AT*SQA

Agile Software Testing Methodologies Micro-Credential

Syllabus

Revision History

Version	Date	Remarks
First Draft	25-Jan-23	Draft compilation of separate modules

Table of Contents

Acknowledgements

This document was produced by a core team from the AT*SQA Syllabus Working Group – Agile Syllabus:

Authors:
Judy McKay

Reviewers:
Randy Rice
Earl Burba

The core team thanks the review team for their suggestions and input.

0. Introduction to this Syllabus

0.1 Purpose of this Document

This syllabus forms the basis of the AT*SQA certification for Agile Software Testing Methodologies. 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 Why Agile?

The Information Technology (IT) world changes almost continuously as new technologies, methodologies, and techniques are created. Some of these are adopted as-is, some are discarded, and others are adapted for various uses. The Agile lifecycle methodology has been widely embraced in principle, but in practice the methodology tends to be modified. In some cases, this modification makes sense to adapt the methodology properly to fit a particular situation, but in other cases the concept of “Agile” remains only in its name, not in the practice. Because Agile is a pervasive methodology in its various forms, it is important for all software testers to be familiar with it - in the base concepts, the pure form, and the various modifications.

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. In an Agile environment, each team member is responsible for contributing to the quality of the product, via the implementation of and participation in quality practices. Software testing, in this environment, is an assessment of the quality of the software that has been built.

This syllabus focuses the Agile methodology from the viewpoint of the software tester. This includes looking at how an effective Agile team works, the basic rules of an Agile methodology, and how a software tester fits into this environment. This syllabus is intended for use by all members of an Agile team as well as anyone managing an Agile project. This includes product owners, business analysts, developers, software testers, project managers, scrum masters and anyone else who is involved with the development and testing of a product in an Agile environment.

0.3 Syllabus Structure

This syllabus has been constructed to be tool agnostic but tools will be discussed as they are an important part of implementing an effective Agile project. When tools are referenced, this is not a recommendation for the use of a particular tool, but examples of commonly used tools are employed to help clarify points.

There are no pre-requisites required for this certification. The certification can be achieved in two ways - via passing the assessments for all three micro-credentials that compose this syllabus, or by taking the one certification exam that covers all three micro-credentials. 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.

0.4 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 Appendix A.

The Learning Objectives for each chapter are shown at the beginning of the chapter and are used to create the examination for achieving the Agile Software Testing Methodology 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 Agile Concepts

Keywords

acceptance criteria, acceptance testing, Agile Manifesto, Agile Principles, Agile software development, alpha testing, behavior-driven development (BDD), beta testing, definition of done (DoD), DevOps, DevSecOps, embedded iterative, epics, incremental development model, integration testing, iterative development model, Kanban, minimum viable product (MVP), planning poker, product backlog, production implementation verification (PIV), Product Owner (PO), retrospective, scrum, software development lifecycle (SDLC), sprint, stand-up, story grooming, systems integration testing (SIT), system testing, test-driven development (TDD), unit testing, user acceptance testing (UAT), user story, V-Model, Waterfall, whole team

Learning Objectives for Agile Concepts

1.1 Goals of Agile Methodologies

LO-1.1.a	(K2) Compare the traditional lifecycle models to the iterative lifecycle models
LO-1.1.b	(K2) Explain the differences between an Agile model and a generic iterative model
LO-1.1.c	(K2) Explain how early engagement from testers impacts the delivery speed for a project
LO-1.1.d	(K2) Summarize how Agile provides a better connection to the end user
LO-1.1.e	(K2) Explain the advantages of using small deliveries of functionality

1.2 The Basics of Agile

LO-1.2.a	(K2) Explain the main concepts of the Agile Manifesto
LO-1.2.b	(K2) Explain the principles of Agile

1.3 Making Agile Successful

LO-1.3.a	(K2) Explain the Whole Team approach
LO-1.3.b	(K2) Summarize why team culture is important in Agile projects
LO-1.3.c	(K2) Explain how an embedded iterative model works in the Agile context
LO-1.3.d	(K2) Summarize the requirements for an ideal Agile project
LO-1.3.e	(K3) For a given scenario, select the best lifecycle model
LO-1.3.f	(K2) Explain how the potential for scope change affects testing

1.4 Risks with Agile Methodologies

LO-1.4.a	(K2) Understand the various risks to Agile projects
LO-1.4.b	(K2) Explain why it's important for testers to ask questions
LO-1.4.c	(K2) Summarize the issues that can occur when the MVP is delivered

1.5 Agile Methodology Types and Usages

LO-1.5.a	(K2) Summarize the differences between Scrum, SAFe®, and Kanban
LO-1.5.b	(K2) Explain how a Kanban board can help the test effort
LO-1.5.c	(K2) Explain how DevOps is a cultural philosophy
LO-1.5.d	(K2) Summarize the best practices for DevOps
LO-1.5.e	(K2) Explain the difference between incremental and iterative development

1.1 Goals of Agile Methodologies

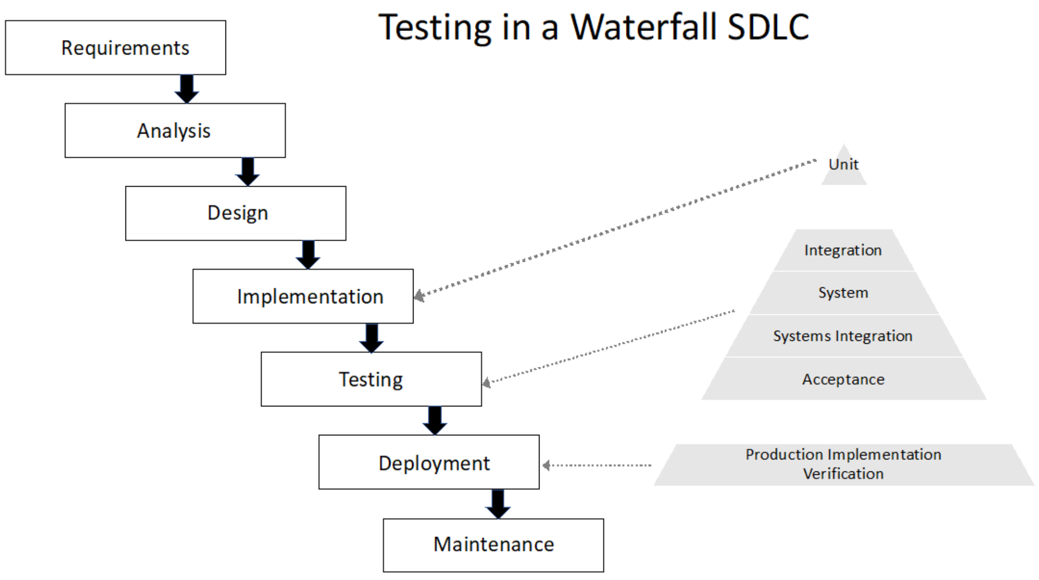
Many organizations turn to Agile in an effort to rectify problems with their current methodologies. Sadly, Agile is not the magic solution that will fix poor processes, badly managed lifecycles and ineffective teams. It is important to remember that a good team can make any lifecycle effective, efficient, productive, and able to meet the needs of the project. Similarly, an ineffective, poorly formed, and internally incompatible team will fail with any lifecycle. While some lifecycles require less dependence on individual responsibility and self-management, people are what makes a lifecycle work or fail.

