Certified Tester

Foundation Level Extension Syllabus
Agile Tester

Version 2014

International Software Testing Qualifications Board

Foundation Level Extension Agile Tester Working Group: Rex Black (Chair), Bertrand Cornanguer (Vice Chair), Gerry Coleman (Learning Objectives Lead), Debra Friedenberg (Exam Lead), Alon Linetzki (Business Outcomes and Marketing Lead), Tauhida Parveen (Editor), and Leo van der Aalst (Development Lead).

Authors: Rex Black, Anders Claesson, Gerry Coleman, Bertrand Cornanguer, Istvan Forgacs, Alon Linetzki, Tilo Linz, Leo van der Aalst, Marie Walsh, and Stephan Weber.

Internal Reviewers: Mette Bruhn-Pedersen, Christopher Clements, Alessandro Collino, Debra Friedenberg, Kari Kakkonen, Beata Karpinska, Sammy Kolluru, Jennifer Leger, Thomas Mueller, Tuula Pääkkönen, Meile Posthuma, Gabor Puhalla, Lloyd Roden, Marko Rytkönen, Monika Stoecklein-Olsen, Robert Treffny, Chris Van Bael, and Erik van Veenendaal; 2013-2014.

Revision History

Version	Date	Remarks
Syllabus v0.1	26JUL2013	Standalone sections
Syllabus v0.2	16SEP2013	WG review comments on v01 incorporated
Syllabus v0.3	20OCT2013	WG review comments on v02 incorporated
Syllabus v0.7	16DEC2013	Alpha review comments on v03 incorporated
Syllabus v0.71	20DEC2013	Working group updates on v07
Syllabus v0.9	30JAN2014	Beta version
Syllabus 2014	31MAY2014	GA version

Table of Contents

Acknowledgements

This document was produced by a team from the International Software Testing Qualifications Board Foundation Level Working Group.

The Agile Extension team thanks the review team and the National Boards for their suggestions and input.

At the time the Foundation Level Agile Extension Syllabus was completed, the Agile Extension Working Group had the following membership: Rex Black (Chair), Bertrand Cornanguer (Vice Chair), Gerry Coleman (Learning Objectives Lead), Debra Friedenberg (Exam Lead), Alon Linetzki (Business Outcomes and Marketing Lead), Tauhida Parveen (Editor), and Leo van der Aalst (Development Lead).

Authors: Rex Black, Anders Claesson, Gerry Coleman, Bertrand Cornanguer, Istvan Forgacs, Alon Linetzki, Tilo Linz, Leo van der Aalst, Marie Walsh, and Stephan Weber.

Internal Reviewers: Mette Bruhn-Pedersen, Christopher Clements, Alessandro Collino, Debra Friedenberg, Kari Kakkonen, Beata Karpinska, Sammy Kolluru, Jennifer Leger, Thomas Mueller, Tuula Pääkkönen, Meile Posthuma, Gabor Puhalla, Lloyd Roden, Marko Rytkönen, Monika Stoecklein-Olsen, Robert Treffny, Chris Van Bael, and Erik van Veenendaal.

The team thanks also the following persons, from the National Boards and the Agile expert community, who participated in reviewing, commenting, and balloting of the Foundation Agile Extension Syllabus: Dani Almog, Richard Berns, Stephen Bird, Monika Bögge, Afeng Chai, Josephine Crawford, Tibor Csöndes, Huba Demeter, Arnaud Foucal, Cyril Fumery, Kobi Halperin, Inga Hansen, Hanne Hinz, Jidong Hu, Phill Isles, Shirley Itah, Martin Klonk, Kjell Lauren, Igal Levi, Rik Marselis, Johan Meivert, Armin Metzger, Peter Morgan, Ninna Morin, Ingvar Nordstrom, Chris O'Dea, Klaus Olsen, Ismo Paukamainen, Nathalie Phung, Helmut Pichler, Salvatore Reale, Stuart Reid, Hans Rombouts, Petri Säilynoja, Soile Sainio, Lars-Erik Sandberg, Dakar Shalom, Jian Shen, Marco Sogliani, Lucjan Stapp, Yaron Tsubery, Sabine Uhde, Stephanie Ulrich, Tommi Välimäki, Jurian Van de Laar, Marnix Van den Ent, António Vieira Melo, Wenye Xu, Ester Zabar, Wenqiang Zheng, Peter Zimmerer, Stevan Zivanovic, and Terry Zuo.

This document was formally approved for release by the General Assembly of the ISTQB® on May 31, 2014.

