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.

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.

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/