testing, a definition of the scope of testing for the project, the testing schedule, the resource requirements, a description of tools and their usage, a definition of

environments and any other information required to describe the testing process, and stakeholder agreement for a project.

- Test conditions – a test condition is a capability or characteristic of the software that needs to be tested. This could be something functional, such as the ability to enter a user name; or something non-functional, such as the expected response time of the application under a defined load.
- Test case – a test case is the information required for a tester to test a test condition. This can include the pre-conditions of the system (e.g., user does not exist), the post-conditions after the test (e.g., the user has been created) and the inputs and actions required to accomplish the goal of the test.
- Defect reports – each defect should be captured in a report that is then processed through a workflow to record all the actions taken to resolve the issue. A defect report normally records information, such as the environment used, steps to reproduce, priority/severity, expected/actual results and other descriptive information.

More information about the documentation used in testing can be found in Section 2.5. Depending on the environment, more or less of these documents will be prepared and maintained as part of the testing process.

1.4. The Role of a Tester

The role of a "tester" can vary with different organizations and different lifecycle models. While software testing is a profession, others may periodically carry the title of a software tester. For example, in an Agile lifecycle model, everyone on the team has testing responsibilities and may be considered to be a tester. Business users may become testers during UAT. Software developers are testers when they are testing their own or another developer's code.

Regardless of the name of the role, testers are responsible for gathering information that can be used to assess the quality of the software. This information includes tests that have been run and have met their goals (passed), tests that have not met their goals (failed), defects found, risks mitigated, test coverage (in terms of tests executed vs. not executed, code covered vs. not covered, risks mitigated vs. not mitigated, or requirements tested vs. not tested) and other information needed by the stakeholders.

All testers need to be familiar with the essential areas of software testing. Specialization in these areas may require further study, but a general familiarity is necessary to understand what can and should be tested for any software product.

2. Test Approaches – 145 mins.

Keywords

acceptance testing, Agile, alpha testing, beta testing, configuration management system, debugger, drivers, end-to-end testing, integration testing, interoperability testing, iterative model, Kanban, operational acceptance testing, product owner, requirements traceability matrix, risk, risk-based testing, Scrum, Scrum Master, sequential model, software development lifecycle, sprint, stubs, system integration testing, system testing, test levels, unit testing, use cases, user acceptance testing, user stories, V-model, waterfall

Learning Objectives for Test Approach

2.1 Introduction

LO-2.1.a (K1) Recall factors to consider when selecting a test approach

2.2 Testing Levels

LO-2.2.a (K1) Recall the purpose of system integration tests

LO-2.2.b (K2) Summarize the activities that take place during each of the four levels of testing

2.3 Software Development Lifecycle

LO-2.3.a (K2) Compare the advantages and disadvantages of following either a sequential or iterative lifecycle

2.4 Product Type

LO-2.4.a (K2) Describe how different product types affect the test approach to be used

2.5 Documentation Requirements and Availability

LO-2.5.a (K2) Describe how different documentation requirements can drive the selection of a test approach

2.6 Risk

LO-2.6.a (K2) Explain how risk affects the choice of a test approach

2.7 Schedule and Budget

LO-2.7.a (K2) Describe how a project's schedule and budget requirements affect the selection of a test approach

2.8 Maturity and Ability of the Team

LO-2.8.a (K1) Recall how the attributes of the team members can affect the choice of test approach

2.1. Introduction

A test approach defines how the testing for a project will be accomplished. The approach may be formally defined in the test plan or may be informally agreed upon by the project team. Approaches can include methods for prioritization (e.g., risk-based) or may specify that certain requirements be met (e.g., regulatory or certification requirements). Test approaches generally reflect the organization's test strategy and are used to ensure that the methods and goals of testing are aligned with the goals of the project team and the stakeholders.

Selecting the proper test approach for a project depends on a number of factors, including:

- Testing levels
- Software development lifecycle
- Product type
- Documentation requirements and availability
- Risk
- Schedule and budget
- Maturity and ability of the team

All of these factors must be considered when determining the best test approach for any project. Realistically, any one of these individual factors can skew the decision. For example, if the project is a safety-critical project requiring approval by a regulatory commission of some type, then documentation requirements and risk management will become the most important factors in the test approach decision.

This section explores each of these factors and how they help to determine the optimal test approach for a project.

2.2. Testing Levels

Regardless of how the software is developed and which lifecycle model is followed, there are four distinct levels of testing. These levels may be combined in some cases, but it is important to follow the level approach to improve the efficiency of testing and reduce the time required for troubleshooting and testing for possible regressions (i.e., regression testing). Adequate testing at each level is more efficient than a big bang approach in which testing is only done once at the end of development.

While testing is generally assigned to particular team members, such as developers or testers, testing can also be shared across team members, with the most suitable team member doing the testing at a given point in time. The following list of levels is a

categorization of the types of testing that need to occur and the logical progression of testing:

- Unit testing
- Integration testing
- System (end-to-end) testing
- Acceptance testing

In some cases, system integration testing may also be required. This happens when multiple systems - that are comprised of complete sets of software that provide functionality independently - must also interface with each other. In this case, testing is needed to ensure that the independent systems integrate properly. This type of testing usually occurs after system testing is completed on each of the independent systems.

2.2.1. Unit Testing

Developers conduct unit testing to ensure that their units (or modules) of code are working according to their requirements and design. Each unit, or set of testable units, is tested either in an automated fashion using a static analysis tool, using a unit test framework such as JUnit, or manually using a debugger to step through a particular test case. The purpose of unit testing is to ensure that the individual units of code function as intended. Performance testing and cybersecurity testing of individual, relevant units may also be conducted during unit testing. Unit testing generally applies structure-based (white-box) testing.

Test-driven development (TDD, also sometimes called test-first development) is a form of unit testing where the test is written before the actual code is written. In this case, the automated test will execute and fail, until the entire testable unit is developed. When the entire unit is available and free of detected defects, the test code will pass. TDD was introduced in Extreme Programming (XP) and is commonly used in Agile environments. It can also be used to develop unit test cases when using other development methodologies such as sequential or incremental, as well as in environments where safety-critical code is being produced and must always adhere to the highest quality standards.

2.2.2. Integration Testing

Developers and/or testers conduct integration testing to ensure that the tested units work together. Integration testing focuses on the communication between units at the points of interaction. For units that are not ready to be integrated yet, drivers and stubs may be used as placeholders. Drivers are used to call the testable modules or units of code. Stubs are used to act like a module or unit of code and generally return a positive response. On a larger scale, service virtualization (SV) can be used to simulate entire services or parts thereof. SV is commonly used when services needed for integration are not yet available or cannot be tested (such as a banking backend interface).

Integration testing can be done in a top down fashion (where the drivers are written first and can be used to call the units as they become ready for testing), or a bottom up fashion (where the individual units are written and tested via a driver that is written specifically for testing purposes). The term continuous integration is used to define a configuration management system that has test automation built in. When a new unit is checked in, it can be exercised via test automation with other units that have also been checked in. Continuous integration is often used after a significant set of code has been developed to avoid spending too much time developing drivers and stubs to simulate code that has not yet been integrated.

Integration testing is primarily functional, but can also include performance testing and cybersecurity testing of the integrated part of the system. Integration testing is often informal, with little documentation or formal test scripting.

2.2.3. System Testing

System testing, or end-to-end testing, is conducted to verify that the software as a whole is working per the defined requirements (specifications, user stories, design documents). Testers or quality assurance (QA) analysts usually conduct this testing in an environment that is configured similarly to the production environment and uses data similar to what would be found in the production system. The primary goal of this testing is to ensure that the stated requirements have been met and test coverage is often tracked with a requirements traceability matrix (RTM). Test management and/or requirements management tools are often used to store test cases, record test execution and to create the traceability matrix - mapping requirements to test cases. Documented test cases may be used to guide the testing, although lighter methods such as checklist-guided or exploratory testing may also be used.

System testing is primarily functional, but should also include performance testing, cybersecurity testing, interoperability testing and usability testing. Depending on the product being tested, system testing may be extended to cover all components of the system, including software, hardware, data, and procedures. In some cases, system testing is the first opportunity to conduct these other types of testing in a realistic environment.

End-to-end testing is a type of system testing that exercises transactional flows through an entire system or set of systems. This testing often simulates real world usage and is guided by process flows and use cases.

2.2.4. Acceptance Testing

The goal of acceptance testing is for the targeted user or operator to “accept” the software as working to meet their requirements for the software. Different types of users can conduct acceptance testing in different environments. The following is a list of the most common types of acceptance testing:

- User Acceptance Testing (UAT) – testing conducted by system users or proxies (e.g., business analysts) for those users in order to determine if the software is fit for purpose. This is normally performed using documented test cases and

exploratory testing. The users of the system exercise the system as they would during normal daily use, including cyclical functions such as end of month and end of year. Business analysts will sometimes guide this testing for the users. The goal is for the users to "accept" the system based on their evaluation of whether the acceptance criteria have been met. The users bring a unique viewpoint that may be missed by testers who are unfamiliar with all aspects of system usage and the real-world conditions that users encounter.

- Operational Acceptance Testing (OAT) – testing conducted by system operators to determine if the software will work in the production environment when fully deployed. Ideally, this testing is conducted in a staging environment that is an exact replica of production; where that is not possible, the environment should be as close as possible to production. System operators use this testing to ensure that the software will work properly with load balancers, firewalls and other production equipment, as well as with production processes such as backups. Testing rollout and rollback plans are often part of this as well.
- Alpha Testing – testing conducted at the development site, but not by the developers or testers who have been working on the project. This testing is sometimes called "internal acceptance testing", meaning that the testing is conducted within the organization, but is not exposed to external users. Training groups and support groups within the organization are often used for this type of testing.
- Beta Testing – testing conducted at a customer (or potential customer) site using the customer's data and network environment. This testing is usually conducted by the customer themselves, although they may have some assistance from the testers or developers to ensure the test coverage is adequate. The goal of this testing is to determine if the software is fit for purpose in the real production environment without fully releasing it to everyone. Feedback from beta testing may result in further internal development and/or testing prior to the full production release.

2.3. Software Development Lifecycles

2.3.1. Sequential Models

Sequential lifecycle models include waterfall and V-model. These are considered to be sequential models because the steps of the development process are sequential: requirements, design, code, test, and release. Sequential models require a fully developed set of requirements before design and coding starts. It should be noted, however, that in some versions of the V-model, verification occurs at each major phase. For example, requirements reviews may be performed during requirements development. Unit testing is usually conducted as the software is being developed. Integration testing, system testing and acceptance testing usually occur after development is completed.

In a pure waterfall model, testers usually are not engaged in the SDLC until the software is completely built and the developers have completed their unit testing. In a

pure V-model, testers are engaged early to review requirements, design documents and to prepare the testware (e.g., test plan, test cases) prior to receiving the code to test.

The advantages of sequential models include:

- The requirements are considered to be stable throughout the project
- Test automation can start at the beginning of testing because the software will not change
- In a waterfall model, the test team is only involved from the moment the code is complete, freeing them for other tasks or other projects
- In the V-model, the test team is involved with reviewing all the documents produced by the business analysts and developers (requirements, high-level and low-level design documents) and can provide input on each of these, thus engaging with the project sooner and having input that can influence the quality of the product
- In the V-model, there is generally more time available to apply structured testing (e.g., prepare the test documentation, including test cases)

The disadvantages of these models include the following:

- Because no code is seen by the testers until all the code is developed, there is little opportunity to influence the usability and user experience
- The testers need time to prepare the test documentation (e.g., test cases) after they have received the code and before they can start testing
- The users' requirements may change while development is occurring, resulting in a product being created that is no longer wanted
- If the development time takes longer than expected and release dates are not moved, the time for testing is compressed

The sequential lifecycle models, in particular the V-model, are still used in the industry and are successful in the proper environments. These models are particularly common where thorough documentation is required (e.g., for safety-critical projects) and where the requirements are not likely to change over the life of the project.

2.3.2. Iterative Models

Iterative models include basic iterative and Agile. Agile is usually implemented via one of the common process frameworks, such as Scrum or Kanban. Iterative development simply means that the software is developed in small sets, with each iteration producing a piece of software functionality. Iterations vary from 2 – 4 weeks and each iteration includes analysis, design, implementation and testing.

In an Agile project using the Scrum framework, iterations are called sprints. Each sprint has a planning session which is used to determine which user stories (i.e., small bits of requirements) will be implemented during the sprint. The self-organizing cross-functional team determines what they can commit to completing within the sprint. A Scrum Master provides guidance and coaching for the team and the product owner, represents the business, and defines and refines the requirements.

In a Kanban project, the emphasis is on continual delivery and managing the workflow to eliminate bottlenecks in the process. While not strictly an Agile framework, it is frequently used in Agile environments to manage the workflow by use of tools such as Kanban boards.

The advantages of the iterative models include the following:

- The team is able to react quickly to changing requirements
- A demonstrable product, or piece thereof, is available for the customer to see and use
- Early feedback can change the direction of the team and the product to better suit the needs of the customer
- Schedule constraints are handled by implementing less functionality
- Testers are more engaged in the overall process and tend to form closer relationships with the developers

The disadvantages of the iterative models include the following:

- Too frequent changes in direction can result in little or no progress
- Lack of detailed documentation means reliance on communication, which may be difficult for a team that is dispersed
- If cultural issues in an organization are significant, people may not be able to work effectively as a cross-functional team
- Lack of detailed documentation may make the model infeasible for some products, particularly those with regulatory or safety-critical aspects
- Test automation is mandatory to avoid manual testing time becoming increasingly long due to the larger scope of regression testing as iterations progress
- Rigid adherence to the process can result in a significant learning curve for a team

Iterative models have been around for many years, long before Agile was defined. These models have worked successfully across a wide variety of software projects and continue to be the most dominant models in the industry.