There are several Agile methodologies or approaches. Agile can be considered a lifecycle model, covering the entire Software Development Lifecycle (SDLC). It is also sometimes referred to as a project framework. In this syllabus, the various forms of Agile are either referred to as methodologies or just by the generic term, Agile.

1.1.1 Common SDLCs

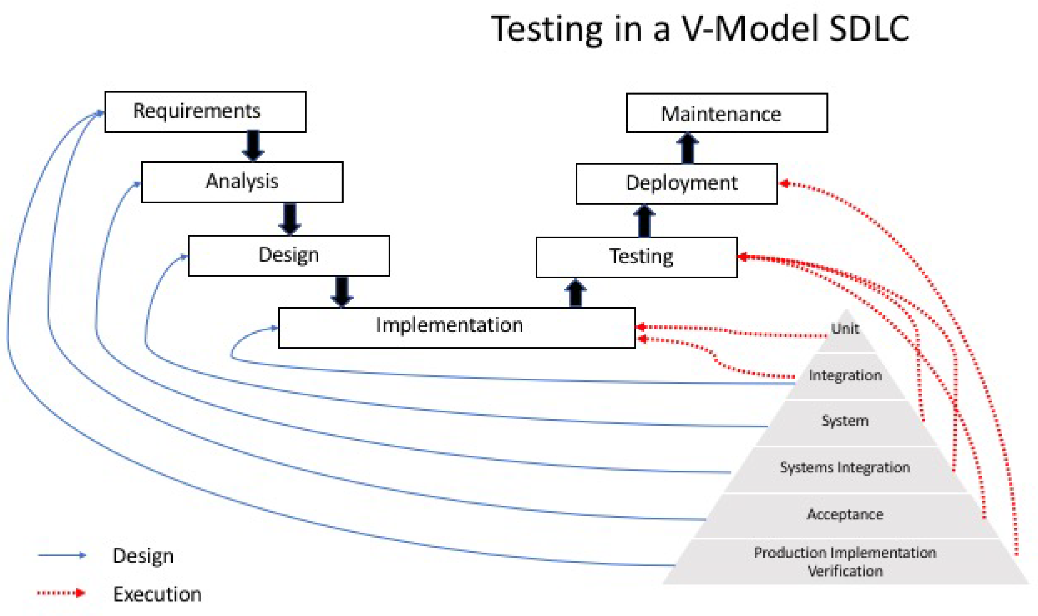
As a quick recap of the common SDLCs, the following is a short explanation of each. This background is needed to understand the problems that Agile is attempting to fix, but also to understand how these methodologies work.

Waterfall - The Waterfall lifecycle model is characterized by a sequential flow of activities with discrete steps in the flow. These are commonly shown as follows:



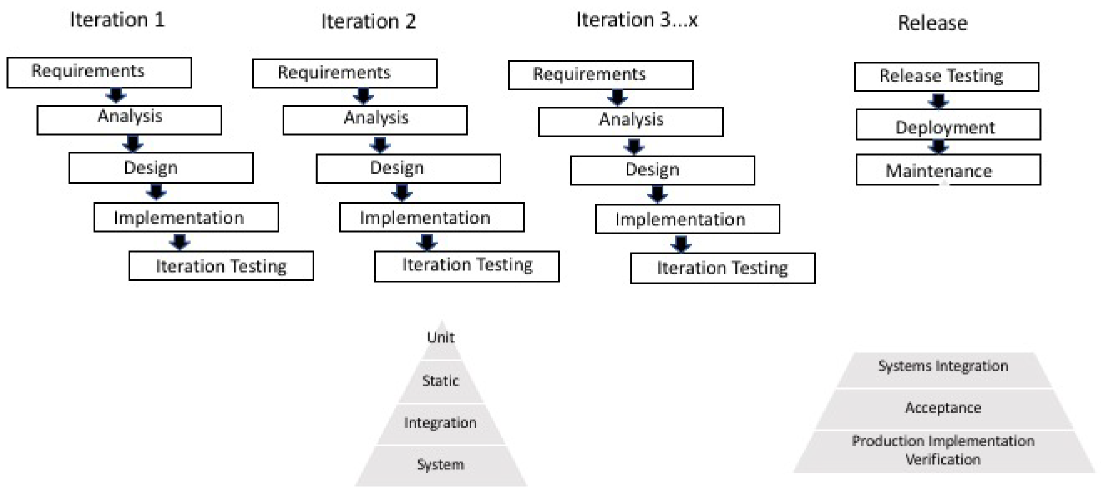
The testing pyramid shows the types of testing that occur in the steps of the process. Formal testing is not actually engaged until implementation is complete as the developers are expected to conduct their own unit testing. It's also important to note that Acceptance testing, such as alpha, beta and User Acceptance Testing (UAT), only occurs after all other aspects of development and testing have been completed. The frequently cited flaws with the Waterfall approach include the extensive time spent defining requirements, the lack of user engagement after the initial requirements gathering, and the engagement of testing late in the lifecycle so that defects are only detected, not prevented.

The V-Model, also a traditional and sequential lifecycle, addresses one of these flaws by engaging testing early in the lifecycle to review documentation such as requirements and design documents and to build test collateral as the software is being built. It still has the flaw of not engaging with the user after the initial requirements gathering but by including early testing participation, it does provide the opportunity for defect prevention. While preparation of testing collateral, such as test cases, early in the lifecycle should save time during test execution, this only happens if the software is built in accordance with the requirements. If the requirements are not strictly adhered to, the resultant software and tests will not match, requiring additional test collateral maintenance.