0. Introduction to this Syllabus

0.1 Purpose of this Document

This syllabus forms the basis for the International Software Testing Qualification at the Foundation Level for the Agile Tester. The ISTQB® provides this syllabus as follows:

- To National Boards, to translate into their local language and to accredit training providers. National Boards may adapt the syllabus to their particular language needs and modify the references to adapt to their local publications.
- To Exam Boards, to derive examination questions in their local language adapted to the learning objectives for each syllabus.
- To training providers, to produce courseware and determine appropriate teaching methods.
- To certification candidates, to prepare for the exam (as part of a training course or independently).
- 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.

The ISTQB® may allow other entities to use this syllabus for other purposes, provided they seek and obtain prior written permission.

0.2 Overview

The Foundation Level Agile Tester Overview document [ISTQB_FA_OVIEW] includes the following information:

- Business Outcomes for the syllabus
- Summary for the syllabus
- Relationships among the syllabi
- Description of cognitive levels (K-levels)
- Appendices

0.3 Examinable Learning Objectives

The Learning Objectives support the Business Outcomes and are used to create the examination for achieving the Certified Tester Foundation Level—Agile Tester Certification. 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. The specific learning objectives at K1, K2, and K3 levels are shown at the beginning of the pertinent chapter.

1. Agile Software Development - 150 mins.

Keywords

Agile Manifesto, Agile software development, incremental development model, iterative development model, software lifecycle, test automation, test basis, test-driven development, test oracle, user story

Learning Objectives for Agile Software Development

1.1 The Fundamentals of Agile Software Development

FA-1.1.1 (K1) Recall the basic concept of Agile software development based on the Agile Manifesto
FA-1.1.2 (K2) Understand the advantages of the whole-team approach
FA-1.1.3 (K2) Understand the benefits of early and frequent feedback

1.2 Aspects of Agile Approaches

FA-1.2.1 (K1) Recall Agile software development approaches
FA-1.2.2 (K3) Write testable user stories in collaboration with developers and business representatives
FA-1.2.3 (K2) Understand how retrospectives can be used as a mechanism for process improvement in Agile projects
FA-1.2.4 (K2) Understand the use and purpose of continuous integration
FA-1.2.5 (K1) Know the differences between iteration and release planning, and how a tester adds value in each of these activities

1.1 The Fundamentals of Agile Software Development

A tester on an Agile project will work differently than one working on a traditional project. Testers must understand the values and principles that underpin Agile projects, and how testers are an integral part of a whole-team approach together with developers and business representatives. The members in an Agile project communicate with each other early and frequently, which helps with removing defects early and developing a quality product.

1.1.1 Agile Software Development and the Agile Manifesto

In 2001, a group of individuals, representing the most widely used lightweight software development methodologies, agreed on a common set of values and principles which became known as the Manifesto for Agile Software Development or the Agile Manifesto [Agilemanifesto]. The Agile Manifesto contains four statements of values:

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

The Agile Manifesto argues that although the concepts on the right have value, those on the left have greater value.

Individuals and Interactions

Agile development is very people-centered. Teams of people build software, and it is through continuous communication and interaction, rather than a reliance on tools or processes, that teams can work most effectively.

Working Software

From a customer perspective, working software is much more useful and valuable than overly detailed documentation and it provides an opportunity to give the development team rapid feedback. In addition, because working software, albeit with reduced functionality, is available much earlier in the development lifecycle, Agile development can confer significant time-to-market advantage. Agile development is, therefore, especially useful in rapidly changing business environments where the problems and/or solutions are unclear or where the business wishes to innovate in new problem domains.

Customer Collaboration

Customers often find great difficulty in specifying the system that they require. Collaborating directly with the customer improves the likelihood of understanding exactly what the customer requires. While having contracts with customers may be important, working in regular and close collaboration with them is likely to bring more success to the project.

Responding to Change

Change is inevitable in software projects. The environment in which the business operates, legislation, competitor activity, technology advances, and other factors can have major influences on the project and its objectives. These factors must be accommodated by the development process. As such, having flexibility in work practices to embrace change is more important than simply adhering rigidly to a plan.

Principles

The core Agile Manifesto values are captured in twelve principles:

- 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, at intervals of between a few weeks to a few 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.

The different Agile methodologies provide prescriptive practices to put these values and principles into action.

1.1.2 Whole-Team Approach

The whole-team approach means involving everyone with the knowledge and skills necessary to ensure project success. The team includes representatives from the customer and other business stakeholders who determine product features. The team should be relatively small; successful teams have been observed with as few as three people and as many as nine. Ideally, the whole team shares the same workspace, as co-location strongly facilitates communication and interaction. The whole-team approach is supported through the daily stand-up meetings (see Section 2.2.1) involving all members of the team, where work progress is communicated and any impediments to progress are highlighted. The whole-team approach promotes more effective and efficient team dynamics.