2.3.3. Hybrid Models

While there are defined software development lifecycle (SDLC) models, it is important to remember that most organizations do not follow a “pure” model. Most organizations follow hybrid models that take bits and pieces from various models to create a best-fit model for the organization. Sometimes this is done wisely, picking the most efficient and practical model; but more often than not, this is done without considering what is being left out. That is where the danger lies.

SDLCs have built-in safeguards to ensure that necessary steps are completed. Picking and choosing the “best” parts from different SDLCs is likely to result in weaknesses being exposed. For example, if an organization were to pick an Agile SDLC, but also chooses to work without defining acceptance criteria for stories, there

is a gap created in the validation aspects of the project. Similarly, if an organization were to pick a waterfall model, but decides to use stories to document the requirements, the concept of completed and well-defined requirements before the start of coding is violated. This may result in a product that is incompletely developed or in a product that necessarily must change as development progresses. This results in a longer development time and a compression of the testing schedule.

When selecting a model, it is important to understand the project, the team, the product and the goals of the organization in order to select the best fit. More information on software lifecycle models can be found in ISO/IEC/IEEE 12207:2017 and ISO/IEC/IEEE 15288.

2.4. Product Type

Another consideration when determining the testing approach is the type of product that is being developed. A mobile application that is expected to last for six months requires a different approach than software that will control the navigation of a fleet of aircraft. In general, the longer the software will stay in production and the more critical the functionality of the software, the more formal the approach. A formal approach may dictate the lifecycle model, the level of documentation required and the test techniques to be used. Similarly, a short-lived application that is used to provide a game interface for idle travelers may be best served by a lightweight approach with an Agile lifecycle, minimal documentation and only exploratory or acceptance testing.

When deciding how the product type may influence the test approach, the following factors should be considered:

- Length of time the software will be used in production before replacement
- Any safety-critical aspects of the software
- Any regulatory requirements that must be met
- Competition and market opportunities (e.g., bigger feature set, better usability)
- Requirements for security and performance
- The testers' understanding of the product and domain, and the degree of changes to either the product or related items

The best test approach for a product is somewhere on the spectrum from formal and fully documented to informal and lightweight.

2.5. Documentation Requirements and Availability

In software testing, documentation has two general categories: documents required to properly test the product, and documents required to demonstrate the testing has been completed. The test approach is heavily influenced by the availability of documentation and the requirement to provide documentation from the test process. If there is little or no documentation regarding what the software is supposed to do or how it will do it, the tester is forced into an approach that includes some amount of exploring the software to understand what it is doing. Creating detailed test cases may

not be worthwhile since the testing will be occurring while the research is being conducted to document the test cases.

On the other hand, if test case documentation and test execution evidence is required by the project, then that documentation will have to be created, maintained and updated as needed. If there are plans to keep a product in production for several years and updates are likely, there is a greater need for reusable test artifacts, particularly test cases. The requirements for a test management system are influenced by the need to track documentation and test evidence during testing.

The types of documentation that can be used as input for the testing effort include:

- Requirements documents
- Specifications (e.g., technical/architectural, user and database specifications)
- User stories
- Business cases
- Use cases
- Design documents
- Screen mockups and wireframes
- Sample reports
- Existing test cases
- Checklists
- Defect reports
- Requirements traceability matrix
- Existing user and operational guides

The types of documentation that can be provided as evidence of test execution include the following:

- Test cases with pass/fail recorded at the test case and step level
- Screen shots
- Defect reports
- Coverage reports
- Test automation logs and reports

Projects have differing documentation requirements. It is important to select a test management and document management system that will help to track, version and report the documentation that is needed across the variety of projects, that will also be supported by the tools. It is important to remember that documentation has no value unless someone will use it. It is good practice to always aim for the lightest suitable documentation for a project; consider re-use, consider true needs and consider other ways of communication to ensure that the documents produced meet the needs of the project without burdening the team.

2.6. Risk

In testing, risk is defined as an event or condition that could occur and would result in a negative outcome. If the event or condition actually occurs, it is called an issue [PMBOK]. While a risk has somewhere between a 1% and 99% probability of occurring, an issue has 100% probability of occurring because it has actually occurred.

Risk is a significant factor in determining the best test approach. Higher risk projects generally require more formal approaches with more complete documentation. Lower risk projects can work with a lighter approach and may require little or no documentation. Using risk prioritization, commonly called risk-based testing, on every project is a strong approach and helps to prioritize and define all testing activities, including the following:

- Formality of the test approach
- Test case preparation and documentation level
- Test execution
- Defect prioritization
- Defect re-testing
- Regression testing
- Timing of other testing such as security and performance
- Test automation requirements
- Depth and breadth of testing

Identifying risk is best done with a cross-functional group who can clearly review the project, its intentions, and identify the risks that are inherent in the project and the software being developed. Once the risks have been identified, they can each be assessed in terms of likelihood of occurrence and impact to the customer or system if they occur. The resulting intersection of likelihood and impact is often expressed as the risk level. For example, high likelihood and high impact would result in a risk level of “High” or “Very High”. Another way to express the risk level is by numeric scores on a scale of 1 – 10. This assessment helps to indicate the mitigation required and can guide the types, extent and priorities of testing. User training may be used to mitigate some risks whereas other risks may require extensive testing in a production-like environment.

Assessing and ranking all identified risks allows the team to determine the best approaches for mitigation and also helps to set the testing schedule. For example, a high likelihood and high impact risk that can be best mitigated by testing will usually require more time in the testing schedule than a risk with low likelihood and low impact. It should be noted that there is a degree of error in assessing risk as it is essentially a qualitative exercise. Contingency plans are helpful when low risks may become high risks.

2.7. Schedule and Budget

Any testing approach must consider the schedule and budget for the project. It is unusual to find a project for which there is not a pre-defined schedule and/or budget. When the test approach can influence the establishment of a project's schedule and budget, adequate time and resources should be allocated for testing. More commonly, the schedule and budget are already set before the testing approach is considered. In this case, the test approach changes from "what should we do" to "what can we do". When defining the best test approach for a project scenario, it is good practice to start with the "should" and then factor that down to fit the schedule/budget.

When schedule is tight, risk-based testing is the most solid approach. It allows testing to be prioritized to mitigate the most important risks first. With a constrained schedule, this will help provide visibility to the project team regarding the risk that has been mitigated and the risk that is still outstanding. Because tight schedules often result in insufficient testing, it is important that the project team understands and accepts that there is significant residual risk. Risk-based testing can be conducted within any lifecycle. It is a method of test organization that addresses testing in a risk-based order, within the overall project or within an individual iteration.

When the budget is tight, testing often suffers from a lack of time and resources. It is important to understand the constraints that will be placed on the testing as early as possible. For example, a constrained budget may mean that there will be no dedicated test environments. This may force the testing effort to share the same environment as the development effort, potentially resulting in inefficiencies and re-testing. This quickly becomes a schedule issue as well. Insufficient tester resources and inadequate tools may also be evident when the budget is constrained.

Any possible issues of this type must be anticipated in the test approach. If testing and development will be forced to work in the same environments, using an iterative approach is logical because of the close interaction. Pushing more testing earlier (i.e., "shift left") is another way to combat tight schedules and budgets. This allows testing to happen sooner and for quality issues to be addressed more quickly. Testing will always be faster and less expensive when the product being tested is of a higher quality.

2.8. Maturity and Ability of the Team

One last factor to consider when determining the test approach is the maturity of the team as well as the team's ability. A mature team who has worked together before and has a high level of skill and product knowledge may work better with less documentation and communication than a team that is new or distributed. Documentation is a way of communicating and bridging time zone issues. Less documentation means more verbal communication is required. A team that is comfortable with web meetings and video conferencing may be more effective with

less documentation than a team that prefers emails and documents to convey information.

Projects that include multiple teams will require more coordination and timely communication to avoid creating bottlenecks and frustration. Teams that have some outsourced aspects may require more formalized communication and documentation due to contractual requirements.

A well-skilled, mature team can make any testing approach work. The challenges often arise in a team with variable or minimal skills and a distributed environment where people cannot easily talk with each other. It is important to consider the approach that will work best for both the product and the people.

3. Testing Techniques – 215 mins

Keywords

Application Programming Interface (API), boundary value analysis (BVA), classification trees, combinatorial testing, decision table, equivalence partitioning (EP), exploratory testing, orthogonal arrays, pairwise testing, session-based testing, test charters, tester

3.1 Introduction

None

3.2 Partitions and Boundaries

LO-3.2.a (K3) For a given set of requirements, create a series of test cases using a combination of equivalence partitioning and boundary value analysis (BVA) testing techniques

LO-3.2.b (K1) Recall the types of defects that are likely to be found using equivalence partitioning and BVA

3.3 Decision Tables

LO-3.3.a (K3) For a given set of requirements, apply the decision table testing technique

LO-3.3.b (K1) Recall the types of defects that are likely to be found using decision table testing

3.4 Combinatorial

LO-3.4.a (K2) Describe combinatorial testing and the tools and techniques that are used

LO-3.4.b (K1) Recall the types of defects that are likely to be found using combinatorial testing

3.5 Exploratory

LO-3.5.a (K2) Explain the concept and application of exploratory testing

LO-3.5.b (K1) Recall the types of defects that are likely to be found using exploratory testing

3.6 API Testing

LO-3.6.a (K2) Describe the purpose and application of API testing

LO-3.6.b (K1) Recall the types of defects that are likely to be found using API testing

3.7 Picking the Best Technique

LO-3.7.a (K1) Recall how to select testing techniques for a given project

3.1. Introduction

Test techniques are procedures that are used to identify and select test conditions that can be targeted by tests. Test techniques can be applied at any stage in the development of software. The earlier testing starts (i.e., the shift left), the more effective and efficient the testing is. In the world of rapid lifecycles and continuous integration and deployment, testing is a critical task that must be executed to ensure that quality is being built into a product. Testing tasks are often shared by developers and testers, particularly in mixed skill teams commonly seen in Agile SDLCs. In this document, the term “tester” means the person designing and executing the tests as an activity, not necessarily a specific role with a “tester” title. For example, a business user may be the tester in the UAT activity, but they would not be a full-time tester.

It can be argued that exploratory testing and API testing are actually test types or even test approaches rather than test techniques. Generally, these two types are lumped together with the test techniques, so this chapter follows that same approach.

This section explores six common testing techniques that are applicable across a wide range of software. For the sake of this chapter, these are grouped together. Each of these has particular targets for the testing and is suited to finding particular types of defects. No one technique is suitable or effective in all situations and each technique has a target coverage. Often, a combination of techniques is used to provide the most efficient coverage.

3.2 Partitions and Boundaries

3.2.1 Equivalence Partitioning

Equivalence Partitioning (EP) is used to reduce the number of specific tests while still assuring broad coverage. EP is usually focused on determining a set of input values to use during testing, although it can also be used to categorize output values, processing variables or even environments. To apply EP, the set of possible values is divided into partitions (or equivalence classes) in which all values in the partition will be handled the same way by the software (e.g., positive values will be processed, negative values will cause errors).

Partitions can be considered “valid” or “invalid”. All the values in a valid partition should be accepted or processed by the software with no errors. All the values in an invalid partition should be handled as errors. For example, if the valid partition is for all triangles, then everything that is not a triangle is in the invalid partition. For input values, if the valid values are from 1-100, then anything below 1 would be in an invalid partition for values that are too low; and anything above 100 would be in an invalid partition for values that are too high.

Once the partitions are established, one value is selected from each defined partition (valid and invalid) and how that value is handled is assumed to be representative of how all the values in that partition would be handled.

Application

This technique is best applied when there are known sets of values that will receive the same processing. It is commonly used for input values where the set (or partition) of values can be determined. A risk with this technique occurs when partitions are established with values that actually receive different processing. It is important to have good information regarding how the software works when picking the proper partitions.

Types of defects

Defects found by this technique are usually functional in nature and deal with incorrect handling of various sets of data (e.g., no error handling for negative values).

Coverage

Coverage is determined by dividing the number of partitions for which a value has been tested by the total number of partitions identified. For example, if there are ten sets of values for which processing is different, and at least one value from each of five partitions has been tested, then 50% coverage has been achieved with this technique.

3.2.2. Boundary Value Analysis

Boundary Value Analysis (BVA) is an extension of EP and concentrates on testing the values that fall on or near the boundaries of partitions. BVA requires ordered partitions (i.e., ranges of numbers), to be able to test the boundaries of the ranges. Testing can be done with a two-value or three-value approach. With two-value BVA, the actual boundary value (in the valid partition) is tested as well as the value that falls immediately outside of the partition (in the invalid partition). With three-value BVA, the value immediately before the boundary (valid), the value on the boundary (valid) and the value immediately over the boundary (invalid) are tested. Two-value is the more common application of BVA; however, the three-value method can be very helpful in cases when a single threshold is crossed, such as a processing date.

Application

BVA can be applied to any ordered partition to determine if the values on and over the boundary are handled properly. Because this is a common place for errors to be made when programming, this tends to be a high yield technique that is relatively easy to apply.

Types of Defects

Defects detected are related to incorrect boundary handling, such as the value of the boundary not being included in a valid range of values or a boundary that is not in the correct place. Essentially, BVA detects defects due to the incorrect usage of a relational operator in the code or a requirement, such as > or =.