Iterative models attack several of these problems by breaking the software design / development / testing into iterations. This provides a means to divide a project into smaller projects, providing the potential for early delivery of some pieces, better engagement with the user, and early test involvement.

Testing in an Iterative SDLC



In Iterative software development, the duration of the iterations is based on the software being implemented, the logical grouping of features, the overall release / deployment methods, test environments, and other factors. With Iterative software development there is still commonly an overarching architecture or design that comprises the entire product. This is then divided into logical iterations that provide efficient development and testing. Iterations are often determined based on sets of functionality. For example, for an ERP system, one iteration might include displaying item pictures on a Point-of-Sale (POS) device and another might be allowing two-for-one discounting. When the software must be integrated into other legacy functionality (such as using an existing financial system) integration layers may also be required, both temporary and long term.

In some cases, the release of a product may be the result of many iterations and can occur as a single major release. An example of this would be commercial software, such as tax preparation software, which is released once a year.

Agile is a form of the Iterative methodology. It seeks to release small pieces of functionality in short iterations. Agile goes beyond standard Iterative in that it doesn't require the definition of the product as a whole. Agile is designed to allow a product to be developed and grow as it is being built rather than defining it in totality before development starts. This allows the maximum flexibility of changing requirements based on early and frequent user feedback, but also requires a flexible budget and schedule to allow for the potential scope creep or even rescoping.

Agile was created to address particular flaws in the traditional sequential lifecycles such as Waterfall. The goals of Agile can be grouped into the following four categories:

- Deliver useful functionality to production faster
- Provide a better connection to the user
- Approach the release as small individual projects
- Spend less time on unnecessary overhead

Each of these goals is discussed in the following sections. These goals are tightly interwoven, making it difficult to achieve one without the others.

1.1.2 Deliver Faster

One of the major flaws with the traditional models is the speed with which software is delivered to the users. In traditional models, it is not unusual for there to be a flurry of activity working with users to discern the requirements. This activity is usually carried out by Business Analysts (BAs) who have a close connection with and understanding of the needs of the users. For an ERP system, this could mean documenting and understanding the various business processes. For a web application, this could mean analyzing the market and understanding the needs and desires of the potential users of the application.

Requirements gathering can take a considerable amount of time. The size of the project determines the amount of time between requirements gathering and product delivery. The longer this time becomes, the more likely the requirements have changed. The software may be delivered months (or even years) after the requirements were gathered as a *fait accompli* when it's too late to make changes even if the requirements have truly changed.

The late engagement of the software testers in the sequential models also leads to longer release times. Testing later will always be slower than testing earlier unless the software is perfect. A simple algorithm, similar to the cost of quality models, can be used to determine how much time a defect requires to be discovered, diagnosed and repaired. For example:

- Defect found in requirements reviews - 0.25 hours
- Defect found in unit testing - 0.5 hours
- Defect found in integration testing - 3 hours
- Defect found in system testing - 10 hours
- Defect found in systems integration testing - 20 hours
- Defect found in acceptance testing - 30 hours
- Defect found in production - 40 hours

While individual project times may vary, the relative difference in time required will remain the same - it takes longer and more peoples' time to fix a defect the further it escapes through the lifecycle. Simple math tells us that if defects are found and resolved early, less time will be required.

Agile emphasizes building quality software, engaging in software testing early and in building test automation so that quality gates are created, ensuring only software meeting those quality requirements is deployed to the next stage. It's important to note here that Agile projects do not normally define the levels of software testing, but, even if they are not clearly defined, they still exist. All software should pass through these levels - skipping levels only increases the chance of an escape which will cost more time when it is finally caught. For example, projects that perform only unit and acceptance testing are destined to have each defect cost them a considerable amount of time - time that could have been used to develop more features.

Agile also favors building software over documenting what it should do. This can save significant time in building the requirements documents, going through approval processes, and updating the documents as needed. Agile prefers a "necessary minimum" approach to documentation, for example using Epics to define a set of functionality rather than an overarching requirements document that defines all the functionality. This approach allows coding to start sooner, resulting in functionality being delivered to the user faster.

1.1.3 Better Connection with the User

Because requirements definition is done in small pieces, frequent interaction with the user or the user representative is possible. The user representative, usually called the Product Owner (PO), reviews the requirements (usually documented as User Stories), provides input regarding prioritization, and receives demonstrations of the stories as they are completed. This allows early feedback and minor changes to be accommodated in the existing stories as well as those to come. For example, a PO may provide feedback on a web application indicating that the UI is too difficult for the target users. This feedback will affect multiple stories but can fuel the necessary changes that would be more costly to make after the software has reached production.

The concept of early and frequent feedback is critical to the success of an Agile project. It corrects one of the major flaws of sequential projects where the requirements have changed during the time it took to develop the software. This added engagement also helps to resolve small questions that arise during development and testing, such as “how will the user actually use this feature” or “should this button be moved?”. These questions are easily answered by the PO who is always available to the project. In a sequential project, the BA who wrote the requirements may be long gone before these questions arise, leaving the developers and testers to negotiate for the resolution.

1.1.4 Small Deliveries

By working in iterations, called Sprints in the commonly used Scrum methodology, the team is able to produce small deliverables on a regular cadence. This provides predictability to the users and allows the PO to coordinate deliveries with the release and production teams. These small bits of functionality allow the team to pivot quickly if the users have different needs or if new opportunities are discovered. This also allows the team to respond quickly to changes made by competitors. In some competitive markets, such as online hotel booking, competitors are continuously making changes hoping to secure a higher percentage of the market. Being able to respond to these changes quickly is a competitive advantage.

Working with small deliveries allows a team to provide working software faster to the users, assess the acceptance by the users, identify quality issues quickly, and to adjust future work based on feedback. While this can result in some churning, it occurs on a relatively small scale unlike the upheaval that would occur in a Waterfall project if the requirements substantially changed.

1.1.5 Less Overhead

Overhead is commonly considered to be a burden on a project. While some is necessary, any reduction in unnecessary overhead is a welcome change to any project. Because Agile teams tend to be small and somewhat self-managed, there should be less management overhead. Requirements for progress reporting should be reduced or even eliminated because progress is shown in the deliverables rather than a percentage of completion for a section of code.

Meetings are a standard time and morale killer for projects. Agile meetings are intended to be short, inclusive and goal-oriented. Daily stand-ups are called “stand-ups” because the intention is that they are short enough for everyone to stand through. As soon as chairs are needed, the stand-up is wandering off course and taking too much of everyone’s time. Planning meetings are similarly targeted and interactive. Group estimation using tools such as planning poker keep everyone engaged and actively participating.