The use of a whole-team approach to product development is one of the main benefits of Agile development. Its benefits include:

- Enhancing communication and collaboration within the team
- Enabling the various skill sets within the team to be leveraged to the benefit of the project
- Making quality everyone’s responsibility

The whole team is responsible for quality in Agile projects. The essence of the whole-team approach lies in the testers, developers, and the business representatives working together in every step of the development process. Testers will work closely with both developers and business representatives to ensure that the desired quality levels are achieved. This includes supporting and collaborating with business representatives to help them create suitable acceptance tests, working with developers to agree on the testing strategy, and deciding on test automation approaches. Testers can thus transfer and extend testing knowledge to other team members and influence the development of the product.

The whole team is involved in any consultations or meetings in which product features are presented, analyzed, or estimated. The concept of involving testers, developers, and business representatives in all feature discussions is known as the power of three [Crispin08].

1.1.3 Early and Frequent Feedback

Agile projects have short iterations enabling the project team to receive early and continuous feedback on product quality throughout the development lifecycle. One way to provide rapid feedback is by continuous integration (see Section 1.2.4).

When sequential development approaches are used, the customer often does not see the product until the project is nearly completed. At that point, it is often too late for the development team to effectively address any issues the customer may have. By getting frequent customer feedback as the project progresses, Agile teams can incorporate most new changes into the product development process. Early and frequent feedback helps the team focus on the features with the highest business value, or associated risk, and these are delivered to the customer first. It also helps manage the team better since the capability of the team is transparent to everyone. For example, how much work can we do in a sprint or iteration? What could help us go faster? What is preventing us from doing so?

The benefits of early and frequent feedback include:

- Avoiding requirements misunderstandings, which may not have been detected until later in the development cycle when they are more expensive to fix.
- Clarifying customer feature requests, making them available for customer use early. This way, the product better reflects what the customer wants.
- Discovering (via continuous integration), isolating, and resolving quality problems early.
- Providing information to the Agile team regarding its productivity and ability to deliver.
- Promoting consistent project momentum.

1.2 Aspects of Agile Approaches

There are a number of Agile approaches in use by organizations. Common practices across most Agile organizations include collaborative user story creation, retrospectives, continuous integration, and planning for each iteration as well as for overall release. This subsection describes some of the Agile approaches.

1.2.1 Agile Software Development Approaches

There are several Agile approaches, each of which implements the values and principles of the Agile Manifesto in different ways. In this syllabus, three representatives of Agile approaches are considered: Extreme Programming (XP), Scrum, and Kanban.

Extreme Programming
Extreme Programming (XP), originally introduced by Kent Beck [Beck04], is an Agile approach to software development described by certain values, principles, and development practices.

XP embraces five values to guide development: communication, simplicity, feedback, courage, and respect.

XP describes a set of principles as additional guidelines: humanity, economics, mutual benefit, self-similarity, improvement, diversity, reflection, flow, opportunity, redundancy, failure, quality, baby steps, and accepted responsibility.

XP describes thirteen primary practices: sit together, whole team, informative workspace, energized work, pair programming, stories, weekly cycle, quarterly cycle, slack, ten-minute build, continuous integration, test first programming, and incremental design.

Many of the Agile software development approaches in use today are influenced by XP and its values and principles. For example, Agile teams following Scrum often incorporate XP practices.

Scrum

Scrum is an Agile management framework which contains the following constituent instruments and practices [Schwaber01]:

- Sprint: Scrum divides a project into iterations (called sprints) of fixed length (usually two to four weeks).
- Product Increment: Each sprint results in a potentially releasable/shippable product (called an increment).
- Product Backlog: The product owner manages a prioritized list of planned product items (called the product backlog). The product backlog evolves from sprint to sprint (called backlog refinement).
- Sprint Backlog: At the start of each sprint, the Scrum team selects a set of highest priority items (called the sprint backlog) from the product backlog. Since the Scrum team, not the product owner, selects the items to be realized within the sprint, the selection is referred to as being on the pull principle rather than the push principle.
- Definition of Done: To make sure that there is a potentially releasable product at each sprint's end, the Scrum team discusses and defines appropriate criteria for sprint completion. The discussion deepens the team's understanding of the backlog items and the product requirements.
- Timeboxing: Only those tasks, requirements, or features that the team expects to finish within the sprint are part of the sprint backlog. If the development team cannot finish a task within a sprint, the associated product features are removed from the sprint and the task is moved back into the product backlog. Timeboxing applies not only to tasks, but in other situations (e.g., enforcing meeting start and end times).
- Transparency: The development team reports and updates sprint status on a daily basis at a meeting called the daily scrum. This makes the content and progress of the current sprint, including test results, visible to the team, management, and all interested parties. For example, the development team can show sprint status on a whiteboard.

