SYLLABUS

Version 2022

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

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.

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/