Documentation is also often considered to be overhead: unnecessary and time-consuming. Agile projects focus on “just enough” documentation, ensuring that the proper amount of information is recorded, but no more. This helps to decrease reading time as well as maintenance time. Although not one of the tenets of Agile, in practice the use of tools to capture information and to map traceability and dependencies can considerably reduce the documentation burden on all members of the team. The effective use of tools, including test automation, help to reduce overhead, speed release, and improve quality.

1.2 The Basics of Agile

The Agile methodology is defined by the Agile Manifesto and the Agile Principles.

1.2.1 The Agile Manifesto

The Agile Manifesto is the encapsulation of the concepts behind the Agile methodology. It is as follows:

“We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.” (Beck o. , 2001)

It is important to notice that the items on the left are preferred over the items on the right, but they do not advise eliminating the items on the right in all circumstances. This is sometimes lost when people are implementing Agile. In general, the manifesto is saying that it’s better to put the emphasis on the items on the left of the list whenever possible. Each of these items is addressing in the goals discussed earlier: delivering faster, engaging with users, delivering working software rather than documentation, and eliminating overhead (processes, contracts, extensive documentation) wherever possible.

1.2.2 The Agile Principles

The Manifesto is further expanded in the Agile Principles. These have formed the basis for processes such as stand-ups, retrospectives, short iterations, etc.

“Our highest priority is to satisfy the customer
through early and continuous delivery
of valuable software.

Welcome changing requirements, even late in
development. Agile processes harness change for
the customer's competitive advantage.

Deliver working software frequently, from a
couple of weeks to a couple of months, with a
preference to the shorter timescale.

Business people and developers must work
together daily throughout the project.

Build projects around motivated individuals.
Give them the environment and support they need,
and trust them to get the job done.

The most efficient and effective method of
conveying information to and within a development
team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development.
The sponsors, developers, and users should be able
to maintain a constant pace indefinitely.

Continuous attention to technical excellence
and good design enhances agility.

Simplicity--the art of maximizing the amount
of work not done--is essential.

The best architectures, requirements, and designs
emerge from self-organizing teams.

At regular intervals, the team reflects on how
to become more effective, then tunes and adjusts
its behavior accordingly." (Beck e. a., 2001)

The Agile Manifesto and Principles are a set of rules and guidelines. When implementing an Agile project, it is important to consider and implement all of these and to ensure that processes and tools are used to support these processes. When organizations pick and choose from these items, they end up in a situation where they have taken unwitting and sometimes misguided shortcuts. In order for Agile to work, and it can, abiding by all these guidelines is critical. Most, if not all, Agile failures can be mapped back to having not implemented one or more of these principles.

1.3 Making Agile Successful

The statements in the Manifesto and the Principles seem like common sense, but actual implementation in a business environment is difficult and adherence to the Principles requires discipline and maturity. It should also be noted that not all projects and teams are suitable for an Agile approach. The following sections discuss some of the key components of a successful Agile implementation.

1.3.1 Culture

Proper implementation of Agile requires the Whole Team approach. This means that each member of the team is responsible to create a high-quality product, to work effectively with each other, and to meet the needs of the user / customer. Everyone on the team is equally responsible for the quality and success of the product being developed and each member of the team must bring their own skills and apply those for the betterment of the team and its products.

A successful Agile team works well collaboratively, with an open exchange of ideas in an atmosphere of mutual respect. This means that when one's own tasks are completed, that team member looks for ways to help others who may need some assistance.

Team culture is important to create an environment of success. All team members must believe in and be committed to achieving the stated goals of the project. Because team members contribute to the estimates and determining the workload, there must be 100% buy-in that the work can be done. If not, that needs to be discussed with the whole team. There is no individual success in Agile - only team success.

1.3.2 Project Suitability

Not all projects are well suited for Agile. As will be discussed in more depth below, the actual implementation of Agile has some limitations. Not all projects can have requirements that are not pre-set. Not all schedules and budgets are flexible. Not all defined scope can be changed. When selecting an Agile methodology, it's important to pick an appropriate project and team.

Project restrictions usually occur around schedule, budget and scope. In some cases, it is possible to embed an Agile approach within a traditional lifecycle. This is sometimes called an "Embedded Iterative" model. In this case, there are still defined, agreed and signed off requirements. There is a set schedule, scope and budget. Within those parameters, the team can still work in an Agile mode for the development, testing and delivery. This is commonly seen in industry where the team wants to use an Agile approach, but the parameters of the project require certainty regarding the requirements and end product. This still gives some of the benefits of a full Agile project with frequent user feedback via a PO, close collaboration, short iterations and frequent deliveries.

Ideal Agile projects have latitude for change in the scope, budget and schedule. This allows the requirements to evolve via story grooming and PO input. Most projects do not have the flexibility needed for a full Agile methodology, but can use the embedded model to achieve some of the gains from the Agile approach.

1.3.3. Engagement with Users

All good software development teams want engagement with the users of their software. This is needed during requirements gathering, design, implementation, testing and deployment. Agile provides a specific requirement for this engagement and this is often manifested through the Product Owner who is responsible for representing the user throughout the SDLC. The validity of the PO's input is directly related to their relationship and knowledge of the users, and their willingness to ask questions when they don't know the answer.

It may not be possible for one person to adequately represent all the users for a product. In this case, multiple representatives may be needed, or frequent demos to sets of users may be required. Because Agile projects have lighter documentation, usually captured in User Stories, the team needs access to people with answers when fleshing out the stories, defining acceptance criteria and developing test scenarios. Without this vital information, and without a single source of truth (SSOT) requirements document, the project risks implementing incorrect or insufficient functionality.

Another point that is sometimes missed in Agile projects is that the users or their representatives must be readily available. Questions arise throughout development and testing; the more quickly these are answered, the faster everyone can get back to work.

1.3.4 Individual Responsibility

One differentiator of Agile from the traditional methodologies is the concept of individual responsibility. Because Agile requires quickly responding to change, sharing tasks, and communicating effectively, each individual must accept responsibility not just for their tasks, but for the success of the team as a whole. While an individual may be an expert in their area, such as development or testing, this does not limit their responsibility within the team. In order to be an effective team, each individual must leverage their skills and time as effectively as possible - for the good of the team.

Self-management is an important tenet of Agile. Individuals should expect to be responsible for completing their own work and accountable for the outcome, but they must also be responsible for estimating their work, scheduling their tasks, performing their work, and then helping others as needed - with the overall goal being to make the team successful. Managers tend to be external to the team, providing administrative support, but not day-to-day direction as that is handled within the team.