Scrum defines three roles:

- Scrum Master: ensures that Scrum practices and rules are implemented and followed, and resolves any violations, resource issues, or other impediments that could prevent the team from following the practices and rules. This person is not the team lead, but a coach.
- Product Owner: represents the customer, and generates, maintains, and prioritizes the product backlog. This person is not the team lead.
- Development Team: develop and test the product. The team is self-organized: There is no team lead, so the team makes the decisions. The team is also cross-functional (see Section 2.3.2 and Section 3.1.4).

Scrum (as opposed to XP) does not dictate specific software development techniques (e.g., test first programming). In addition, Scrum does not provide guidance on how testing has to be done in a Scrum project.

Kanban

Kanban [Anderson13] is a management approach that is sometimes used in Agile projects. The general objective is to visualize and optimize the flow of work within a value-added chain. Kanban utilizes three instruments [Linz14]:

- Kanban Board: The value chain to be managed is visualized by a Kanban board. Each column shows a station, which is a set of related activities, e.g., development or testing. The items to be produced or tasks to be processed are symbolized by tickets moving from left to right across the board through the stations.
- Work-in-Progress Limit: The amount of parallel active tasks is strictly limited. This is controlled by the maximum number of tickets allowed for a station and/or globally for the board. Whenever a station has free capacity, the worker pulls a ticket from the predecessor station.
- Lead Time: Kanban is used to optimize the continuous flow of tasks by minimizing the (average) lead time for the complete value stream.

Kanban features some similarities to Scrum. In both frameworks, visualizing the active tasks (e.g., on a public whiteboard) provides transparency of content and progress of tasks. Tasks not yet scheduled are waiting in a backlog and moved onto the Kanban board as soon as there is new space (production capacity) available.

Iterations or sprints are optional in Kanban. The Kanban process allows releasing its deliverables item by item, rather than as part of a release. Timeboxing as a synchronizing mechanism, therefore, is optional, unlike in Scrum, which synchronizes all tasks within a sprint.

1.2.2 Collaborative User Story Creation

Poor specifications are often a major reason for project failure. Specification problems can result from the users' lack of insight into their true needs, absence of a global vision for the system, redundant or contradictory features, and other miscommunications. In Agile development, user stories are written to capture requirements from the perspectives of developers, testers, and business representatives. In sequential development, this shared vision of a feature is accomplished through formal reviews after requirements are written; in Agile development, this shared vision is accomplished through frequent informal reviews while the requirements are being written.

The user stories must address both functional and non-functional characteristics. Each story includes acceptance criteria for these characteristics. These criteria should be defined in collaboration between business representatives, developers, and testers. They provide developers and testers with an extended vision of the feature that business representatives will validate. An Agile team considers a task finished when a set of acceptance criteria have been satisfied.

Typically, the tester's unique perspective will improve the user story by identifying missing details or non-functional requirements. A tester can contribute by asking business representatives open-ended questions about the user story, proposing ways to test the user story, and confirming the acceptance criteria.

The collaborative authorship of the user story can use techniques such as brainstorming and mind mapping. The tester may use the INVEST technique [INVEST]:

- Independent
- Negotiable
- Valuable
- Estimable
- Small
- Testable

According to the 3C concept [Jeffries00], a user story is the conjunction of three elements:

- Card: The card is the physical media describing a user story. It identifies the requirement, its criticality, expected development and test duration, and the acceptance criteria for that story. The description has to be accurate, as it will be used in the product backlog.

- Conversation: The conversation explains how the software will be used. The conversation can be documented or verbal. Testers, having a different point of view than developers and business representatives [ISTQB_FL_SYL], bring valuable input to the exchange of thoughts, opinions, and experiences. Conversation begins during the release-planning phase and continues when the story is scheduled.
- Confirmation: The acceptance criteria, discussed in the conversation, are used to confirm that the story is done. These acceptance criteria may span multiple user stories. Both positive and negative tests should be used to cover the criteria. During confirmation, various participants play the role of a tester. These can include developers as well as specialists focused on performance, security, interoperability, and other quality characteristics. To confirm a story as done, the defined acceptance criteria should be tested and shown to be satisfied.

Agile teams vary in terms of how they document user stories. Regardless of the approach taken to document user stories, documentation should be concise, sufficient, and necessary.

1.2.3 Retrospectives

In Agile development, a retrospective is a meeting held at the end of each iteration to discuss what was successful, what could be improved, and how to incorporate the improvements and retain the successes in future iterations. Retrospectives cover topics such as the process, people, organizations, relationships, and tools. Regularly conducted retrospective meetings, when appropriate follow up activities occur, are critical to self-organization and continual improvement of development and testing.

Retrospectives can result in test-related improvement decisions focused on test effectiveness, test productivity, test case quality, and team satisfaction. They may also address the testability of the applications, user stories, features, or system interfaces. Root cause analysis of defects can drive testing and development improvements. In general, teams should implement only a few improvements per iteration. This allows for continuous improvement at a sustained pace.