Coverage

Coverage with this technique is determined based on how many boundaries are tested divided by the number of boundaries there are (determined by the number of partitions with each partition having two boundaries). A boundary test consists of either two values or three values, depending on the approach selected.

3.3. Decision Tables

Decision tables are used in requirements engineering and testing to help define how business rules should be handled. Decision tables consist of two halves of a table, with the top half typically showing the conditions to be tested (one condition per row) and the bottom half showing the expected results from a set of conditions (one result per row). Columns are used to determine if a condition is to be tested, usually with a true/false or yes/no value for each condition. Results are indicated in each column based on the results expected for that combination of conditions. Multiple results are possible for a particular condition combination (e.g., display an error and return to the previous screen). Condition combinations and interactions are tested with decision tables by verifying that different combinations of conditions result in the proper outcomes or expected results.

Application

Decision tables are well suited for software that must make decisions based on sets of conditions in order to return a proper result. Business rules and any type of non-trivial decision logic are good targets for this type of testing. Because the decision table itself presents sets of conditions and expected outcomes, it is often used as a shortcut to creating detailed test cases. Decision tables can serve as an organized checklist to ensure all significant decision logic is tested without having to further document concrete test cases.

Decision tables can also be used to derive additional rules from the software, based on the knowledge of only one rule. When only one rule and its conditions and outcomes are known, the tester can surmise the proper outcomes of other combinations of the conditions.

Types of Defects

Erroneous decisions and the resulting incorrect outcome(s) are targeted by this type of testing. Decision defects may be caused by incorrect coding or incorrect/unclear requirements. When used in requirements engineering and analysis, decision tables will often identify condition combinations that are not handled or where the expected outcome is unknown, indicating that further analysis is needed.

Coverage

Decision table coverage is determined by the number of columns covered by at least one test divided by the possible combinations (the columns of the table).

3.4. Combinatorial

Combinatorial testing techniques are used to limit the number of combinations of supposedly non-interacting (independent) parameters or conditions that need to be tested. The parameters must be compatible, meaning that any one parameter can be paired with any other parameter. Because some combinations will be eliminated with this technique, it is important to ensure that the conditions should not interact. In the case of testing software across a large set of different browsers and operating systems, testing every possible combination would be prohibitive in effort. Combinatorial testing applies algorithms that are built into tools that mathematically reduce the number of combinations to a manageable set while still preserving a good level of coverage.

This technique is very helpful in reducing the number of test cases when the potential number of test condition combinations are too many to test, either manually or with test automation. It is important to note that additional test cases may be needed to cover important condition combinations not derived from combinatorial test design. In addition, expected results must be documented for each test case, as the combinatorial approach only identifies efficient combinations of test conditions, not outcomes.

There are a number of tools and approaches used in combinatorial testing. The most common of these are:

- Pairwise testing - in this approach all pairs of combinations are tested together, but not all possible combinations
- Classifications trees - this approach allows the user to create a diagram of a "tree" that shows the variables to be tested and then applies an algorithm that will cover all singles, pairs, tuples, etc. of the combination of the variables
- Orthogonal arrays - this approach uses preset arrays of values to determine the combinations to be tested

Application

Any testing that needs to be conducted with non-interacting conditions or variables can benefit from this technique. Such instances could include environment variables (e.g., operating system, browser, or device type) or combinations of internal variables (e.g., car color, car type, or car price).

Types of Defects

This technique generally identifies defects where a particular combination that should be handled is not (e.g., a particular device type is not handled) or where there is an interaction between the conditions (e.g., the color of the car influences the price of the car).

Coverage

Coverage with this technique is determined by dividing the number of test combinations tested by the number of combinations generated by the specific tool or technique.

3.5. Exploratory Testing

Exploratory testing is a combination of learning how the software works (exploring) and testing that it works as expected. Exploratory testing is often session-based (sometimes called session-based testing) and may be guided by test charters which define the objective for a test session. Session sheets may be filled in at the conclusion of the testing to log what has been tested and to note any unexpected occurrences for further investigation. Timeboxing is commonly used to set a time limit for a session. Timeboxing focuses the testing on the defined objective and controls the time that is devoted to a particular charter.

Exploratory testing is most effective when conducted by an experienced tester who is trained to detect issues that an untrained operator could easily miss. Those with good domain knowledge and testing skills are best suited for this type of testing. In an Agile project, someone with a testing background paired with a product owner can help produce the best outcome from these testing sessions.

Application

This technique is well suited to an environment where quick feedback is needed regarding the overall quality of an area of the software. This is sometimes called “smoke testing” or “sanity testing”. It also works well in environments where there is only minimal documentation regarding the expected functionality of the software. In Agile projects, exploratory testing is often used as a first validation that the acceptance criteria for a story have been met. This may be followed by more methodical testing as time allows. In more formal testing environments, exploratory testing may be used to augment other testing techniques in order to check for gaps in the test coverage and to allow the tester to bring more creativity to the task.

Types of Defects

Defects found tend to be functional issues where the requirements have not been implemented correctly or where user transactions and scenarios are not supported. Non-functional issues may be found in the areas of usability and performance, particularly when the testing is concentrated on end-to-end transactions. Security issues, particularly access control, may also become apparent with this type of testing. Although performance and security defects may be found when exploring, it is not a substitute for formal, planned performance and cybersecurity testing.

Coverage

One of the drawbacks of exploratory testing is the difficulty in determining coverage. Because an individual tester may take any number of paths when testing the software, it is likely that the coverage will vary widely and that repeatability of the tests may not be possible unless detailed notes are recorded in the session sheets. It is possible to equate a test session to a test case. When this is done, coverage of sorts can be measured based on the number of sessions (test cases) completed vs. not run.

3.6. API Testing

API testing is more of an approach to testing than an actual technique. API testing focuses on the interfaces between software components rather than the techniques discussed above that are applied to testing primarily conducted from the UI. For example, an application may have an interface that it uses to communicate to a web service. That interface is called an API. When testing this API, the testing would focus on the information passed between the application and the web service, error recovery and data handling.

API testing is often conducted with the assistance of tools that will analyze the expected inputs of an API and present the user with parameters that must be assigned values during the testing. Testing of an API usually focuses on sending values to the API and verifying that the values returned from the API meet the expectations. Understanding the purpose of the API is important for creating valid test data and to validate the response.

Application

API testing is often conducted when testing via another interface, such as the User Interface (UI), would require more effort than is justified by the result. In many cases, manual testing is concentrated on the UI, including the look and feel, while API testing is used to validate that the services used by the front end will perform correctly both with valid and invalid data. In cases where the UI is not yet available, or is unstable, testing from the API may be the most effective approach. Test automation is also sometimes concentrated at the API, where it will not be subjected to changes in the UI that may break the automation scripts.

API testing requires either the use of tools or programming to access the APIs, send data and receive responses. Automating this testing is an efficient approach and will allow testing of multiple services independently without having to drive the interactions from the UI. Because API testing does not depend on a stable UI, testing can often start earlier and automation can be built earlier as well.

Types of Defects

API testing can find a variety of defects, including functional issues where the right data is processed incorrectly, or incorrect data is not detected and reported properly. Error recovery issues, such as transactions being re-processed when a service is not available or is not responding in a timely manner, can also be detected with API testing. Non-functional issues such as performance can be detected with API testing. Cybersecurity testing, including access rights and vulnerability detection, can also be conducted through the API.

Coverage

API test coverage is dependent on the capabilities of the API. At a minimum, all input and output parameters should be checked with a variety of valid and invalid data.

3.7. Picking the Best Technique

There is no single perfect technique, which is why anyone involved in testing should have a good understanding of the various techniques and be able to apply them appropriately. Combinations of techniques are often used to get the best coverage for the least amount of effort. For example, pairing decision tables with equivalence partitioning can help determine the values that need to be entered to exercise the various decision combinations.

It is important to understand the applicability and coverage that can be achieved with any of the techniques. Using techniques in combination will help to provide the level of testing needed for any product. When developers use API testing as part of unit testing, it may make sense to leverage those tests to build the test automation that will be part of a continuous integration/continuous deployment implementation. Similarly, developing good decision tables and automating the high priority condition combinations can give a good level of assurance that the main functionality of an application is working.

Exploratory testing, in both formal and informal approaches, is used extensively in the industry. It provides quick feedback and can be leveraged to learn about a new software release without combing through documentation that may or may not exist. While it is a useful tool in the arsenal, it does not provide a way to measure coverage and, therefore, large areas of the code can be missed. This is particularly so when applied by less experienced testers or developers who are concentrating only on certain areas. It is important to understand the goals of testing and the necessary level of coverage in order to pick the most appropriate technique(s).

4. Test Automation – 200 mins.

Keywords

automation engineer, data-driven, emulators, keyword-driven, simulators, test automation framework, testware

Learning Objectives for Test Automation

4.1 Introduction

None

4.2 Selecting Test Automation Candidates

LO-4.2.a (K1) Recall how ROI for test automation is determined

LO-4.2.b (K2) Explain the factors to be considered when determining if a project is well suited for test automation

4.3 Building Maintainable Test Automation Software

LO-4.3.a (K2) Explain the differences between data-driven and keyword-driven test automation

LO-4.3.b (K2) Understand the purpose of a test automation framework

LO-4.3.c (K1) Recall why test automation must be updated

4.4 Benefits of Automated Testing

LO-4.4.a (K1) Recall why test automation can increase test coverage

LO-4.4.a (K2) Describe the benefits of test automation

LO-4.4.b (K1) Recall how test automation can reduce costs

4.5 Test Automation Risks

LO-4.5.a (K2) Explain the risk factors for automation projects

4.6 Test Automation Success Factors

LO-4.6.a (K2) Explain the success factors for automation projects

LO-4.6.b (K1) Recall the recommended order for automation implementation

4.7 Test Automation Tools

None

4.1. Introduction

Test automation is a method of testing that uses automated test scripts rather than manual test cases. The test scripts are small software programs written in a scripting or programming language that accomplish the goals of a test by controlling the inputs to a software module and verifying that the results match the expectations. In its simplest form, a script mimics the user interaction with the system under test (SUT) and is programmed to report any variances from the expected behavior.

Test automation can be expensive to implement, but, when done correctly, can save enormous amounts of manual testing effort. With effective use of test automation, the quantity of tests executed can be increased, which results in greater requirements coverage, shorter time for execution and higher reliability and repeatability in the testing.

4.2. Selecting Test Automation Candidates

In order to maximize the efficiency of the automation effort, the test automation must be designed for long term use and maintainability. The return on investment (ROI) for test automation is based on the amount of money/time required to build the initial software and to maintain it over its lifetime (the investment), compared with the time saved from the equivalent manual testing effort (the return). A test automation project is only worthwhile if the return will be higher than the investment. Not all software is suitable for automation and attempting to automate unsuitable software will reduce the potential return while increasing the investment.

Good project candidates for automation share some common characteristics.

Expected long term usage – Because test automation is generally expensive to develop due to the cost of the tools and the effort to create the test scripts, the resulting automated tests need to be run multiple times to recover the costs. A standard heuristic is that the target software should remain in production for 2-5 years in order to regain the automation costs.

Stable functionality and interface – To reduce maintenance costs due to changes in the programmed scripts, it is more cost effective to create the test automation at the point when the target software has stabilized. Stabilization is generally reached when a set of core functionality is working and will not be substantially changed. It is also important for the interfaces used by the automation (e.g., the UI, the APIs) to be established and unchanging. Changes to the interface will require updates/changes to the test automation scripts that access the SUT via that interface.

Need for frequent regression testing – The best automated tests are those that are used frequently. When the software under test has need of frequent regression testing, test automation can be utilized to effectively and quickly conduct those tests

and provide reliable and repeatable results. The more frequently the tests are used, the higher the return on investment.

Adequate tool support – Not everything can or should be automated. While there are many tools, and the tool family continues to improve and expand, there may be situations where the right tool is not available. This sometimes happens with code that uses unique interfaces or for embedded software that is communicating with hardware. Some of these issues can be resolved with simulators (i.e., software that is created to act like the software under test) or emulators (i.e., software that is created to act like the software working on the hardware under test), but sometimes the only option is to create a custom tool. Before this option is selected though, there must be an understanding of who will provide on-going support for the tool and how much effort will be required to create the tool.

Adequate skills in the team – Developing test automation is a software development project and should be conducted as such, with proper design, architecture, development, testing and documentation. While some tools provide a more user-friendly interface (generally at a higher cost), true programming skills are usually required to either write the test automation scripts or to write “glue-ware” that will stand between the test automation scripts and other capabilities (such as opening and parsing emails).

Management support – Test automation can be an expensive process and requires management support, understanding and approval. Tools can be expensive to purchase and may have license renewal considerations. Specific programming skills are required which may necessitate hiring people with skills for the selected tools. Because schedules can sometimes be delayed, it is important for management to have a clear understanding of the work required to achieve the desired level of automation.

4.3. Building Maintainable Test Automation Software

Maintainable test automation starts with building an automation framework. The value of a test automation framework is that it provides a way to identify and control all test automation testware. Without a framework, the result is often an inconsistent and unmanageable collection of automated test scripts. Building the framework also requires implementing the right tools, selecting and training the right people and creating the overall automation plan. The following steps are needed to create an effective framework for test automation.