1.3.5 Respect

In order for an Agile team to work successfully, they must respect and value each other. Each individual must both earn and give respect to their team members. Respect is a requirement of trust - Agile team members must trust each other. Trust extends to being able to ask for help, admit mistakes, and to work together to find solutions. Mistakes will happen, but the team must have an environment of safety, trust and respect so that mistakes can be identified, rectified and prevented.

Respect does have to be earned and each team member is responsible for performing in a way that will be respected by the rest of the team. Teaching, supporting and helping each other are key aspects of earning respect. Consistently exhibiting professionalism, competence, and a willingness to learn, supports the team environment and results in the best performance from each individual.

Arrogance has no place in an Agile team. Arrogant team members will tend to isolate themselves, provide little opportunity for feedback or development, and will eventually divide the team into compartments rather than a working whole.

When Agile teams are formed, it’s important to ensure there is enough seniority to support and guide the less experienced team members. An environment of continuous learning and growth is critical to growing the team, promoting the individuals, and bringing the best skills to a project.

1.3.6 Maturity

Similar to the requirements for responsibility and respect, successful Agile team members also demonstrate individual maturity. This is not a factor of age or even experience, but rather an attitude that showcases competence, calmness and camaraderie. Immature team members may try to draw attention to their accomplishments, may demonstrate arrogance and will tend to divide rather than unite the team. Mature team members ensure everyone is working effectively by providing support and guidance while also being individual contributors.

Maturity is often demonstrated in stressful situations where the team is behind schedule, has hit a technical difficulty or has experienced outside issues (such as unexpected illnesses). By taking a mature approach, the team can work together on finding solutions to problems and moving forward despite obstacles. In self-managed teams, the maturity aspect is particularly important. There is an expectation that problems will be resolved internally rather than escalating to management.

1.3.7 Time to Experiment and Evolve

As stated in the Agile Principles, the team should be able to welcome changing requirements. Anyone experienced with software development understands that late changes often extend the time required to complete a project. In order for an Agile project to be successful, there must be adequate schedule time and budget to allow for scope changes.

In testing, because scope may change, it is important to build flexibility into the tests, particularly test automation. Tools used must support easy changes and updates. Tests must be traceable to the requirements so that changes can be assessed and addressed easily. Test automation should be at least keyword-driven to allow encapsulation of functionality within the test automation's scripts. These techniques are suited to any lifecycle but are particularly important in an Agile project because change should be expected.

1.4 Risks with Agile Methodologies

The risks to Agile projects are primarily the inverse of the success factors. These are very real risks and often occur in the real world. Agile projects can be fragile because of the high dependency on the interworking of a set of people with diverse skills and interests. The following risks are those most commonly seen in Agile projects, but this list is not exhaustive.

1.4.1 Immaturity

Immaturity is a risk to an Agile project in several ways. Immature people will not work well in a self-managed environment where individual responsibility is critical. Immature processes will also cause failures because adherence to the Agile processes is critical to the success of an Agile project. If a team is not willing to do planning sessions, stand ups, and other Agile meetings (sometimes called "ceremonies"), the project is at risk because these are critical times to work together as a team.

1.4.2 Silos

Agile teams, by nature, tend to become siloed because they are internally focused. They have their epics / stories / requirements and they are concentrating on implementing those capabilities in the agreed time. This is fine if one team is producing the entire product, but in the case where there are multiple teams working on a large project, the teams must work together to ensure functionality is delivered at the right time to enable other teams to use and test it together with their functionality.

1.4.3 Lack of Quality Ownership

Although it should not happen according to the whole team approach, it does sometimes happen that some team members feel that quality is the responsibility of those conducting the final testing. Quality has to be built in, monitoring throughout the SDLC, and assessed - and this mindset must be embraced by the entire team with quality-oriented tasks shared throughout the team. Allowing quality to become the responsibility of a small part of team results in breaking the team culture, not respecting all the team members, and points to a lack of maturity and knowledge of how to develop good software.

1.4.4 Insufficient Communication

Because there is not a deep source of information in the project documentation, communication between team members is critical for the success of the project. This means frequent, open, and honest communication where questions are answered and team members are treated with respect. If a tester isn't sure what to test for a story, they need to be comfortable asking the rest of the team because it's likely the PO and the developer have their own view of how the software should work. In general, if a tester doesn't know what to test, a developer probably didn't really know what to develop. Getting these types of questions out early, preferably during story grooming, allows the entire team to understand the requirements and how the assessment of the implementation will be conducted.

Although an Agile team is ideally co-located, that is often not possible due to geographical limitations or even external forces such as pandemics. This means that communication must be considered, both in terms of message and media. For example, if in-person, casual communication is not possible, it may be effective to have an open online chat where everyone can easily converse as needed. Each individual has a responsibility to monitor the chat and offer assistance and support as needed. Open communication cannot be assumed to be a default behavior, it must be consciously undertaken.

1.4.5 Insufficient Documentation

While Agile projects prefer working code over documentation, that does not mean that no documentation is needed. If user stories are used to define the requirements, those stories must contain sufficient detail for the developer to know what to develop and the tester to know what to test. Without this level of detail, side conversations will occur and agreements will be made, but won't be documented. Clarifications are often needed but not documenting these clarifications will result in the same questions being asked by multiple team members, potentially with differing answers.

1.4.6 Missing Teamwork and Camaraderie

The Agile methodology requires a high level of individual responsibility and maturity as well as respect for the other team members. When any of these are lacking, the team concept is at risk and the necessary camaraderie that allows the team to work together effectively, even through difficult times, will not be present. A strong Agile team works well together. A broken Agile team consists of people exhibiting arrogance, unwillingness to help, and a me-first attitude rather than a team-first attitude. As mentioned at the beginning of this syllabus, any good team can make any methodology work. Some methodologies are better able to cope with a sub-optimal team, but Agile is not one of these. A high-performing, well-integrated team is critical for success in an Agile project.

1.4.7 Schedule and Budget Explosion

Generally, when a schedule extends, more budget is required. It logically follows that if the requirements for a project are continually changing, more work will be required which will eventually extend the schedule and cost more money. While requirements changes are to be “welcomed”, they come at a cost that must be understood at the project management level. Agile projects require more latitude to increase the schedule and use more budget than projects with a strictly defined scope. Because of the tendency for “scope creep”, it is not uncommon for Agile projects to resort to releasing their Minimum Viable Product (MVP) as their final product because the money and time have run out. An MVP is not intended to be the end point, but rather a point in the progress toward the final delivery.