The timing and organization of the retrospective depends on the particular Agile method followed. Business representatives and the team attend each retrospective as participants while the facilitator organizes and runs the meeting. In some cases, the teams may invite other participants to the meeting.

Testers should play an important role in the retrospectives. Testers are part of the team and bring their unique perspective [ISTQB_FL_SYL], Section 1.5. Testing occurs in each sprint and vitally contributes to success. All team members, testers and non-testers, can provide input on both testing and non-testing activities.

Retrospectives must occur within a professional environment characterized by mutual trust. The attributes of a successful retrospective are the same as those for any other review as is discussed in the Foundation Level syllabus [ISTQB_FL_SYL], Section 3.2.

1.2.4 Continuous Integration

Delivery of a product increment requires reliable, working, integrated software at the end of every sprint. Continuous integration addresses this challenge by merging all changes made to the software and integrating all changed components regularly, at least once a day. Configuration management, compilation, software build, deployment, and testing are wrapped into a single, automated, repeatable process. Since developers integrate their work constantly, build constantly, and test constantly, defects in code are detected more quickly.

Following the developers' coding, debugging, and check-in of code into a shared source code repository, a continuous integration process consists of the following automated activities:

- Static code analysis: executing static code analysis and reporting results
- Compile: compiling and linking the code, generating the executable files
- Unit test: executing the unit tests, checking code coverage and reporting test results
- Deploy: installing the build into a test environment
- Integration test: executing the integration tests and reporting results
- Report (dashboard): posting the status of all these activities to a publicly visible location or e-mailing status to the team

An automated build and test process takes place on a daily basis and detects integration errors early and quickly. Continuous integration allows Agile testers to run automated tests regularly, in some cases as part of the continuous integration process itself, and send quick feedback to the team on the quality of the code. These test results are visible to all team members, especially when automated reports are integrated into the process. Automated regression testing can be continuous throughout the iteration. Good automated regression tests cover as much functionality as possible, including user stories delivered in the previous iterations. Good coverage in the automated regression tests helps support building (and testing) large integrated systems. When the regression testing is automated, the Agile testers are freed to concentrate their manual testing on new features, implemented changes, and confirmation testing of defect fixes.

In addition to automated tests, organizations using continuous integration typically use build tools to implement continuous quality control. In addition to running unit and integration tests, such tools can run additional static and dynamic tests, measure and profile performance, extract and format documentation from the source code, and facilitate manual quality assurance processes. This continuous application of quality control aims to improve the quality of the product as well as reduce the time taken to deliver it by replacing the traditional practice of applying quality control after completing all development.

Build tools can be linked to automatic deployment tools, which can fetch the appropriate build from the continuous integration or build server and deploy it into one or more development, test, staging, or even production environments. This reduces the errors and delays associated with relying on specialized staff or programmers to install releases in these environments.

Continuous integration can provide the following benefits:

- Allows earlier detection and easier root cause analysis of integration problems and conflicting changes
- Gives the development team regular feedback on whether the code is working
- Keeps the version of the software being tested within a day of the version being developed
- Reduces regression risk associated with developer code refactoring due to rapid re-testing of the code base after each small set of changes
- Provides confidence that each day's development work is based on a solid foundation
- Makes progress toward the completion of the product increment visible, encouraging developers and testers
- Eliminates the schedule risks associated with big-bang integration
- Provides constant availability of executable software throughout the sprint for testing, demonstration, or education purposes
- Reduces repetitive manual testing activities
- Provides quick feedback on decisions made to improve quality and tests

However, continuous integration is not without its risks and challenges:

- Continuous integration tools have to be introduced and maintained
- The continuous integration process must be defined and established
- Test automation requires additional resources and can be complex to establish
- Thorough test coverage is essential to achieve automated testing advantages

- Teams sometimes over-rely on unit tests and perform too little system and acceptance testing

Continuous integration requires the use of tools, including tools for testing, tools for automating the build process, and tools for version control.

1.2.5 Release and Iteration Planning

As mentioned in the Foundation Level syllabus [ISTQB_FL_SYL], planning is an on-going activity, and this is the case in Agile lifecycles as well. For Agile lifecycles, two kinds of planning occur, release planning and iteration planning.

Release planning looks ahead to the release of a product, often a few months ahead of the start of a project. Release planning defines and re-defines the product backlog, and may involve refining larger user stories into a collection of smaller stories. Release planning provides the basis for a test approach and test plan spanning all iterations. Release plans are high-level.

In release planning, business representatives establish and prioritize the user stories for the release, in collaboration with the team (see Section 1.2.2). Based on these user stories, project and quality risks are identified and a high-level effort estimation is performed (see Section 3.2).