4.3.1. Deciding on Data-Driven vs. Keyword-Driven Approach

Data-driven test automation separates the test script (the steps to be executed) from the data to be used (for input and verification). The data is usually accessed by the script from a spreadsheet or database and is maintained by a test analyst with good

knowledge of the domain to be tested. This allows for the best use of the programming skills of the automation engineer while the testing skills of the test analyst are leveraged to supply and control the data. This separation provides a higher level of maintainability by allowing a single script to conduct many tests where the only variance is the test data.

Keyword-driven test automation goes one step further and uses action words, or keywords, to describe the actions to be performed by the script. The test analyst defines the actions that are to be tested (e.g., add a user) and the data to be used by the actions (e.g., first name, last name, and address). The automation engineer writes a test script that will read the action and then perform the appropriate steps using the provided data. This type of coding allows the script and the data tables with action words to be reusable for multiple tests and limits the areas where changes are needed when new functions are added.

Keyword-driven test automation is particularly well suited for early automation development with Agile and similar SDLCs where automation is built while the software is still evolving. New keywords can be added as new functionality is developed, allowing automation to start early without creating a large maintenance effort later (i.e., accruing technical debt).

4.3.2. Implementing the Framework

When a framework is created, standards are established for the test automation development project. Naming conventions are defined and reusable functions are created to form the basis of the library. Elements from the library can be re-used in various scripts, reducing the need for development and lowering the maintenance requirements by having common shared code. A good framework is essential for creating efficient test automation that will be maintainable across a set of automation engineers. A framework, once established, also allows automation engineers to work on adding more to the function library as time allows, making the framework a living structure.

4.3.3. Building Continuously

An automation project is generally considered complete only when the software under test is no longer changing and no changes to existing tests are needed. Until that time, the automation must be continuously monitored, maintained and augmented to maintain and increase coverage of the software under test. Test automation that is not updated will result in declining coverage over the life of the software under test, increasing the chances of regressions escaping unnoticed. It is important to understand the on-going maintenance costs of a test automation suite to factor into the budget.

4.4. Benefits of Automated Testing

There are a number of benefits to automating testing. The primary benefits include the following:

- Automation can execute more tests in a shorter period of time, which in turns helps to increase test coverage
- Scripted tests will always run the same steps in the same order, providing greater reliability, repeatability, and improved consistency
- Reusability of automated tests facilitates future regression testing
- Faster release cycles are possible with lower regression risk
- Tests that are complex and difficult to execute manually can be good candidates for automation, to reduce the burden on the manual test effort
- Using data-driven or keyword-driven techniques allows more tests to be generated by adding more actions/data with no scripting changes
- The same tests can be run against various hardware and software configurations resulting in compatibility tests being automated
- More time is available for testers to explore new areas of the software that may have previously been untested, resulting in higher quality software
- By improving the frequency of testing (particularly regression testing) the continuous testing required for DevOps initiatives is possible
- By testing earlier with the use of automated tests, defect detection and remediation is less expensive
- Tests can be executed at times when people do not need to be using the system

A well-implemented test automation program will result in overall improvements to the efficiency of the testing, which in turn results in lower test execution costs. Test automation allows an organization to move to a faster release cycle with higher quality, allowing better responsiveness to market needs.

4.5. Test Automation Risks

There are risks with a test automation effort. These risks include the following:

- Management may perceive there is less need for manual testers when automation is in place. In reality, manual testers' roles are expanded when test automation is introduced.
- Automated testing is not automatic testing. Test automation systems can be brittle if not designed and constructed properly. This is a problem that is frequently seen with scripts that have been recorded from execution rather than having been designed for maintainability and reusability. Recorded scripts can be a basis for script development, but the recording must be converted to a reusable, well-structured and maintainable script.
- Test automation does not necessarily improve the effectiveness of testing in terms of defects found, as the quality of the automated tests depend on the quality of the test cases (such as the correct test conditions) and the basis for the tests (such as specifications). It is common for automated tests to become more confirmatory in nature as opposed to discovering new defects.
- Proper tool selection is critically important. Selecting the wrong tool due to an inadequate evaluation process can create extra effort and may render the implementation impossible.

- Tool support must be reliable. Open source tools may go dormant if there is no strong community support. A commercial tool's vendor may go out of business or change directions. Tools may experience unexpected changes requiring unplanned changes in the testware.
- Automation will not solve all testing problems. If a good testing process is not already in place, the automation effort may just speed up chaos.
- Accurate reporting can be difficult. Failures may cascade causing the numbers to inaccurately reflect the quality of the software. Defects must still be analyzed and documented by a person.
- Poor maintainability will be expensive. Potentially, very expensive.

4.6. Test Automation Success Factors

In addition to the characteristics of good automation targets, there are also some common success factors that will help to ensure the automation effort achieves the defined goals.

4.6.1. Find the Right Project

The first step in a test automation project is identifying the appropriate software system candidate. Systems that are near their sunset years and will soon be retired are generally not considered good candidates for automation as the return on investment will be short-lived. An exception to this might be starting test automation development on a retiring system to capture data and tests that will be valid in the replacement system, albeit with adjustments for the new interface.

4.6.2. Build Automatability into the System

Building a maintainable and reusable test automation suite starts with ensuring that the design of the system to be tested supports and facilitates test automation. One common issue in test automation efforts is trying to automate software which is inherently difficult to automate. This could be because it contains inconsistently named or mis-named objects. For object-based test automation (which is the standard and preferred approach in test automation), object identification is essential for creating maintainable test automation. Limited test access to APIs, a lack of observability during testing, insufficient logging, etc. can also significantly complicate the test automation effort, leading to more time and money spent on maintenance throughout the life of the automation. Early involvement by the automation engineer during the design of the system to be tested can help to ensure that the necessary support for the automation is built into the system.

4.6.3. Show Early Success

A large automation project can take years to complete. It is important to create interim milestones and demonstrate that the milestones are being met. In general, there are three areas of automation focus. In order of priority (and to provide the most visible return on investment quickly), implementation should proceed as follows:

- Create acceptance/verification tests for the software build – these are positive path tests that are used to verify that the code added to the build did not "break" the build. These tests are run every time code is committed and provide fast feedback to the developers if any issues have been introduced. In a frequent build environment, such as Agile, or in a continuous integration model, this automated testing is critical to the success of the process.
- Regression tests – regression tests are generally stable and well-defined. Automation for these tests is efficient because they will be used many times over the life of the product and that automation will save significant manual test execution time. Automated regression tests allow a team to release software safely, more frequently, and free testing time for more important areas.
- Functionality tests – in general, test automation is faster when the software being tested is stable. This results in fewer changes to the automation because the code being tested is not changing. That said, functional testing still needs to be automated, particularly in the rapid SDLCs, such as Agile. In this case, the code may not be stable as new features are still being added and evolved. Maintainability in design is critical for the test automation effort to be able to make forward progress and not be consumed by maintenance issues.

4.6.4. Review the Plan

Original estimations for an automation effort are sometimes wrong. This may be due to technical issues, frequently changing SUT, slow ramp-up of the testing team or other reasons. Reviewing the plan and validating the schedule should occur on a periodic basis to understand changes to the schedules and identified risks. It is also important to continually set realistic milestones and report on the achievement of those to the management sponsors of the effort.

4.6.5. Define Ownership

Automation requires upkeep and analysis to ensure it is working properly. Identifying a resource that can be called upon to detect and correct issues is critical to continued, uninterrupted use of automation in testing. Likewise, it is important to identify who will run the test automation and when it will be run. Often the manual testers assume the responsibility to execute the automation and do the preliminary debugging if an issue is found (by manually testing any failures that occur). This helps the manual tester to engage with the automation and also frees the automation engineers from debugging issues that may be due to data or environment changes.

4.7. Test Automation Tools

Early tools depended on recognizing characters that a user entered onto the screen or via a command line. With the advent of graphical systems, tools evolved to support coordinate-based positioning and text recognition of fonts and screen characters. Eventually, tools further evolved to recognize the native objects displayed on the windows, where multiple object attributes could be verified and the objects themselves could be accessed for interaction. With the advent of interfaces such as those used

for web services, tools additionally supported the ability to bypass the UI and call the function, service, or method directly, often through the API.

Future uses of automation tools will include robotic process automation (RPA), artificial intelligence (AI) and machine learning to allow the tests to adapt to the executing software, providing greater coverage and less up-front programming from the automation engineer.

Test tools cover a range of features, functions and levels of customizability. For teams with strong programming skills, tools that allow development of purpose-built functions are appropriate. For less technical test teams, tools that require minimal programming may be more appropriate. It is important to identify a tool that meets both the needs of the SUT and the skills of the team.

5. Performance Testing – 200 mins.

Keywords

concurrency testing, load test, objectives-based reporting, operational profiles, performance testing, risk-based reporting, scalability, stress test

Learning Objectives for Performance Testing

5.1 Introduction

None

5.2 The Purpose of Performance Testing

LO-5.2.a (K1) Recall the goals of performance testing
LO-5.2.b (K2) Explain why defining and getting agreement on performance requirements is important
LO-5.2.c (K2) Explain how performance testing aligns with the SDLC

5.3 Performance Testing Risks, Benefits, and Challenges

LO-5.3.a (K1) Recall the unique challenges of performance testing
LO-5.3.b (K1) Recall the risks of performance testing
LO-5.3.c (K1) Recall the benefits of performance testing

5.4 Performance Testing Approach

LO-5.4.a (K2) Describe the components that should be included in a performance test plan
LO-5.4.b (K2) Explain the factors to be considered when defining the test approach

5.5 Conducting Performance Testing

LO-5.5.a (K1) Recall the steps for conducting performance testing
LO-5.5.b (K1) Recall the components that should be checked during performance test preparation
LO-5.5.c (K1) Recall how performance test results differ from functional test results
LO-5.5.d (K1) Recall the types of performance test reporting

5.6 Performance Test and Analysis Tools

LO-5.6.a (K1) Recall why tools are necessary for performance testing
LO-5.6.b (K2) Describe the challenges that can be encountered with performance testing tools

5.1. Introduction

Performance is a major quality factor in most systems and applications, regardless of the computing environment. A system or application may be functionally correct, but if it fails to deliver the needed performance it is likely to be considered a failure. Performance testing is one way to measure system performance in advance of deploying the system.

Performance testing should start at the component (unit) test level and continue until system deployment. Waiting until the end of a project to conduct performance testing carries the high risk that performance problems will not be solvable due to time, money and technological constraints.

One of the most costly and publicized performance failures occurred during the Facebook Initial Public Offering (IPO), when the NASDAQ stock exchange could not handle the volume of trading on the day of the IPO. The cause of the failure was determined to be three lines of code that got into an endless loop. To date, NASDAQ has paid over $80 million in fines and restitution. In addition, the United States Securities and Exchange Commission imposed requirements on NASDAQ to help prevent similar failures in the future [NASDAQ].

5.2. The Purpose of Performance Testing

Performance testing can have multiple goals and purposes, including:

- Measuring system performance under given conditions
- Determining the maximum concurrent user load or transaction volume a system can handle before it fails
- Providing information to assist in capacity planning for a system

Significant time and money can be invested in performance testing. In order to obtain the best return on investment, the testing must be targeted correctly and the goals clearly defined. This can be difficult since different stakeholders may have varying, even conflicting, views of “acceptable” performance. In order to be successful, there must be agreement in the project team regarding what the performance testing will measure and what will constitute a successful result.

5.2.1. Defining Performance Requirements and Getting Stakeholder Agreement

Documenting performance requirements can be challenging. Eliciting the requirements can be even more difficult. When documented, performance requirements are normally recorded in over-arching requirements, such as “all system responses to users that require longer than 3 seconds must display a wait notification”. In an Agile environment, these requirements may be documented in specific non-functional Epics.

Without stakeholder performance requirements, it is hard to know if the observed levels of performance are adequate, making it difficult for testers to make a reasoned assessment of test results. A complicating factor in determining system performance requirements is that the assessment of “acceptable” performance can be subjective.

Getting agreement on performance requirements can be quite challenging because of the subjectivity of opinions and the cost required to achieve higher levels of performance. For example, one stakeholder group might feel that a response time of two seconds is acceptable, while another group might feel that a response time of one second or less is required. The cost to achieve the one second response time might be quite high (e.g., 5x improvements in hardware, more robust networks, major modifications to code or databases), whereas two second response time may be easily achieved.

5.2.2. Aligning Performance Testing in the SDLC

As with all forms of testing, performance testing needs to be integrated with the SDLC. Regardless of the SDLC, full lifecycle performance testing is needed to detect performance problems early, when they are easiest to fix.

The following activities in an SDLC have key tasks for performance testing:

- During requirements definition, the specific requirements for performance (and definition of the expected load) should be clearly defined and agreed to by all stakeholders. This will eliminate debate when the results are reviewed later.
- During design and coding, performance factors must be considered and built into the design and the subsequent code. Inefficient code or an inefficient design will be difficult and costly to fix later.
- During integration and system testing, new performance testing opportunities can appear. Any new integration may introduce inefficiency and potential performance bottlenecks.
- During acceptance testing, a major objective is to assess performance within the business or operational context. In some cases, performance testing is part of contract acceptance testing and operational acceptance testing.

5.3. Performance Testing Risks, Benefits, and Challenges

Performance testing has unique challenges, primarily due to lack of understanding of the complexity and cost of good performance testing. These include the following:

- Getting definition and agreement from stakeholders with regards to acceptable system performance
- Getting adequate funding for performance testing tools
- Acquiring the best fit solution for a performance testing tool, within the schedule and budget of the project
- Accurately profiling load levels at given periods of system usage

- Acquiring skilled test engineers to plan, design and conduct a realistic performance test
- Building a representative performance test environment

Unless these challenges are met and understood, the performance test effort may never start or will not be effective. In addition to the above challenges, performance test efforts have risks that must be considered and mitigated in order to achieve the expected benefits.