1.4.8 Disappoint Instead of Delight

Close interactions with the end user (or their representative) should enable delivering a product that will “delight” the user. Too frequently, the user is disappointed because although they were allowed to ask for more functionality, there wasn’t the time and money needed to deliver it. Instead, the concentration is on the MVP which is, by definition, the minimum viable product. That is more likely to disappoint than delight. If there is money to add more functionality, the schedule will likely increase which may frustrate the users who are anxious to get the new features. In poorly managed Agile projects, the users tend to love the engagement during the development but are quickly frustrated by receiving a “minimum” product, later than promised.

1.4.9 Two-weeks is not Enough

In some Agile methodologies, such as Scrum, there is an emphasis on very short iterations, often only two-weeks long. The idea here is that this will provide pieces of functionality faster to the users, but in reality, it’s very difficult to divide meaningful functionality into pieces that can be created in two-weeks. As a result, there are pieces of work, usually defined as Epics, that span multiple iterations. This may result in little visible functionality being delivered to the users at the end of each iteration.

It’s important for a team to determine their optimal iteration length. There is significant overhead in an iteration - planning, grooming, regression testing, delivery activities - that may result in an iteration having few actual coding and testing days. In this case it may make sense to have longer iterations so there is less disruption to the team’s primary purpose of producing quality software.

1.4.10 MVP is not Viable

By definition, the MVP should be the minimum viable product, meaning that it has useful functionality that can be used to complete all the required activities provided by the product. While this is fairly easy to define in a small application, it can be very difficult to define in a complex multi-system integration project such as an ERP because of the inter-relationships of the various components. If the MVP will be a delivery goal, it must be carefully defined right from the start of the project with good input from users who understand the entire project.

An MVP can end up requiring additional interfaces being developed to “bridge” between applications. This code is normally considered “throw away” because it will not be needed with the final product, but it must be reliable as it will remain in production until the full product is delivered.

Because Agile projects tend to expand in scope as the requirements are refined, it is not unusual to run out of time and money. Sometimes this results in the MVP being the final product. While this is not ideal, it is a real consideration when defining the MVP. How long will the MVP exist in production with real users?

1.4.11 Technical Debt

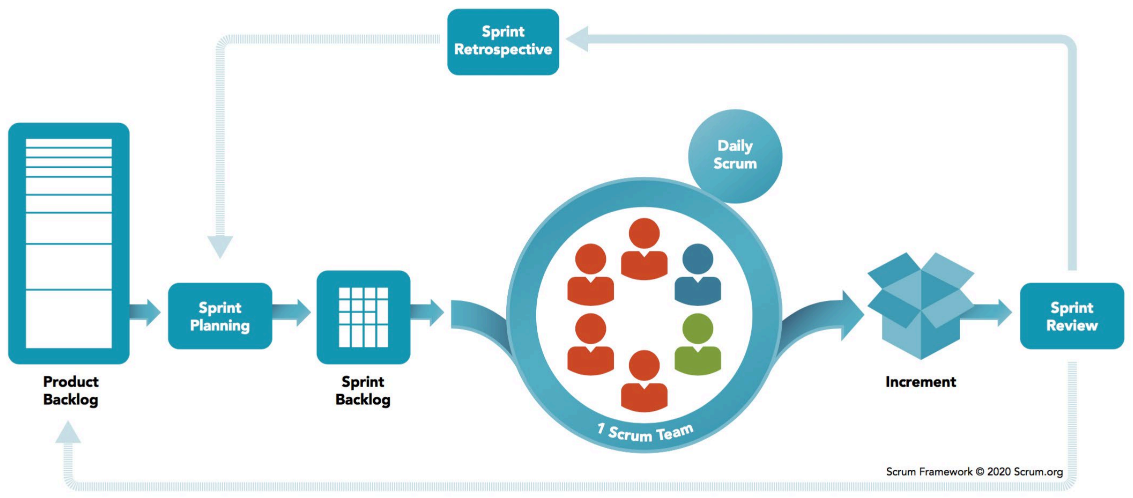
Writing code with the plan to re-write it later is not going to result in high quality code. Leaving all the defect fixes until the final iterations is not going to improve efficiency. While acknowledging that technical debt occurs (it does in all projects), it should not be a crutch for poor software development practices. Technical debt is not just accumulated by the developers, the testers may also contribute to it by shortcutting good test automation implementation practices such as re-usable frameworks and maintainable scripts. Because project schedules tend to extend and budget tends to run low, planning a major effort for refactoring and clearing the technical debt may not be realistic. It’s important to plan a part of each iteration to deal with technical debt rather than allowing it to accumulate to the end of the project. If time isn’t found at the end of the project to clear the debt, this may result in project failure due to low quality and intrinsic architectural issues. Like personal debt, technical debt becomes more costly the longer it lingers.

1.5 Agile Methodology Types and Usages

New forms of Agile methodologies surface periodically, but this syllabus will concentrate on the most commonly used forms. In addition to the actual Agile methodologies, it is important to look at the environments and other methodologies that are in use that may influence the way in which Agile is implemented.

1.5.1 Scrum

Scrum is the most commonly used form of Agile. It is sometime characterized as the guideline for implementing Agile. The following diagram shows how Scrum works:



According to Scrum.org,

> "Scrum is a lightweight framework that helps people, teams and organizations generate value through adaptive solutions for complex problems.
>
> In a nutshell, Scrum requires a Scrum Master to foster an environment where:
>
> 1. A Product Owner orders the work for a complex problem into a Product Backlog.
> 2. The Scrum Team turns a selection of the work into an Increment of value during a Sprint.
> 3. The Scrum Team and its stakeholders inspect the results and adjust for the next Sprint.
> 4. Repeat" (Sutherland, 2020)

Scrum is characterized by being controlled and "managed" by the Scrum Master and the Product Owner. The requirements are captured as User Stories and stored in the Product Backlog, which is then prioritized and divided into the iterations for implementation. Iterations are called "Sprints" and the "Definition of Done" is used to define the quality level that must be achieved prior to the release of any software components. Scrum is further explained in the section Implementing Agile.

1.5.2 Scaled Agile Framework (SAFe®)*** Registered trademark of Scaled Agile, Inc.

The Scaled Agile Framework was initially released in 2011 and is now on version 5 which was released in January of 2020. SAFe® expands the Agile concept to include all areas of the organization that are involved in the delivery of projects, including areas such as sales, finance, etc. The idea is that the entire organization must be working in an agile way in order to achieve the necessary level of agility in delivery.