Testers are involved in release planning and especially add value in the following activities:

- Defining testable user stories, including acceptance criteria
- Participating in project and quality risk analyses
- Estimating testing effort associated with the user stories
- Defining the necessary test levels
- Planning the testing for the release

After release planning is done, iteration planning for the first iteration starts. Iteration planning looks ahead to the end of a single iteration and is concerned with the iteration backlog.

In iteration planning, the team selects user stories from the prioritized release backlog, elaborates the user stories, performs a risk analysis for the user stories, and estimates the work needed for each user story. If a user story is too vague and attempts to clarify it have failed, the team can refuse to accept it and use the next user story based on priority. The business representatives must answer the team's questions about each story so the team can understand what they should implement and how to test each story.

The number of stories selected is based on established team velocity and the estimated size of the selected user stories. After the contents of the iteration are finalized, the user stories are broken into tasks, which will be carried out by the appropriate team members.

Testers are involved in iteration planning and especially add value in the following activities:

- Participating in the detailed risk analysis of user stories
- Determining the testability of the user stories
- Creating acceptance tests for the user stories
- Breaking down user stories into tasks (particularly testing tasks)
- Estimating testing effort for all testing tasks
- Identifying functional and non-functional aspects of the system to be tested
- Supporting and participating in test automation at multiple levels of testing

Release plans may change as the project proceeds, including changes to individual user stories in the product backlog. These changes may be triggered by internal or external factors. Internal factors include delivery capabilities, velocity, and technical issues. External factors include the discovery of

new markets and opportunities, new competitors, or business threats that may change release objectives and/or target dates. In addition, iteration plans may change during an iteration. For example, a particular user story that was considered relatively simple during estimation might prove more complex than expected.

These changes can be challenging for testers. Testers must understand the big picture of the release for test planning purposes, and they must have an adequate test basis and test oracle in each iteration for test development purposes as discussed in the Foundation Level syllabus [ISTQB_FL_SYL], Section 1.4. The required information must be available to the tester early, and yet change must be embraced according to Agile principles. This dilemma requires careful decisions about test strategies and test documentation. For more on Agile testing challenges, see [Black09], Chapter 12.

Release and iteration planning should address test planning as well as planning for development activities. Particular test-related issues to address include:

- The scope of testing, the extent of testing for those areas in scope, the test goals, and the reasons for these decisions.
- The team members who will carry out the test activities.
- The test environment and test data needed, when they are needed, and whether any additions or changes to the test environment and/or data will occur prior to or during the project.
- The timing, sequencing, dependencies, and prerequisites for the functional and non-functional test activities (e.g., how frequently to run regression tests, which features depend on other features or test data, etc.), including how the test activities relate to and depend on development activities.
- The project and quality risks to be addressed (see Section 3.2.1).

In addition, the larger team estimation effort should include consideration of the time and effort needed to complete the required testing activities.

2. Fundamental Agile Testing Principles, Practices, and Processes – 105 mins.

Keywords
build verification test, configuration item, configuration management

Learning Objectives for Fundamental Agile Testing Principles, Practices, and Processes

2.1 The Differences between Testing in Traditional and Agile Approaches

FA-2.1.1 (K2) Describe the differences between testing activities in Agile projects and non-Agile projects
FA-2.1.2 (K2) Describe how development and testing activities are integrated in Agile projects
FA-2.1.3 (K2) Describe the role of independent testing in Agile projects

2.2 Status of Testing in Agile Projects

FA-2.2.1 (K2) Describe the tools and techniques used to communicate the status of testing in an Agile project, including test progress and product quality
FA-2.2.2 (K2) Describe the process of evolving tests across multiple iterations and explain why test automation is important to manage regression risk in Agile projects

2.3 Role and Skills of a Tester in an Agile Team

FA-2.3.1 (K2) Understand the skills (people, domain, and testing) of a tester in an Agile team
FA-2.3.2 (K2) Understand the role of a tester within an Agile team

2.1 The Differences between Testing in Traditional and Agile Approaches

As described in the Foundation Level syllabus [ISTQB_FL_SYL] and in [Black09], test activities are related to development activities, and thus testing varies in different lifecycles. Testers must understand the differences between testing in traditional lifecycle models (e.g., sequential such as the V-model or iterative such as RUP) and Agile lifecycles in order to work effectively and efficiently. The Agile models differ in terms of the way testing and development activities are integrated, the project work products, the names, entry and exit criteria used for various levels of testing, the use of tools, and how independent testing can be effectively utilized.

Testers should remember that organizations vary considerably in their implementation of lifecycles. Deviation from the ideals of Agile lifecycles (see Section 1.1) may represent intelligent customization and adaptation of the practices. The ability to adapt to the context of a given project, including the software development practices actually followed, is a key success factor for testers.