5.3.1. Risks

Performance testing has the following risks:

- Inadequate performance test design, resulting in inaccurate or incomplete test results
- Incomplete coverage of protocols and connectivity, resulting in important aspects of performance testing for a particular system or application being missed
- Inadequate test environments, resulting in inaccurate test results and erroneous conclusions
- Inability to apply the performance tool correctly, resulting in inefficient, inaccurate, and inconclusive tests
- Inadequate coverage of functions, resulting in incomplete test results
- Inadequate amounts of user and data load, resulting in inaccurate and incomplete test results
- Lack of defined stakeholder requirements for performance, resulting in the inability to identify performance targets
- Lack of agreement on defined stakeholder performance requirements, leading to conflicts concerning acceptable levels of system performance
- Lack of appropriate performance metrics, resulting in test reports that are incomplete or lack depth of meaning.

5.3.2. Benefits

If the challenges are met and risks mitigated, significant benefits can be expected from performance testing. These include the following:

- Opportunities to “right-size” the system to handle expected load
- Opportunities to plan mitigation steps if loads exceed expected levels
- Early test results can help define the acceptable levels of performance
- Expectations for future growth can be tested to know if the system will be able to support it
- System performance weaknesses can be identified and fixed before being discovered in production
- Long lead time items, such as adding more hardware or improving network resiliency, can be addressed in adequate time before a production launch

Because performance issues may be expensive or time-consuming to rectify, identifying them early on allows the greatest chance for mitigation before an exposure

in the production environment. Performance testing is often intended to merely confirm expectations that the overall system is fit for purpose, but frequently results in identifying significant problems that must be corrected.

5.4. Performance Testing Approach

Performance testing often requires a separate planning effort. This effort is focused on the particulars of the performance testing, such as sizing the test systems, procuring resources and planning uninterrupted testing time.

5.4.1. Defining the Test Plan

Performance test plans generally include the following information:

Scope

The correct setting of scope in performance testing is a critical activity to ensure that the targeted objectives are met. The scope is often framed by the following:

- Functionality – What features will be included or excluded from the performance testing?
- Architecture – Which aspects of the system architecture (e.g., networks, APIs, devices, databases) will be included?
- Transactions – What typical user transactions will be included in the testing?
- Users – Which classes of users should be included based on selection factors (e.g., location, type of transactions, demographics, concurrent usage)?
- Data – How much data should be used and how should it be accessed?

Strategy and Approach

The performance test strategy and approach can define the way performance testing is to be conducted. For example, an organization might decide to outsource performance testing if it lacks the tools and skilled people to conduct the testing on its own.

Risk Assessment

As in much of testing, performance testing can also be risk-driven to focus testing on areas of the system, or on functions and transactions, that carry the highest risk, such as those with the highest use.

Definition of Test Objectives

Performance test objectives describe, at a high level, what the performance test is intended to achieve. Oftentimes, these are worded as performance objectives to verify and/or validate. For example, “Verify the system response time is 1 second or less at peak load times.”

Responsibilities

Team roles must be defined, along with roles and responsibilities for those external to the team and the organization. Performance testing often requires the support of

system architects, database analysts, network engineers and other specialists who can participate in troubleshooting and system tuning.

Reporting Metrics
Metrics have a very important role in evaluating performance test results. It is important to define which metrics are most meaningful to stakeholders – both business and technical – and ensure those metrics are tracked by the selected monitoring tools. Some metrics, such as average response time, may not be meaningful or as useful as the response time for 90% of the users. This will help eliminate any outliers that can skew the average.

5.4.2. Defining the Test

Depending on the formality of the environment, the test approach may be defined as part of the performance test plan or defined separately. The test approach must consider the following:

Environments
Test environments are a key concern in performance testing because of the need to obtain representative results. The challenge is that full-scale dedicated test environments are often difficult to build due to cost and complexity. One possible alternative is to use a cloud-based test environment that can be rapidly scaled to simulate production configurations.

Unfortunately, it is not wise to simply predict performance at higher levels of scale based on lower system capacities. It is normally necessary to apply realistic load with realistic system conditions.
In some cases, such as in new system development, the system under development can be tested in the same system configuration that will eventually become the actual production system.

Load and Throughput Profiles
To accurately design performance tests, current and predicted profiles must be understood for both load levels and data throughput, such as peak load times and load levels. A common way to know performance profiles is to measure and study user behavior and levels as captured in system analytics.

Operational Profiles
Operational profiles describe the functions users are expected to perform on the system and the frequency of these activities. These can be seen as simple functions or wider-ranging workflows for end-to-end transactional testing of the system, sometimes called “transaction threads”. Operational profiles are implemented as scripts to be executed by virtual users, creating an accurate environment for measuring performance.

Test Data

Representative amounts and types of test data are often at the heart of performance testing. This data may be generated by tools or copied from a production environment if no private data is involved.

5.5. Conducting Performance Testing

In order to run the performance test, several other steps are needed. The tests must be prepared (including the environment and data), they must be executed, the results must be evaluated and reported, and there may be a need for on-going monitoring.

5.5.1 Test Preparation

Prior to test execution, the following components should be verified as ready for the testing:

Test Environment
It is often helpful to run preliminary tests to make sure basic test environment elements are in place and working correctly and that proper access is available. It is also important to validate that all co-existing applications and systems that could cause a performance load are in place and operational, as these systems can also place performance load on the test environment.

Test environment capacities need to be verified as correct. For example, CPU levels, database size and network configuration all have an impact on system performance.

Data
All databases and files should be populated with the designed volumes of test data to ensure the tests are not blocked due to insufficient or incorrect data. Generating this data and verifying its correctness must be conducted prior to performance test execution.
Testware (Test Scripts, Procedures, etc.)
As part of test preparation, it is vital that the designed and implemented testware is in place and executing correctly as designed. An effective activity is to conduct a preliminary test using a representative sampling of test procedures to verify functional correctness and data accessibility and accuracy.

5.5.2. Test Execution

If the preparatory steps have been completed, then test execution becomes a matter of running the tests in the tool and monitoring the execution. Monitoring features are a part of many performance test tools. However, the interpretation of test results requires human intelligence.

Sometimes, it is only by observing test results in real-time that some performance anomalies may be identified. For example, it is common in a performance test to ramp up load levels. In some test tools, the impact of increasing system load can be seen by analyzing graphical representations after the test is complete. However, other tools

may primarily display test results in a text-based format. In those cases, it is helpful to observe system behavior and metrics as the test is in progress.

5.5.3. Test Evaluation

In contrast to functional testing, where test outcomes can be evaluated as “pass” or “fail”, performance test results tend to be more comprehensive and informative. The main question that performance testing seeks to answer is “Does system or application performance meet stated performance goals or requirements?”

It is important to understand that performance testing is highly dynamic and dependent on many factors, such as user load, workflows performed, system capacity (CPU speed, database efficiency, code efficiency, and networking configuration), and system configuration. Changing any of the variables can significantly affect the results.

Performance testing is a snapshot view. Even a small change to the system can cause improvements or degradation of performance. For this reason, performance testing should be performed throughout system development and after release to production.

Performance testing is sometimes used to assess the system’s capabilities. While load testing can determine performance levels at given load levels, stress testing can reveal the maximum capacity the system can handle based on a given configuration. This information can help capacity planners to know if system upgrades will be needed, and if so, when they may be needed.

5.5.4. Test Reporting

The main deliverable of performance testing is the reporting of the test results. Performance test reporting will vary depending on the audience and the purpose of the test. Since each stakeholder group has differing levels of technical understanding, test results need to be presented in ways that each group can understand and find meaningful.

Risk-based Reporting
As in other forms of testing, risk-based reporting can help identify which aspects of system performance may carry the most relative risk. This risk-based view of performance test reporting can help stakeholders understand where to focus efforts for the most effective system implementation decisions.

Objectives-based Reporting
In objectives-based reporting, performance test results are reported with traceability to performance test objectives, such as the expected response time to a specific user action. This allows testers and stakeholders to know whether or not performance objectives have been met as demonstrated by the success or failure of performance tests.

5.6. Performance Test and Analysis Tools

5.6.1. Why Tools are Essential for Performance Testing

Performance testing is one type of testing that is not feasible without tools. Tools are used for the following:

- Creating simulated concurrent user load – This simulated load is used for concurrency testing. It is more precise and does not require as many resources as trying to test with large numbers of people. Concurrency testing is used to determine how the system will perform with a consistent load of virtual users who are “concurrently” using the system.
- Sustaining high levels of load – Even if it was possible to generate enough load with actual people, it would be difficult to sustain the load long enough to get a good measurement or to allow repetition of the test.
- Accurately measuring load levels and system response time, CPU utilization, memory allocation, etc. – It takes more than a stopwatch to measure response times. Most performance test tools have features to measure aspects of the SUT that directly impact performance. For tools that lack robust monitoring functionality, it is possible to obtain tools that independently monitor test performance.
- Repeating performance testing whenever needed – Performance tests must be repeated as changes are made to the application and system. Often, during performance testing, system tuning takes place to improve the performance. This requires re-execution of the tests to ensure that the expected improvement has been realized.

5.6.2. Tool Challenges

Although tools are needed for performance testing, there are some challenges to overcome:

- Performance test tools do not know what to test – The performance tester must determine which functions to test, how many concurrent users to simulate, which data to use, and other conditions to achieve the test objectives.
- The environment has to be representative – For many people, building a performance test environment may be the greatest challenge of all. Adequate performance test environments may require an investment in hardware, software, people, and tools – all scalable to the level of production use. Cloud-based virtual test environments have become an attractive alternative to physical environments in some cases.
- High volumes of test data will be needed – This may require the use of test data generation tools or strategies to modify a copy of existing production data.
- Expertise is needed – Performance testing requires skills and experience not commonly found in many organizations. It is common to obtain outside consultation when first starting to plan and conduct performance testing.
- Tools can be expensive – The most robust and popular performance test tools can be very expensive. While new tools in the marketplace help to keep pricing competitive, organizations that have large investments in performance test

tools may be less inclined to switch to more affordable tools. Even open source tools have a cost in learning and maintenance.

Selecting the proper tools requires evaluating a number of factors including long-term cost, training requirements, vendor reliability, etc. Because performance tools may be significantly expensive, it is important to consider the life expectancy of the tool, the ROI and the likely future requirements for tooling.

6. Cybersecurity Testing – 200 mins.

Keywords

cyberattack, cybersecurity, cybersecurity testing, fuzzing, cybersecurity controls, social engineering, system hardening, threat actors, threat intelligence, threat modeling, vulnerabilities

Learning Objectives for Cybersecurity Testing

6.1 Introduction

LO-6.1.a (K1) Recall the definition of cybersecurity
LO-6.1.b (K1) Recall the cybersecurity attributes of assets
LO-6.1.c (K1) Recall the function of cybersecurity controls

6.2 The Purpose of Cybersecurity Testing

LO-6.2.a (K2) Explain the necessity to both verify requirements and validate the system during cybersecurity testing

6.3 Cybersecurity Differences

LO-6.3.a (K2) Explain the unique differences of cybersecurity testing

6.4 Cybersecurity Testing Approaches

LO-6.4.a (K1) Recall when cybersecurity test planning should take place within the SDLC
LO-6.4.b (K2) Explain how cybersecurity testing contributes to risk management
LO-6.4.c (K2) Explain how testing can enhance the stages of a cybersecurity effort

6.5 Conducting Cybersecurity Testing

LO-6.5.a (K1) Recall why cybersecurity testing must be pre-authorized
LO-6.5.b (K2) Explain how white-box testing is used in cybersecurity testing
LO-6.5.c (K2) Explain how black-box testing is used in cybersecurity testing
LO-6.5.d (K1) Recall the controls that can be targeted by cybersecurity tests

6.6 The Environment of Constant Change

LO-6.6.a (K1) Recall ways to keep up to date with cybersecurity and cybersecurity testing practices

6.1. Introduction

Cybersecurity was commonly called information security or information assurance - the latter emphasizing the need to be confident or assured that digital resources are in fact secured. Cybersecurity refers to protecting computer systems, including their electronic data, software and hardware, from theft or damage as well as from disruption or misdirection of the services they provide.

Valued data or computerized processes are referred to as assets. Their security attributes typically are described as confidentiality (and its corollary, privacy), integrity, and accessibility: the so-called "CIA" triad. Confidential material should only be entrusted to designated parties for specific purposes, respecting the owner's privacy. The integrity of an asset is dependent on the assurance that it has not been tampered with via unauthorized access. When data or a service is needed, it needs to be accessible only to authorized users.

Cybersecurity addresses the need to protect the integrity and confidentiality of digital assets, as well as the accessibility of online processes. Computerized systems in the past suffered almost all of their failures due to defects in design or implementation. Today, however, so-called threat actors (malicious individuals or groups) conduct cyberattacks to compromise systems by deliberately misusing them. Critical infrastructures – including energy, communication, transportation, and finance – are now heavily computerized; defenses against online assaults are the essential responsibility of cybersecurity.

Protection of system resources and processes is the function of cybersecurity controls. These controls range from straightforward precautions such as account passwords to sophisticated automated detection and response mechanisms. Different cybersecurity controls may be designed to reduce the likelihood of successful attacks or to minimize the damage done if any attacks succeed. None of these protections are flawless, hence, anyone wishing to compromise computerized systems will be looking for weaknesses, known as vulnerabilities, in the controls.