In addition to emphasizing agility, SAFe® also emphasizes a "Lean" approach, meaning that an organization concentrates on the essentials without wasting time and money on activities that are not essential. The concept of the "Lean Enterprise" is introduced with SAFe®.

The following are the seven core competencies of the Lean Enterprise (Leffingwell, 2021):

1. Lean-Agile Leadership
2. Team and Technical Agility
    a. Built-in Quality
    b. Behavior-Driven Development (BDD)
    c. Agile testing
    d. Test-Driven Development (TDD)
3. Agile Product Delivery
    a. Design thinking
    b. Customer-centricity
    c. DevOps
    d. Continuous Delivery Pipeline
    e. Release on Demand
4. Enterprise Solution Delivery
5. Lean Portfolio Management
6. Organizational Agility
7. Continuous Learning Culture

SAFe® is not as widely adopted as Scrum, but aspects promoted in SAFe such as Behavior Driven Development (BDD) can be seen in many organizations.

1.5.3 Kanban

The Kanban methodology is a way of managing the flow of work. This method originated in Japan when Toyota began using it in the 1940's to modify the production process to ensure a smooth flow and up to date information regarding inventory and tasks.

Kanban has been adopted as a part of Agile projects or even independent of the Agile methodology. The goal is to capture the stages or steps in a project or process and form them into a workflow. The stages vary based on the project team, the actual stages required, and the amount of detail the team wants to track.

In software projects these stages may be:

- Analysis
- Design
- Development
- Testing
- Production

Once the stages are defined, they are converted to columns on the Kanban board. Work items (such as Stories) are then recorded on "cards" which can be physical or digital depending on the preferred tools. Cards are moved from stage to stage as they achieve completion, i.e. the Definition of Done, at each stage.

There are two primary purposes of Kanban (Monday, 2023):

- Visualize and manage the workflow of a project
- Identify and fix any bottlenecks or issues

Kanban does not prescribe a particular lifecycle but it does work well with Agile and the Kanban board sometimes replaces the Agile task board. For testers, one big advantage to a Kanban board is that it shows what is coming up for testing and allows the entire team to see where bottlenecks are occurring. Ideally, the workflow should be modulated so there are no bottlenecks created. This ensures that each stage of the lifecycle has adequate resources to allow efficient processing at the same rate as the other stages. This helps to highlight issues where the test team does not have sufficient people or test systems to allow them to test a number of “cards” at the same time. This can be solved by putting less work into the workflow, resourcing appropriately at each stage to support the smooth flow at the desired level of work, or by shifting people between the stages to add support where needed.

1.5.4 DevOps and CI/CD

Neither of these is actually a methodology, but these approaches are often tied into Agile projects to facilitate rapid delivery.

DevOps is a change in cultural philosophy removing the silos between the development and operations teams and merging them together. The result of this should be a product that is developed for success in production. The cross-education of the merger allows developers to better understand the operational requirements of the software (e.g., deployment, roll back, operational environments) and the operations people to provide input into the development process to ensure production success.

DevOps teams with a strong focus on security are sometimes called DevSecOps. Effective DevSecOps implementations should result in software being released faster and more smoothly, resulting in higher reliability, better collaboration and built in security.

The best practices for DevOps include (AWS, 2023):

- Continuous Integration - This software development practice requires that developers regularly merge their code into the central repository where it is built and tested with automated testing. The software is “continuously integrated” into the whole, and the automated tests cover both the unit testing of the modules as well as the simple integration of modules together.
- Continuous Delivery - This is the logical next step after the code is checked in, automatically built and tested. Deployment is usually done first into one or more test environments where additional tests (preferably automated) are conducted, prior to deployment to production. This continuous flow requires continuous testing so that each step in the process has a quality gate. The code cannot progress to the next step until it has met the quality criteria of the previous step.
- Microservices - Rather than monolithic web services that perform many functions, microservices focus on a single purpose. Using microservices provides more flexibility and speed in development, testing, and deployment, and improves maintainability.
- Infrastructure as Code (IaC) - Instead of requiring an environment to be procured, set up and configured by a person, IaC automates all of those processes. This allows environments to be easily reproduced, configuration settings to be clearly identified, and fast set up of new environments. IaC is primarily used in cloud environments using virtual machines but can be implemented with on-premise servers as well.
- Monitoring and Logging - The operations influence shows here in building adequate monitoring and informative logging in the event of an issue occurring. Monitoring and logging are often implemented with various detail settings so that more details can be gathered when problems are occurring versus just general information when all is running well.
- Communication and Collaboration - Sharing information between teams, particularly between operations and development, is a key component of the DevOps model.

1.5.5 Incremental vs. Iterative

While these terms are often used interchangeably in industry, they are not the same.

Incremental software development often starts from a prototype, which is designed to demonstrate functionality to a user. There may be nothing actually behind the first demonstration, but this becomes the base point for evaluation. Further refinement of the prototype is conducted based on user feedback. The software grows by increments from the base prototype. Rapid Application Development (RAD) is another form of incremental software development where the core prototype is built first, and then additions are made to it.

Iterative software development is based on the concept of deliverables resulting from each iteration. Unlike the incremental model where there is base software delivered that is then augmented, iterative software projects deliver pieces of functionality that may be unrelated to other software that has been delivered in different iterations.

From a testing standpoint, building test automation and manual testing are easier in an incremental environment because the base software is in the first delivery and will always be there. Testing in an iterative environment requires testing the software for each iteration, but there may be no way to execute a particular delivery without creating sets of drivers and stubs or mock services.

1.5.6 Failed Agile

This section would not be complete without a discussion of some of the derogatory Agile nicknames that have resulted from real experience with Agile projects. These include:

- TrAgile - the result of the project was a tragedy resulting in late delivery, blown budgets, burned out team members and unhappy customers. A project is truly Tragile when the result is the termination of the team.
- FrAgile - when technical debt is allowed to accumulate and is not addressed prior to release, the resultant software can be subject to breakage, long fix times, and general dissatisfaction.
- WAgile - when a project is touted as Agile, but the surrounding processes are clearly Waterfall, the result is a WAgile project where the Agile principles of flexibility, welcoming change, and close interaction with the user are broken.
- GolapAgile - sometimes a team will become so mired down in process and meetings that their release speed becomes akin to the walking speed of a Galapagos tortoise. These projects usually result in complete abandonment of Agile and a return to traditional methodologies such as Waterfall.

