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.

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).