2.1.1 Testing and Development Activities

One of the main differences between traditional lifecycles and Agile lifecycles is the idea of very short iterations, each iteration resulting in working software that delivers features of value to business stakeholders. At the beginning of the project, there is a release planning period. This is followed by a sequence of iterations. At the beginning of each iteration, there is an iteration planning period. Once iteration scope is established, the selected user stories are developed, integrated with the system, and tested. These iterations are highly dynamic, with development, integration, and testing activities taking place throughout each iteration, and with considerable parallelism and overlap. Testing activities occur throughout the iteration, not as a final activity.

Testers, developers, and business stakeholders all have a role in testing, as with traditional lifecycles. Developers perform unit tests as they develop features from the user stories. Testers then test those features. Business stakeholders also test the stories during implementation. Business stakeholders might use written test cases, but they also might simply experiment with and use the feature in order to provide fast feedback to the development team.

In some cases, hardening or stabilization iterations occur periodically to resolve any lingering defects and other forms of technical debt. However, the best practice is that no feature is considered done until it has been integrated and tested with the system [Goucher09]. Another good practice is to address defects remaining from the previous iteration at the beginning of the next iteration, as part of the backlog for that iteration (referred to as “fix bugs first”). However, some complain that this practice results in a situation where the total work to be done in the iteration is unknown and it will be more difficult to estimate when the remaining features can be done. At the end of the sequence of iterations, there can be a set of release activities to get the software ready for delivery, though in some cases delivery occurs at the end of each iteration.

When risk-based testing is used as one of the test strategies, a high-level risk analysis occurs during release planning, with testers often driving that analysis. However, the specific quality risks associated with each iteration are identified and assessed in iteration planning. This risk analysis can influence the sequence of development as well as the priority and depth of testing for the features. It also influences the estimation of the test effort required for each feature (see Section 3.2).

In some Agile practices (e.g., Extreme Programming), pairing is used. Pairing can involve testers working together in twos to test a feature. Pairing can also involve a tester working collaboratively with a developer to develop and test a feature. Pairing can be difficult when the test team is distributed, but processes and tools can help enable distributed pairing. For more information on distributed work, see [ISTQB_ALTM_SYL], Section 2.8.

Testers may also serve as testing and quality coaches within the team, sharing testing knowledge and supporting quality assurance work within the team. This promotes a sense of collective ownership of quality of the product.

Test automation at all levels of testing occurs in many Agile teams, and this can mean that testers spend time creating, executing, monitoring, and maintaining automated tests and results. Because of the heavy use of test automation, a higher percentage of the manual testing on Agile projects tends to be done using experience-based and defect-based techniques such as software attacks, exploratory testing, and error guessing (see [ISTQB_ALTA_SYL], Sections 3.3 and 3.4 and [ISTQB_FL_SYL], Section 4.5). While developers will focus on creating unit tests, testers should focus on creating automated integration, system, and system integration tests. This leads to a tendency for Agile teams to favor testers with a strong technical and test automation background.

One core Agile principle is that change may occur throughout the project. Therefore, lightweight work product documentation is favored in Agile projects. Changes to existing features have testing implications, especially regression testing implications. The use of automated testing is one way of managing the amount of test effort associated with change. However, it's important that the rate of change not exceed the project team's ability to deal with the risks associated with those changes.

2.1.2 Project Work Products

Project work products of immediate interest to Agile testers typically fall into three categories:

1. Business-oriented work products that describe what is needed (e.g., requirements specifications) and how to use it (e.g., user documentation)
2. Development work products that describe how the system is built (e.g., database entity-relationship diagrams), that actually implement the system (e.g., code), or that evaluate individual pieces of code (e.g., automated unit tests)
3. Test work products that describe how the system is tested (e.g., test strategies and plans), that actually test the system (e.g., manual and automated tests), or that present test results (e.g., test dashboards as discussed in Section 2.2.1)

In a typical Agile project, it is a common practice to avoid producing vast amounts of documentation. Instead, focus is more on having working software, together with automated tests that demonstrate conformance to requirements. This encouragement to reduce documentation applies only to documentation that does not deliver value to the customer. In a successful Agile project, a balance is struck between increasing efficiency by reducing documentation and providing sufficient documentation to support business, testing, development, and maintenance activities. The team must make a decision during release planning about which work products are required and what level of work product documentation is needed.

Typical business-oriented work products on Agile projects include user stories and acceptance criteria. User stories are the Agile form of requirements specifications, and should explain how the system should behave with respect to a single, coherent feature or function. A user story should define a feature small enough to be completed in a single iteration. Larger collections of related features, or a collection of sub-features that make up a single complex feature, may be referred to as "epics". Epics may include user stories for different development teams. For example, one user story can describe what is required at the API-level (middleware) while another story describes what is needed at the UI-level (application). These collections may be developed over a series of sprints. Each epic and its user stories should have associated acceptance criteria.