Consider the situation in terms of the classic elements of criminality: motive, means, and opportunity. Vulnerabilities in cybersecurity controls are the opportunities - success depends on bypassing these controls. Malicious individuals and groups might be motivated by greed, revenge or ideology. But they will need knowledge, skills, and resources – the means of attacking – to take advantage of an opportunity and successfully compromise security. The strength of their motivation determines their persistence or willingness to risk detection.

6.2. The Purpose of Cybersecurity Testing

Cybersecurity testing probes systems to reveal potential failures in furnishing the desired level of security. As with any other testing, cybersecurity testing reveals the

presence of defects but cannot confirm their absence. It provides value both as verification ("Was it built right?") and as validation ("Was the right thing built?").

Where security requirements have been specified, testing can verify how adequately these requirements are satisfied. Threat modeling, based on threat intelligence, anticipates the nature of assaults that might be encountered in the operating environment of the system. System designers provide corresponding controls, and testing can verify the extent to which those controls provide the specified protection.

Specification of what a system should do is often less difficult than elaborating what the system should not do. Beyond defining and confirming controls, one must probe for unacceptable behaviors and validate proper functioning in an operational environment. To be realistic and meaningful, these tests need to model the projected capabilities of realistic adversaries, as identified through threat modeling.

6.3. Cybersecurity Differences

- **It must be responsive to a rapidly changing environment** – Vulnerabilities in complex interrelated systems are usually subtle and may lie dormant for years before suddenly being exposed. Systems also evolve by the modification or addition of features and by new interactions with other hardware and software, any of which can introduce new vulnerabilities.
- **The adequacy of established controls is under constant active probing** – Threat actors of widely varying motivations, capabilities, and resources fill the arena. When they can expose the weakness of a control they will promptly seek to exploit it and, in some cases, expose it to other threat actors. The ability and resources possessed by threat actors will only increase, especially as state-sponsored attacks increase.

- **Cybersecurity testing must address detection, response, and recovery after controls fail** – It needs to establish the resilience of the system when vulnerabilities are successfully exploited. It must evaluate how well risk mitigation supplements risk avoidance.
- **Human shortcomings are at least as prominent as technological challenges** – Exploits through various low-tech means (known as social engineering) are a leading cause of security failures. Countermeasures must address policies, procedures, and awareness campaigns.

6.4. Cybersecurity Testing Approaches

Of all the quality attributes, security might be the most difficult to "bolt on" or "patch in" after substantial software development has already been done. Cybersecurity test planning must contribute to the design and implementation of a system as early as possible. It should reinforce secure design and coding best practices, and include a risk analysis that is focused on cybersecurity.

Risk exists when a threat might take advantage of a vulnerability. Lowering the probability of that occurring is known as risk avoidance, and decreasing the consequences when it does occur is known as risk mitigation. To go one step further, risk mitigation must also be tested by probing the responses that occur when a control fails. By revealing otherwise undetected weaknesses, cybersecurity testing can direct efforts that support risk avoidance. The goal is to reduce overall risk exposure (and its uncertainty) to an acceptable level.

Potential security weaknesses in isolated system components may be investigated by static techniques. Manual or automated inspections, such as port scans and code scans, can provide early identification of insecure design patterns or coding practices. After a completed system is installed in its operational environment, additional dynamic security issues will arise (i.e., issues that can only be seen during execution) and must also be addressed.

Testing can enhance each stage of a cybersecurity effort as follows:

- Identify – Confirm identification, categorization and prioritization of the data and processes to be safeguarded (note that an adversary might value these assets differently and have different priorities)
- Protect – Analyze the protection of each asset category with corresponding controls that will stop or slow attempts to compromise the system
- Detect – Demonstrate the ability to detect intrusions and provide timely actionable analysis
- Respond – Exercise the responses and determine how well affected parties are notified, damage is minimized, and further access is shut off
- Recover – Demonstrate the speed and extent of recovering operational status and restoring data and confidence

6.5. Conducting Cybersecurity Testing

A testing project should establish terms of engagement that define the nature and scope of activities. If a live system is to be tested, it should specify how much notice will be given and to whom. All cybersecurity testing must be pre-authorized in writing by appropriate management to avoid testers being mistaken for malicious insiders (which can carry criminal penalties).

Cybersecurity testing covers the full software development lifecycle, so it will parallel and reinforce secure software engineering practices. It may be performed or be a part of reviews and audits at key specification, design, implementation, and integration stages. It will evaluate the performance of threat modeling and static code analysis.

Cybersecurity testing, like other types of testing, can and should be conducted in two different modes: “black-box” - which attends to external system behaviors; and “white-box” (or clear-box) – which utilizes structural knowledge of the as-built system. In black-box testing, an “outsider” will gather information via system reconnaissance. They will use that information to craft attacks that can be mimicked in penetration

testing to see what they can attack. In white-box testing, the “insider” threat is modeled by treating the system’s internals as visible and thus uniquely exploitable.

A wide range of testing approaches are available. One automated black-box technique involves fuzzing - inserting random variations of expected input values to detect sensitivities that might be exploited. Another mode of testing is derived from the practice of military exercises. A designated “red team” of attackers (composed of internal or external experts) is pitted against a “blue team” of defenders (drawn from internal operations personnel). The exercise may be performed on a facsimile of the SUT, or may be a “tabletop” exercise without a real system.

Targeted tests should probe the adequacy of specific controls. These may include:

- Policy and procedure enforcement
- System hardening (reducing the attack surface by eliminating as many security risks as possible)
- Access control mechanisms for authentication (who you are), authorization (what you can do), and accountability (what did you do)
- Input validation
- Encryption
- Error and exception handling
- Intrusion detection
- Malware scanning
- Resistance to social engineering

6.6. The Environment of Constant Change

Best cybersecurity and cybersecurity testing practices may be found in various technical guidance documents and consensus standards, as well as through professional organizations and educational institutions. “Best practices” may be “required practices”, if mandated by government laws. Regulatory requirements may also apply to specific financial, medical, transportation, energy or other sectors. In such settings, the adequacy of testing would be judged by the extent to which it confirmed compliance with applicable mandatory requirements.

New technologies and new applications of existing technologies will continually introduce new security concerns. Seemingly stable, long-time tools and applications often reveal long-time vulnerabilities under the scrutiny of determined adversaries. External laws and regulations, as well as user expectations, change over time too. Keeping up-to-date with changes in these requirements and best practices, as well as technological advances, should be accomplished through ongoing professional development - reading, conversing and attending events with fellow practitioners.

7. Usability Testing – 150 mins.

Keywords

accessibility, personas, usability, user experience, user interface, Web Content Accessibility Guidelines (WCAG)

Learning Objectives for Usability Testing

7.1 Introduction

LO-7.1.a (K2) Explain the difference between efficiency, effectiveness and satisfaction in usability testing

7.2 Focusing the Usability Testing

LO-7.2.a (K1) Recall sources for determining the needs and expectations of the users

LO-7.2.b (K2) Explain why depth or breadth testing would be used

7.3 Usability Test Participants

LO-7.3.a (K1) Recall the roles of professional testers and end-users in usability testing

7.4 Usability Test Planning and Design

LO-7.4.a (K1) Recall the use of personas in usability testing

LO-7.4.b (K1) Recall the purpose of user experience evaluation

7.5 Scheduling and Conducting the Tests

LO-7.5.a (K2) Explain when usability testing should occur in the SDLC

LO-7.5.b (K2) Describe how usability tests are conducted

LO-7.5.c (K2) Explain how results are gathered from usability tests

7.6 Standards

LO-7.6.a (K1) Recall standards that support usability testing

LO-7.6.b (K1) Recall the components of the usability software quality characteristic

7.7 Accessibility

LO-7.7.a (K1) Recall the meaning and purpose of accessibility testing

LO-7.7.b (K2) Summarize the common references and standards for accessibility testing

7.1. Introduction

Usability and accessibility of software are important quality characteristics and may determine whether or not the software is successful. Usability testing is conducted to evaluate how well a system can be used by the target users to accomplish a specified goal. Accessibility testing, which is considered a subset of usability testing, is conducted to ensure that users of all abilities can successfully use the product.

Efficiency (how much effort is required to accomplish a goal?), effectiveness (is the desired result achieved?) and user satisfaction (is the user “happy” with the software?) are factors that are often considered during usability testing. In addition, proper accessibility testing may be required as part of the overall testing project if accessible software is mandated by law. The results of both types of testing can then be used to improve the design to better meet the needs of the users.

The timing and scope of usability and accessibility testing can vary widely from project to project and, depending on the areas of a product targeted for evaluation, may be difficult to schedule and fit into the project lifecycle. Because of this, it is important to understand the goals of the tests before starting, consider when the testing should occur, and verify that there is adequate time in the schedule to plan the tests, review the results, and consider what changes should be made to the product.

7.2. Focusing the Usability Testing

In order to conduct effective usability testing, the tester must understand the needs and expectations of the users. This may include gaining an understanding of the product itself, as well as the day-to-day activities of the users. This information may be gathered from the following sources:

- Process maps
- Business cases
- Use cases
- Interviews
- Observation
- User guides and documentation
- Expert users or domain experts

Usability testing should concentrate on areas of the software that will be most frequently used and are most important to the user. If there is an existing product to use for comparison, complex user interfaces and existing features with a history of a high number of technical support issues should also be evaluated, to avoid perpetuating any problems.

Prioritization is important because the scope of usability testing is often limited by project schedule and budget. For new software, where usage is anticipated but not known, sampling may be done to allow testing across the overall User Interface (UI).

This type of testing will provide information regarding the expected user experience and can serve as a breadth-based test. For areas where extensive usage is expected, depth-based testing is appropriate. By approaching the testing with a calculated combination of breadth and depth testing, the best coverage can be obtained in the shortest period of time.

7.3. Usability Test Participants

In larger organizations, usability is a specialized discipline that leverages usability experts with training in psychology and design. Ideally, these experts also have some knowledge of software development and at least some exposure to testing. When professional testers conduct usability testing, they often work closely with usability experts to design tests and check for standard usability characteristics (e.g., navigation, number of mouse clicks to accomplish a task, or screen layouts). This type of controlled testing should be augmented with testing by real (or potential) users because users know what they expect the software to do, what they need it to do and how they expect it to work.

For a given project, the usability testers should represent the skill levels and knowledge of the actual users. If there will be Subject Matter Experts (SMEs) using the software, they will have a different approach than a novice user. Both of these user types should be considered during usability testing.

While both types of testing are important, the feedback from real users is sometimes given more credence than feedback from professional testers, since the users often have actual experience with the product. This is particularly true when an observation is subjective (e.g., “the layout is confusing”).

7.4. Usability Test Planning and Design

Test planning is recommended for usability testing, in order to define the test approach, identify the goals, and to gain agreement among different stakeholders regarding the scope and expectations. With formal usability testing, guidelines for observer behavior are usually described in the test plan. Observers may be expected to not interfere, to provide help only when requested or to step in when the user appears to be getting frustrated.

As part of usability testing, users are often categorized into personas. A persona is a representation of a type of user and is often given specific characteristics (such as age, gender, profession, etc.). These personas are used during usability design to help ensure that the needs and desires of all targeted user sets are met. During testing, the activities and reactions of real people can be evaluated against the personas that were used during design.

Another aspect of usability testing is user experience (UX) evaluation, which should also occur as early as possible. User experience refers to a person’s perceptions of

and responses to the software before, during and after usage. For example, happy anticipation of the use of the software is a UX characteristic. Brand image and presentation of the software contribute to user satisfaction. Like usability, the user experience must be designed into the software in order to be achieved in a cost-effective manner.

7.5. Scheduling and Conducting the Tests

7.5.1. Early and Continuous Testing

As with all forms of testing, it is less expensive to correct usability issues when they are uncovered early in the SDLC. Early testing is often conducted as the software is being designed (sometimes called formative testing as the software is being "formed"). This allows users and specialists to review the design before it is coded. It may be done with written descriptions, wireframes (low fidelity) or prototypes (high fidelity).

Ideally, usability testing is a continuous activity, performed frequently during software design and development. This will provide timely feedback and will help influence the design as the project progresses. In an Agile methodology, the product owner normally assists with usability testing during iteration testing to ensure that the product meets expectations.

7.5.2. Conducting the Usability Test

Formal usability tests are typically conducted in a usability lab, which may be equipped with video and audio recording devices, two-way mirrors and a separate seating area for observers. A formal lab may have software installed which tracks eye movements of the subjects and records the interactions of the subject with the product. Less formal testing may be conducted in an office setting or even a test lab environment. Ideally, the tests are conducted in an environment that closely mimics the user's work environment. This will provide a more realistic experience (e.g., lighting, noise, and distractions) and will help the user to better evaluate the software.

Usability tests are sometimes conducted by having the subjects perform tasks that have been designed in advance by a usability tester. Testing may also be conducted by assigning general tasks to the users and having them figure out how to accomplish those tasks on their own. User manuals and process documents may be used as guidelines for testing and the testing is sometimes used to review the correctness of the documents.

During usability tests, subjects generally are asked to vocalize what they are thinking while they are working through their tasks (i.e., think out loud), express any confusion that they may have and talk about what they are doing. If observers are present in the room during the usability tests and are required to be silent and passive, they can give no help or feedback to the participants. This is an important consideration as it is sometimes difficult for the observers to avoid influencing the tests.

7.5.3. Gathering Results

Feedback from the tests are usually gathered in two ways: defect reports and questionnaires/surveys.

