Test Approaches Micro-Credential

Syllabus

MICRO-CREDENTIAL

Test Approaches

Table of Contents

Test Approaches

References

General Information

STUDY TIME – 145 MINS.

KEYWORDS

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

LEARNING OBJECTIVES FOR TEST APPROACHES

Introduction

(K1) Recall factors to consider when selecting a test approach

Testing Levels

(K1) Recall the purpose of system integration tests
(K2) Summarize the activities that take place during each of the four levels of testing

Software Development Lifecycle

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

Product Type

(K2) Describe how different product types affect the test approach to be used

Documentation Requirements and Availability

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

Risk

(K2) Explain how risk affects the choice of a test approach

Schedule and Budget

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

Maturity and Ability of the Team

(K1) Recall how the attributes of the team members can affect the choice of test approach

Introduction

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

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

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

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

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

Testing Levels

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

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

The following list of levels is a categorization of the types of testing that need to occur and the logical progression of testing:

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

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

Unit Testing

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

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

Integration Testing

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

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

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

System Testing

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

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

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

Acceptance Testing

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

- **User Acceptance Testing (UAT) –** testing conducted by system users or proxies (e.g., business analysts) for those users in order to determine if the software is fit for purpose. This is normally performed using documented test cases and exploratory testing. The users of the system exercise the system as they would during normal daily use, including cyclical functions such as end of month and end of year. Business analysts will sometimes guide this testing for the users. The goal is for the users to "accept" the system based on their evaluation of whether the acceptance criteria have been met. The users bring a unique viewpoint that may be missed by testers who are unfamiliar with all aspects of system usage and the real-world conditions that users encounter.
- **Operational Acceptance Testing (OAT) –** testing conducted by system operators to determine if the software will work in the production environment when fully deployed. Ideally, this testing is conducted in a staging environment that is an exact replica of production; where that is not possible, the environment should be as close as possible to production. System operators use this testing to ensure that the software will work properly with load balancers, firewalls and other production equipment, as well as with

production processes such as backups. Testing rollout and rollback plans are often part of this as well.

- **Alpha Testing –** testing conducted at the development site, but not by the developers or testers who have been working on the project. This testing is sometimes called "internal acceptance testing", meaning that the testing is conducted within the organization, but is not exposed to external users. Training groups and support groups within the organization are often used for this type of testing.
- **Beta Testing –** testing conducted at a customer (or potential customer) site using the customer's data and network environment. This testing is usually conducted by the customer themselves, although they may have some assistance from the testers or developers to ensure the test coverage is adequate. The goal of this testing is to determine if the software is fit for purpose in the real production environment without fully releasing it to everyone. Feedback from beta testing may result in further internal development and/or testing prior to the full production release.

Software Development Lifecycles

Sequential Models

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

In a pure waterfall model, testers usually are not engaged in the SDLC until the software is completely built and the developers have completed their unit testing. In a pure V-model, testers are engaged early to review requirements, design documents and to prepare the testware (e.g., test plan, test cases) prior to receiving the code to test.

The advantages of sequential models include:

- The requirements are considered to be stable throughout the project
- Test automation can start at the beginning of testing because the software will not change
- In a waterfall model, the test team is only involved from the moment the code is complete, freeing them for other tasks or other projects
- In the V-model, the test team is involved with reviewing all the documents produced by the business analysts and developers

(requirements, high-level and low-level design documents) and can provide input on each of these, thus engaging with the project sooner and having input that can influence the quality of the product

- In the V-model, there is generally more time available to apply structured testing (e.g., prepare the test documentation, including test cases)

The disadvantages of these models include the following:

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

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

Iterative Models

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

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

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

The advantages of the iterative models include the following:

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

The disadvantages of the iterative models include the following:

- Too frequent changes in direction can result in little or no progress
- Lack of detailed documentation means reliance on communication, which may be difficult for a team that is dispersed
- If cultural issues in an organization are significant, people may not be able to work effectively as a cross-functional team
- Lack of detailed documentation may make the model infeasible for some products, particularly those with regulatory or safety-critical aspects
- Test automation is mandatory to avoid manual testing time becoming increasingly long due to the larger scope of regression testing as iterations progress
- Rigid adherence to the process can result in a significant learning curve for a team Iterative models have been around for many years, long before Agile was defined. These models have worked successfully across a wide variety of software projects and continue to be the most dominant models in the industry.

Hybrid Models

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

SDLCs have built-in safeguards to ensure that necessary steps are completed. Picking and choosing the "best" parts from different SDLCs is likely to result in weaknesses being exposed. For example, if an organization were to pick an Agile SDLC, but also chooses to work without defining acceptance criteria for stories, there is a gap created in the validation aspects of the project. Similarly, if an organization were to pick a waterfall model, but decides to use stories to document the requirements, the concept of completed and well-defined requirements before the start of coding is violated. This may result in a product that is incompletely developed or in a product that necessarily must change as development progresses. This results in a longer development time and a compression of the testing schedule.

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

Product Type

Another consideration when determining the testing approach is the type of product that is being developed. A mobile application that is expected to last for six months requires a different approach than software that will control the navigation of a fleet of aircraft. In general, the longer the software will stay in production and the more critical the functionality of the software, the more formal the approach.

A formal approach may dictate the lifecycle model, the level of documentation required and the test techniques to be used. Similarly, a short-lived application that is used to provide a game interface for idle travelers may be best served by a lightweight approach with an Agile lifecycle, minimal documentation and only exploratory or acceptance testing.

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

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

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

Documentation Requirements and Availability

In software testing, documentation has two general categories: documents required to properly test the product, and documents required to demonstrate the testing has been completed. The test approach is heavily influenced by the availability of documentation and the requirement to provide documentation from the test process. If there is little or no documentation regarding what the software is supposed to do or how it will do it, the tester is forced into an approach that includes some amount of exploring the software to understand what it is doing. Creating detailed test cases may not be worthwhile since the testing will be occurring while the research is being conducted to document the test cases.

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

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

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

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

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

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

Risk

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

Risk is a significant factor in determining the best test approach. Higher risk projects generally require more formal approaches with more complete documentation. Lower risk projects can work with a lighter approach and may require little or no documentation.

Using risk prioritization, commonly called risk-based testing, on every project is a strong approach and helps to prioritize and define all testing activities, including the following:

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

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

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

Schedule and Budget

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

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

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

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

Maturity and Ability of the Team

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

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

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

References

ISO/IEC/IEEE Standards

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

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

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

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/

AT*SQA

MICRO-CREDENTIAL

Test Approaches

www.atsqa.org