Conversely, there is also PrAgile:

- PrAgile - when some of the Agile principles are used, but these are mixed with other effective methodologies in such a way that quality, speed and satisfaction are still met, the result is a practical, partially Agile, project.

All humor aside, there are a number of ways in which Agile projects can fail. For more information regarding successful implementation of Agile, see the next section “Implementing Agile”.

All Terms for this MicroCredential

Acceptance Criteria: The exit criteria that a component or system must satisfy in order to be accepted by a user, customer, or other authorized entity.

Acceptance testing: Formal testing with respect to user needs: requirements: and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user: customers or other authorized entity to determine whether or not to accept the system.

Agile Manifesto: A statement on the values that underpin Agile software development. The values are: individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, responding to change over following a plan.

Agile Principles: The set of 12 guidelines for the implementation of the Agile Manifesto.

Agile software development: A group of software development methodologies based on iterative incremental development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

Alpha testing: Simulated or actual operational testing by potential users/customers or an independent test team at the developers' site, but outside the development organization. Alpha testing is often employed for commercial off-the-shelf software as a form of internal acceptance testing.

Behavior-driven development (BDD): A collaborative approach to development in which the team is focusing on delivering expected behavior of a component or system for the customer, which forms the basis for testing.

Beta testing: Operational testing by potential and/or existing users/customers at an external site not otherwise involved with the developers, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes. Beta testing is often employed as a form of external acceptance testing for commercial off-the-shelf software in order to acquire feedback from the market.

Definition of done (DoD): The collection of exit criteria which is used to determine if a backlog item is complete.

DevOps: A set of practices that combines software development practices (Dev) and operations practices (Ops) to provide continuous delivery with high quality.

DevSecOps: A set of DevOps practices with a particular focus on the implementation and testing of security related aspects.

Embedded iterative: A software development process that allows the development and testing tasks to follow an iterative pattern while the overarching project is implemented with a sequential model such as V-Model or Waterfall.

Epic: A large form of a user story which forms the basis of a set of smaller, related user stories. An epic may define a set of identifiable user functionality or an internal capability of the software needed to support user functionality.

Incremental development model: A development lifecycle where a project is broken into a series of increments, each of which delivers a portion of the functionality in the overall project requirements. The requirements are prioritized and delivered in priority order in the appropriate increment. In some (but not all) versions of this lifecycle model, each subproject follows a mini V-model with its own design, coding and testing phases.

Integration testing: Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.

Iterative development model: A type of software development lifecycle model in which the component or system is developed through a series of repeated cycles.

Kanban: A process used to visualize work and the workflow and to minimize work in progress.

Minimum viable product (MVP): A version of a product under development which contains just enough features to be usable by early users. These users are expected to then provide feedback which will guide additional product development.

Planning poker: A consensus-based estimation technique, mostly used to estimate effort or relative size of user stories in Agile software development. It is a variation of the Wideband Delphi method using a deck of cards with values representing the units in which the team estimates.

Product backlog: The list of prioritized functionalities which will form that basis of the product being developed. This is sometimes referred to as the to-do list and may include tasks such as refactoring and defect fixing as well as new development.

Production implementation verification (PIV): A test that is used to verify that the software is working correctly in the production environment.

Product Owner (PO): The person who represents the future product users and interacts with the development team by defining requirements (user stories), prioritizing stories/tasks, and answer questions throughout the development process.

Retrospective: A regular event in which team members discuss results, review their practices, and identify ways to improve.

Scrum: An Agile development framework commonly used in industry.

Software development lifecycle (SDLC): The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software lifecycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. Note these phases may overlap or be performed iteratively.

Sprint: An iteration of development in the Scrum framework, normally defined as 2-4 weeks long, which results in a small piece of usable functionality.

Stand-up: A daily meeting in the Scrum framework used for each team member to recall what was completed yesterday, what is planned for today, and to highlight any blocking issues.

Story grooming: The process by which a user story is further defined and prioritized by the team.

Systems integration testing (SIT): Testing the integration of systems and packages; testing interfaces to external organizations (e.g., Electronic Data Interchange, Internet).

System testing: Testing an integrated system to verify that it meets specified requirements.

Test-driven development (TDD): A way of developing software where the test cases are developed, and often automated, before the software is developed to run those test cases.

Unit testing: Testing usually performed by a developer to ensure that the unit or component of code is working as intended. This testing is often automated via the implementation of a unit testing framework.

User acceptance testing (UAT): Acceptance testing carried out by future users in a (simulated) operational environment focusing on user requirements and needs.

User story: A high-level user or business requirement commonly used in Agile software development, typically consisting of one sentence in the everyday or business language capturing what functionality a user needs and the reason behind this, any non-functional criteria, and also includes acceptance criteria.

V-Model: A framework to describe the software development lifecycle activities from requirements specification to maintenance. The V-model illustrates how testing activities can be integrated into each phase of the software development lifecycle.

Waterfall: A traditional software development lifecycle which prescribes the sequential activities used to design, develop, test, and release a software product.

Whole team: The concept that the entire team in an Agile lifecycle model is responsible for developing a successful product.

All References for this MicroCredential:

Works Cited

AWS. (2023). *What is DevOps?* Retrieved from DevOps: https://aws.amazon.com/devops/what-is-devops

Beck, e. a. (2001). *Principles behind the Agile Manifesto*. Retrieved from Agile Manifesto: https://agilemanifesto.org/principles.html

Beck, o. (2001). *Manifesto for Agile Software Development*. Retrieved from Manifesto for Agile Software Development: https://agilemanifesto.org/

Leffingwell, D. (2021). *Welcome to Scaled Agile Framework*. Retrieved from Scaled Agile Framework: www.scaledagileframework.com

Monday. (2023). *The ultimate guide to Kanban and how to use it in 2023*. Retrieved from Mondayblog: https://monday.com/blog/project-management/kanban

scrum.org. (2020). *scrum.org*. Retrieved from Scrum.org: www.scrum.org

Sutherland. (2020). *The 2020 Scrum Guide*. Retrieved from Scrum Guides: https://scrumguides.org/scrum-guide.html

Sutherland, S. (2020). *scrumguides.org*. Retrieved from Scrum Guide: https://scrumguides.org

Registered Trademarks:

SAFe® is a registered trademark of Scaled Agile, Inc.

AT*SQA

MICRO-CREDENTIAL

Agile
Software Testing
Methodologies

www.atsqa.org