Where usability or user interface defects are identified, the normal defect lifecycle should be followed, with particular attention paid to maintaining consistency. For example, if users object to the way a button is displayed, all buttons should be reviewed to determine if broader changes are needed.

Questionnaires and surveys are used to gather feedback regarding the effectiveness and efficiency of the software and the user's satisfaction with their experience. These surveys may also extend into the UX areas (e.g., emotions, perceptions, preferences, image).

7.6. Standards

There are several international standards that deal with usability. The following are commonly used:

ISO 9241-210 discusses human-centered design. This is based on understanding the expected use, specifying requirements, producing solutions, evaluating the solutions and eventually designing the best solution to meet the usability requirements.

ISO 25010 describes the software quality characteristic called usability. This breaks usability into a set of characteristics as follows:

- Appropriateness recognition – Can the user determine if the software is appropriate for their needs?
- Learnability – Can the user figure out how to accomplish a task and are they able to apply that knowledge the next time they want to accomplish the same or a similar task?
- Operability – Is the software easy for the user to operate and control?
- User error protection – Does the software help prevent the user from making errors?
- User interface aesthetics – How pleasing or attractive is the software to the user?
- Accessibility – Can the software be used by people with a wide range of capabilities?

These two standards help to guide usability design, as well as usability testing.

7.7. Accessibility

Accessibility testing is considered a subset of usability testing. Accessibility is the degree to which a component or system can be used by people with the widest range of characteristics and capabilities to achieve a specific goal in a specified context of

use. While sometimes targeted at specific disabilities, such as color blindness or hearing impairment, accessibility has generally become a broadened concept to ensure that software works for everyone, with or without disabilities.

Accessibility compliance requirements may drive accessibility testing goals and methods. It is important to clearly identify any pertinent legislation or regulations that apply, such as the ADA (Americans with Disabilities Act) and Section 508, before planning the testing, as specific goals may be defined in the regulations. Section 508 Compliance Testing, an amendment to the United States Workforce Rehabilitation Act of 1973, is a federal law mandating that all electronic and information technology developed, procured, maintained, or used by the federal government be accessible to people with disabilities.

An internationally used reference for accessibility testing is the Web Content Accessibility Guidelines (WCAG). These are widely used guidelines that were published by the Web Accessibility Initiative (WAI) or the World Wide Web Consortium (W3C). There are three conformance levels defined in WCAG: A, AA, and AAA, with AAA being the most difficult to achieve [WCAG].

Accessibility testing tools are available and can be used to quickly scan code to identify compliance issues. These tools will look for such items as text descriptions for all graphic items, usage of colors and font sizing. The tools are frequently updated and provide coverage for different accessibility areas. Research is needed to find the best tool for a particular situation. Accessibility tends to be a specific area of testing expertise because it requires a deep understanding of the regulatory requirements and the tools that will determine conformance to the standards.

8. Testing Connected Devices – 180 mins.

Keywords

connected devices, emulators, lightweight testing, simulators

Learning Objectives for Testing Connected Devices

8.1 Introduction

None

8.2 Connected Devices

LO-8.2.a (K2) Understand the challenges with connected device testing

8.3 Environments and Tools

LO-8.3.a (K2) Understand the differences between simulators and emulators, and which is appropriate for a given situation

LO-8.3.b (K1) Recall how test automation tools can assist in connected device testing

8.4 Quality Characteristics

LO-8.4.a (K1) Recall why understanding user expectations is particularly challenging for connected devices

LO-8.4.b (K1) Recall why requirements for quality characteristics must be identified early in the lifecycle

LO-8.4.c (K2) Explain why usability is an important quality characteristic to consider in connected device testing

LO-8.4.d (K2) Explain why performance is an important quality characteristic to consider in connected device testing

LO-8.4.e (K2) Explain why security is an important quality characteristic to consider in connected device testing

LO-8.4.f (K2) Explain why interoperability is an important quality characteristic to consider in connected device testing

LO-8.4.g (K2) Explain why accuracy is an important quality characteristic to consider in connected device testing

LO-8.4.h (K2) Explain why reliability is an important quality characteristic to consider in connected device testing

8.5 Lightweight Testing

LO-8.5.a (K2) Describe how lightweight testing can be advantageous in connected device testing

8.1. Introduction

The world of connected devices is expanding. What started with mobile phones soon became smart phones and tablets, then blossomed into the Internet of Things (IoT). As software has adapted to be quicker and smaller, testing must also adapt to be quick and lightweight. That said, good testing practices still apply and quality characteristics are still important to the users. However, it is important to remember that users have expectations that their mobile applications and IoT devices will "just work".

8.2. Connected Devices

Devices in the connected world vary from smart phones to refrigerators, from tiny humidity detectors to cars. Anything that is capable of supporting an Internet enabled component is capable of joining the IoT. The same software may be supported on a variety of devices and operating systems (OSs), making compatibility testing more challenging and future-proofing difficult. As the industry leaps forward, backward compatibility is often receiving less emphasis when actually more is needed. There is an expectation from users not to be forced to upgrade in order to take advantage of new features and applications.

Testers can no longer expect to have access to all the devices that will use the software under test. Selecting a representative sample set is a critical part of defining test coverage and risk mitigation. Just because the software runs on a refrigerator allowing the user to increase or decrease the temperature remotely does not mean that all models of refrigerators need to be tested. It is important for testers to understand what exactly is to be tested. Is it the device that provides the connectivity? Is it the response of the target device? Is it the communication between the two? Realistically, it is all of these, but if all refrigerators use the same communication interface with the Internet device, then testing one may be sufficient (i.e., applying equivalence partitioning).

8.3. Environments and Tools

The proliferation of devices supported by Internet appliances has resulted in a plethora of tools and simulators/emulators that can be used for testing. This reduces the need for having a large set of devices available for testing and can greatly speed up manual testing and the development of test automation. Device labs are available for popular devices, such as smart phones, and new simulators/emulators are constantly being created. It is always good practice to verify the results from a simulator/emulator against a real device, but the majority of the testing does not require the more expensive real devices.

Simulators generally provide a standard response to various inputs and “simulate” the interaction with a real device at the software level. Emulators go a step further and “emulate” the responses of the hardware device as well as the software running on that device. For example, testing an application’s interactions with a smart phone’s gyroscope requires an emulator, whereas testing interaction with the device’s email application can be done with a simulator. Similarly, simulators in the form of service virtualization can be used to simulate a web service’s interaction with an application.

Test automation tools are quickly adapting to the IoT and mobile device market. Many of these tools will interact with simulators and, sometimes, even include their own simulators. Simulators for compatibility testing, such as cross-device or even cross-browser, are commonly supported by test automation tools. The tendency is for these automation tools to be more lightweight in features than the traditional client/server or web services tools. Cost is always a consideration and the open source market quickly adapts to the needs for new and purpose-built tools.

Even the best simulators/emulators cannot simulate/emulate everything. There are user actions, such as a complex set of gestures, that must be tested on a real device. It is important for the tester to determine which tests are best conducted with which environment. Generally a mix of simulators/emulators and real devices will yield the most accurate result for the least cost.

8.4. Quality Characteristics

While quality characteristic testing is important for all types of software, there are some unique needs for quality characteristic assessment when dealing with connected devices. One of the most difficult aspects of connected device testing is defining the users’ expectations. The user group for an application can be quite large and the expectations may vary dramatically, even within the target users. It is particularly important that the requirements for the quality characteristics are clearly defined in a measurable way in the requirements or acceptance criteria for the software. With all quality criteria, there is a range of “acceptable”. Defining and documenting this range early in a product’s lifecycle is particularly important for connected devices. By defining the criteria for the quality characteristics at the beginning of the project, all architecture, design and implementation decisions can be made to align with and fulfill those objectives.

While all quality characteristics are important, the following characteristics are particularly important with connected devices:

- Usability
- Performance
- Security
- Interoperability
- Accuracy
- Reliability

8.4.1. Usability

As connected devices become more integral to modern life, users expect "good" usability and learnability. Particularly for downloaded applications for smartphones, users expect software to be attractive, inviting, easy to use, easy to understand, and enjoyable. Users are becoming less patient with software that does not provide a good user experience. This can result in a product that is functionally sound being rejected by the users because it does not provide the expected feedback (e.g., prompts and sounds). Of course, being functionally sound is still the basic building block of software quality, but the ease with which a new user can and wants to work with an application often determines the market share. See Section 7 for more about usability.

8.4.2. Performance

Performance criteria tend to be loosely defined and are often identified when a product does not meet expectations. While this is true for any software product, it is apparent for applications for connected devices which may have limited capability (memory, bandwidth). Ideally, the performance criteria should be defined and captured early in the requirements phases of a project. Defining these criteria, setting up the proper personas and benchmarks, and testing to ensure the criteria are met on the target device, are critical for the success of a connected product. It might be acceptable for a refrigerator to be slow to respond to a request to decrease temperature, but it is unacceptable for a GPS-based guidance application on a smart phone to not respond in time for a driver to make a turn.

Another variable with connected devices is the possibility of a lack of connectivity. This can significantly impact both functionality and performance. A poor connection can result in poor performance. A product may not be able to control the quality of the connection, but it can control its response to poor, intermittent or non-existent connections. See Section 5 for more about performance.

8.4.3. Security

Connected devices are sometimes used for safety-critical applications as well as for financial transactions. Cybersecurity testing is difficult with the shortened development cycles of connected devices, but it is a critical component of a quality product. Architecting, developing, and testing for security must occur throughout the development lifecycle, as there will rarely be enough time at the end of development for thorough cybersecurity testing (and any necessary corrections/changes).

It is important to understand the expected and potential usage of an application running on a connected device. Is the GPS-based guidance application used to get to the mall or to direct ambulances to emergency scenes? Is a web-monitored intruder alarm used to guard the refrigerator from marauding teenagers or to protect a pharmacy? Seemingly simple devices can easily be used in safety-critical situations, necessitating the highest levels of security testing. Like quality, security must be built into the product right from the start.

Connected devices have unique security risks. For example, an attacker can gain access through man-in-the-middle attacks by exploiting Bluetooth vulnerabilities of the device. Using connected devices with public Wi-Fi hotspots can lead to security vulnerabilities that would not be experienced on a controlled and protected network. Because mobile devices are easily carried with a person, they are also easily lost or stolen, which can allow personal information to fall into the hands of a criminal. Connected medical devices present a unique security risk as they have been compromised in past Wi-Fi cyberattacks to gain access to the larger network in a hospital. The list goes on and on. In general, security requirements for connected devices are always expanding, particularly as new vulnerabilities are discovered.

See Section 6 for more about cybersecurity.

8.4.4. Interoperability

Connected devices offer some unique challenges for interoperability. Software is often developed to work on multiple operating systems and a range of devices. For example, a banking application may be intended for use on a wide range of smart phones and tablets. The portability of an application to different devices, the ability of an application to interoperate with other software and the compatibility of the software across different browsers and operating systems are often lumped into the general category of interoperability.

From a testing perspective, this gives a nearly infinite number of combinations to test and those combinations continue to evolve rapidly. Using combinatorial testing techniques can bring this potential set of test targets down to a manageable number. This type of testing tends to be pushed to the end of the testing cycles, when there is enough functionality to support the testing.

The architecture of the product often determines its interoperability capabilities. If the architecture is limiting, the capability of the end product will also be limited.

8.4.5. Accuracy

Accuracy testing targets the ability of the software to provide accurate results for a given set of inputs. Users expect software to be accurate, but there are specific considerations for connected device software primarily because the usage can be so varied. A simple humidity detector developed on a Raspberry Pi may be intended for use by plant enthusiasts to ensure proper watering levels. However, this same detector could be used to ensure an asthmatic child has the proper humidity in their room air. Because the use of a product may not be controllable, accuracy must be ensured for any possible safety-critical usage. This tends to push most of the connected device testing into the safety-critical arena, unless a product can be specifically excluded from a safety-critical use.

Testing in a safety-critical environment requires higher levels of documentation and due diligence to protect the developing organization from potential legal action if something should go awry. This is an area that is open to debate, but from a testing

standpoint, more thorough and better documented testing will help mitigate deployment risk and may help meet regulatory and industry standards. This is particularly challenging in the short cycles of connected device products and is why lightweight testing approaches are critical.

8.4.6. Reliability

As with the higher expectations for usability and performance, users expect complete reliability from their connected devices and software. No one expects to reboot their smart phone and much less their refrigerator. Realistic or not, this is the expectation that must be met in order to capture and retain users. Testing for reliability is difficult and requires test environments that are representative and stable. It also requires time, as reliability tests usually require applications to be observed under continual use for a given length of time to determine the mean time between failure (MTBF). The time required for this testing potentially conflicts with the goal of being quick to market.

Reliability cannot be tested into an application – it must be built in. This means that reliability targets must be set early and reviewed frequently. Reliability testing in the connected device area tends to be limited to discovering whether there are significant problems. More subtle problems or problems that only appear over long usage tend to be discovered in production. Perhaps more than any other area, reliability assurance is a development activity more than a testing activity.

8.5. Lightweight Testing

A common theme has emerged in this section – software has to be built right because there is no time to fix it later. This means the requirements for quality characteristics must be defined and understood by the development and testing teams. In lightweight testing models, such as Agile, the whole team approach helps everyone to review and understand these requirements from the beginning and to revisit and validate fulfillment of the requirements throughout development. Any lightweight methodology is dependent on the engagement of the BAs, product owners, developers and testers throughout the lifecycle to ensure that the product is reviewed and tested as it is being built. Testing cannot be pushed to the end of the process if the schedule time is to be reduced and quality criteria met.

Testing documentation must be as lightweight as possible in this environment. This means that detailed test cases are probably not needed, but repeatable tests are. Testing from decision tables and checklists rather than from step-by-step test cases is a good way to test the implementation of business processes and workflows. The testing techniques can be well-applied to reduce testing documentation while increasing coverage and repeatability. Exploratory testing can help fill the gaps between the testing techniques but should not be the only form of testing, as repeatability tends to be compromised or lost. To adapt to this changing environment, testers need to document the minimum needed for repeatability and to let risk and coverage goals guide the testing.

9. DevOps – 200 mins.

Keywords

behavior-driven development, build, commit, continuous delivery, continuous deployment, continuous integration, continuous monitoring, continuous testing, DevOps, DevOps toolchain, infrastructure as code, test-driven development

Learning Objectives for DevOps

9.1 Introduction

LO-9.1.a (K1) Recall the purpose of DevOps
LO-9.1.b (K1) Recall the benefits of DevOps

9.2 The DevOps Pipeline

LO-9.2.a (K2) Compare the differences between Continuous Integration, Continuous Delivery and Continuous Deployment
LO-9.2.b (K2) Explain the concept of Continuous Testing
LO-9.2.c (K1) Recall the purpose of Continuous Monitoring
LO-9.2.d (K2) Describe the full DevOps pipeline

9.3 DevOps Testing

LO-9.3.a (K1) Recall the concept of testing during planning
LO-9.3.b (K2) Understand the difference between TDD and BDD
LO-9.3.c (K2) Understand how unit testing and integration testing are applied in DevOps
LO-9.3.d (K2) Describe the types of testing that occur during staging and deployment

9.4 The Role of Automation in DevOps Testing

LO-9.4.a (K2) Understand the relationship between continuous testing and automation
LO-9.4.b (K2) Describe infrastructure as code
LO-9.4.c (K2) Explain the DevOps toolchain

9.1. Introduction

DevOps bridges the gap between development and operations by bringing the two teams together for the entire software lifecycle, from development to delivery. DevOps is a cultural shift focused on building and operating at high velocity. It is an approach

that involves activities that are “continuous”: continuous development, continuous integration, continuous testing, continuous deployment, continuous delivery, and continuous monitoring.

Development roles in DevOps include everybody who is involved in making the software and everyone who is involved in running and maintaining production. DevOps asks people working in the development and operations teams to work together, from the beginning of a software development project until it is delivered, breaking down the walls that stand between the different departments.

The benefits of DevOps include:

- Assessing quality at every stage of development and operation, resulting in a high-quality product with fewer defects
- Releasing small increments frequently – If a good DevOps pipeline is set up, it helps to release the product more often, in small increments, which provides early and frequent feedback
- On-time (and possibly lower cost of) delivery, in part due to a better delivery process
- Reduction in vendor and third-party issues
- Faster time to market

There are some misconceptions about DevOps. For example, DevOps does not eliminate any roles from development or operations. DevOps is not a separate team – it is a culture shift. DevOps is not a tool, nor is it the use of fancy tools without a defined process.

9.2. The DevOps Pipeline

A DevOps pipeline is set up in every project to aid all the continuous activities.

Continuous Integration (CI): Continuous integration is the practice of merging all developers’ working code into a main branch of the codebase in a shared repository several times a day. The developers’ changes (commits) are validated by creating a build and running automated tests against the build. Continuous integration ensures that the application does not break whenever new commits are integrated into the main branch of the codebase. If problems are introduced with the new code, they are quickly identified and resolved.

Continuous Delivery: Continuous delivery builds on the CI process in which teams produce software in short cycles, ensuring that software can be reliably released at any time. In addition to having automated builds and tests, Continuous Delivery requires an automated release process and a way to easily deploy applications at any time.

Continuous Deployment (CD): Continuous Deployment (CD) takes the continuous delivery process one step further. A change that passes all stages of the DevOps

pipeline is released to the customer. The deployment trigger is automatic; therefore, the whole release process is automated.

Note that continuous integration (CI) is part of continuous delivery and continuous deployment. Continuous deployment is continuous delivery when the releases happen automatically.

Continuous Testing: Continuous testing is the repeated execution of tests against a codebase. It is the quality gate throughout the DevOps pipeline and increases confidence in the product. The success of continuous delivery or deployment relies on continuous testing. At every stage of the DevOps pipeline, continuous testing ensures that what is being delivered through a stage of the DevOps pipeline is correct and good enough to progress to the next stage of the pipeline.

For continuous testing to succeed, the siloed testing and operations teams should be integrated with the development team. Testing should not be a separate activity. Instead, it should be incorporated as much as possible into the DevOps pipeline. For example, performance and cybersecurity testing should not be an independent activity but should be incorporated into the pipeline. The DevOps pipeline should not be bypassed or blocked for a long time. Use of drivers and stubs are highly encouraged for testing any "what-if" scenarios and dependencies.

Continuous Monitoring: Continuous monitoring allows the DevOps team to constantly monitor the application in a production environment to ensure that the application is performing at an optimal level and the environment is stable. Continuous monitoring helps in diagnosing and fixing errors as soon as they are found.

***

Having described all the "continuous activities" in DevOps, the following is a simple DevOps pipeline that incorporates all these activities. In its simplest form, a DevOps pipeline can have the following stages:

Plan → Code → Build → Staging → Deploy

The planning stage in DevOps is a pre-CI/CD stage where requirements are gathered, tested, and polished. Once a set of well-tested requirements is agreed upon, developers start coding those requirements. As they code, developers commit their changes to a source control repository. Each developer's source code must be accompanied by successfully executed unit tests. The commits trigger a build. Once all the code compiles properly and all unit tests pass, the build is considered successful. If any tests fail, the build is unsuccessful and the commit will roll back to a previous successful commit. This cycle repeats for the next commit.

This continuous integration process ensures that passing tests accompany every piece of code written by the developers, providing testing and code coverage at the unit level. The tested and verified build then moves to a staging environment that mimics the production environment. This is where integration tests, as well as other necessary tests, such as functional, non-functional, performance, security, user acceptance and exploratory tests are executed.

Once the application is thoroughly tested in the staging environment, it is ready to be deployed. Depending on the DevOps pipeline, it can be deployed to a pre-production environment, where the operations team may run more tests, or it can be deployed to the customers.

9.3. DevOps Testing

The way to approach testing in DevOps is to break down the pipeline into a few parts and apply different types of testing in each part. A few of those testing types are described here (this is not a comprehensive list). The simple DevOps pipeline shown above is used as a guide.

Testing during planning: This is pre-CI/CD testing. Testing during the planning stage means gathering requirements accurately and making sure they are testable by creating proper acceptance criteria. Static testing techniques, such as reviews and static analysis, should be used to ensure that defects from work products (especially requirements) are eliminated as much as possible.

Testing during coding and building: During this stage of the pipeline, unit and integration tests are written and executed.

Unit tests are developers' tests where developers are responsible for making sure that each unit being built has one or more unit tests associated with it. These tests are automated and are an essential foundation for all other tests. They are simple tests, easy to write, and fast to execute, but cover all significant paths through the code.

A common way to approach writing unit tests that ensure the testability of the code is to write the unit test first, then write the code to make the test pass, and finally doing some refactoring around that. This process is called Test-Driven Development (TDD). TDD ensures coverage around the code that is being written at the unit level. This supports continuous integration because as the developers commit their code and their unit tests, the CI system will flag any failures or missing tests. The pipeline will stop at this point until the tests pass, ensuring that only tested code will proceed.

TDD works very well at the unit level, but the tests still carry the developer's perspective. There is a need to develop tests based on other stakeholders' perspectives. Therefore, a better approach to testing during coding and building is to use Behavior-Driven Development (BDD). BDD uses the underlying concept of TDD, but instead of thinking about writing a failed test, BDD starts with writing a failed feature test. It then follows the same process as TDD to write code to pass each step of the feature test. This way, the feature is traced all the way up to a requirement.

During the coding and building stages of the DevOps pipeline, static analyzers may be used for code reviews before committing the code as part of a build. Peer reviews

should also be used for reviewing the code to ensure good and consistent coding practices within the team.

Once unit testing is done at this stage, integration testing should be carried out to make sure all developer code is successfully integrated and an integration build is created. The purpose of integration testing is to ensure that there are no issues with combining the different developers' units. The integration build should pass all unit and integration tests.

Testing during staging: A critical factor to remember for testing during staging is that the environment in which testing is to take place must match the production environment. Once there is a stable build, the build can be tested in a staging environment using functional testing (does the system do "what" it should) and non-functional testing ("how well" does the system provide the functionality), such as performance, usability and other types of testing including security.

Performance testing is conducted to identify bottlenecks and any performance degradation within the system. It is done by putting demands on an application under normal and peak load conditions. It measures attributes such as response times, throughput rates, resource-utilization, and identifies the application's breaking point.

One of the main reasons that performance testing is often neglected is because performance tests can take a long time to run and can be resource intensive. DevOps and its pipeline allow the execution of performance tests early. Some performance tests can be run in parallel with unit and integration tests. As functional tests are executed in a staging environment, performance tests for the whole system can be run in parallel.

There is a type of DevOps called DevSecOps, where cybersecurity testing is no longer considered as an afterthought but is an integral part of the DevOps pipeline. Cybersecurity testing is specialized testing and, therefore, a partnership with security experts is needed to perform such types of testing in the pipeline. In DevSecOps, security tools are identified and integrated into the DevOps toolchain and the results are made visible to the whole team.

Testing during deployment: At this stage of the DevOps pipeline, smoke tests for the whole application are performed to ensure that the application runs correctly. Every release also needs to pass acceptance tests on deployments conducted by the operations team.

9.4. The Role of Automation in DevOps Testing

Automation plays a significant role in DevOps and DevOps testing. The Agile movement embraced automation of unit testing, acceptance testing and continuous integration. DevOps ties these with automation of the deployment process, eliminating the boundary between developer and operations automation. Continuous testing

would not be possible without automation; it would not be an efficient process if a large number of test cases were run manually in the DevOps pipeline. Moreover, testing is not the only place where automation is needed in DevOps.

Infrastructure as code: In DevOps, CI/CD requires the automatic deployment of changes to the test and production environments, where these environments are launched automatically as needed. CI/CD also requires an automated way to push the latest build to these environments and, eventually, to the customer's environment. Infrastructure as code includes the process and technology needed to provision and manage environments (physical and/or virtual) through scripts.

Treating infrastructure as code is a key element in DevOps. It benefits both the development and the operations team. Infrastructure as code allows operations teams to get involved in the development process from the beginning. Developers can gain a better understanding of the supporting infrastructure because they can be involved in specifying and understanding configurations for servers, networking, storage, and so on.

The DevOps toolchain: The ultimate goal of DevOps is to streamline development with operations. DevOps does not necessarily require tools to do so, and DevOps is not defined by its tools. However, for most organizations, tools play an important role in automating tasks and ensuring processes run as efficiently as possible.

A DevOps toolchain is a combination of tools that help in development, integration, testing, deployment, delivery, and management throughout the SDLC in a DevOps environment. The DevOps toolchain should include tools from different categories, such as:

- Project planning and management tools
- Requirements engineering tools
- Configuration management and provisioning tools
- Source code scanning (for quality and security) tools
- Build automation and continuous integration tools
- Functional and non-functional testing tools
- Deployment tools
- Release orchestration tools
- Application and infrastructure monitoring and performance tools

It is critical to design a DevOps toolchain that is adaptable to accommodate both changes in team preference, application architecture, quality processes and other technology shifts. In order to maintain an effective DevOps pipeline, DevOps teams should include the right choice of tools as part of their DevOps toolchain.

10. References

10.1. ISO/IEC/IEEE Standards

- ISO/IEC/IEEE 12207:2017
- ISO/IEC/IEEE 15288

10.2. Trademarks

The following registered trademarks and service marks are used in this document:

- AT*SQA® is a registered trademark of the Association for Testing and Software Quality Assurance

10.3. Books

[Anderson00]: Anderson, L.W. and Krathwohl, D.R. (2000) A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives, Allyn & Bacon: Boston MA, ISBN-10: 080131903X

[Firtman]: Maximiliano Firtman, "Programming the Mobile Web", O'Reilly Media; Second Edition (April 8, 2013), ISBN-10: 1449334970

[PMBOK] Project Management Institute, "A Guide to the Project Management Body of Knowledge (PMBOK Guide) – Sixth Edition, 2017, ISBN-10: 9781628251845

10.4. Other References

The following references point to information available on the Internet. Even though these references were checked at the time of publication of this syllabus, AT*SQA cannot be held responsible if the references are not available anymore. AT*SQA is not endorsing any of these sites or their products. The references are provided as a source of information only.

https://techcrunch.com/2013/03/25/ip-oh-my-gosh-all-that-money-just-disappeared/

https://www.reuters.com/article/us-facebook-settlement/facebook-settles-lawsuit-over-2012-ipo-for-35-million-idUSKCN1GA2JR

[NASDAQ] https://www.sec.gov/news/press-release/2013-2013-95htm

National Institute of Standards and Technology. Framework for Improving Critical Infrastructure Cybersecurity. Version 1.1. 2018. https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.04162018.pdf

National Institute of Standards and Technology. Risk Management Framework for Information Systems and Organizations: A System Life Cycle Approach for Security and Privacy. Revision 2. 2018. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-37r2.pdf

[WCAG] https://www.w3.org/WAI/policies/