Typical developer work products on Agile projects include code. Agile developers also often create automated unit tests. These tests might be created after the development of code. In some cases, though, developers create tests incrementally, before each portion of the code is written, in order to provide a way of verifying, once that portion of code is written, whether it works as expected. While

this approach is referred to as test first or test-driven development, in reality the tests are more a form of executable low-level design specifications rather than tests [Beck02].

Typical tester work products on Agile projects include automated tests, as well as documents such as test plans, quality risk catalogs, manual tests, defect reports, and test results logs. The documents are captured in as lightweight a fashion as possible, which is often also true of these documents in traditional lifecycles. Testers will also produce test metrics from defect reports and test results logs, and again there is an emphasis on a lightweight approach.

In some Agile implementations, especially regulated, safety critical, distributed, or highly complex projects and products, further formalization of these work products is required. For example, some teams transform user stories and acceptance criteria into more formal requirements specifications. Vertical and horizontal traceability reports may be prepared to satisfy auditors, regulations, and other requirements.

2.1.3 Test Levels

Test levels are test activities that are logically related, often by the maturity or completeness of the item under test.

In sequential lifecycle models, the test levels are often defined such that the exit criteria of one level are part of the entry criteria for the next level. In some iterative models, this rule does not apply. Test levels overlap. Requirement specification, design specification, and development activities may overlap with test levels.

In some Agile lifecycles, overlap occurs because changes to requirements, design, and code can happen at any point in an iteration. While Scrum, in theory, does not allow changes to the user stories after iteration planning, in practice such changes sometimes occur. During an iteration, any given user story will typically progress sequentially through the following test activities:

- Unit testing, typically done by the developer
- Feature acceptance testing, which is sometimes broken into two activities:
  - Feature verification testing, which is often automated, may be done by developers or testers, and involves testing against the user story's acceptance criteria
  - Feature validation testing, which is usually manual and can involve developers, testers, and business stakeholders working collaboratively to determine whether the feature is fit for use, to improve visibility of the progress made, and to receive real feedback from the business stakeholders

In addition, there is often a parallel process of regression testing occurring throughout the iteration. This involves re-running the automated unit tests and feature verification tests from the current iteration and previous iterations, usually via a continuous integration framework.

In some Agile projects, there may be a system test level, which starts once the first user story is ready for such testing. This can involve executing functional tests, as well as non-functional tests for performance, reliability, usability, and other relevant test types.

Agile teams can employ various forms of acceptance testing (using the term as explained in the Foundation Level syllabus [ISTQB_FL_SYL]). Internal alpha tests and external beta tests may occur, either at the close of each iteration, after the completion of each iteration, or after a series of iterations. User acceptance tests, operational acceptance tests, regulatory acceptance tests, and contract acceptance tests also may occur, either at the close of each iteration, after the completion of each iteration, or after a series of iterations.

2.1.4 Testing and Configuration Management

Agile projects often involve heavy use of automated tools to develop, test, and manage software development. Developers use tools for static analysis, unit testing, and code coverage. Developers continuously check the code and unit tests into a configuration management system, using automated build and test frameworks. These frameworks allow the continuous integration of new software with the system, with the static analysis and unit tests run repeatedly as new software is checked in [Kubaczkowski].

These automated tests can also include functional tests at the integration and system levels. Such functional automated tests may be created using functional testing harnesses, open-source user interface functional test tools, or commercial tools, and can be integrated with the automated tests run as part of the continuous integration framework. In some cases, due to the duration of the functional tests, the functional tests are separated from the unit tests and run less frequently. For example, unit tests may be run each time new software is checked in, while the longer functional tests are run only every few days.

One goal of the automated tests is to confirm that the build is functioning and installable. If any automated test fails, the team should fix the underlying defect in time for the next code check-in. This requires an investment in real-time test reporting to provide good visibility into test results. This approach helps reduce expensive and inefficient cycles of "build-install-fail-rebuild-reinstall" that can occur in many traditional projects, since changes that break the build or cause software to fail to install are detected quickly.

Automated testing and build tools help to manage the regression risk associated with the frequent change that often occurs in Agile projects. However, over-reliance on automated unit testing alone to manage these risks can be a problem, as unit testing often has limited defect detection effectiveness [Jones11]. Automated tests at the integration and system levels are also required.

2.1.5 Organizational Options for Independent Testing

As discussed in the Foundation Level syllabus [ISTQB_FL_SYL], independent testers are often more effective at finding defects. In some Agile teams, developers create many of the tests in the form of automated tests. One or more testers may be embedded within the team, performing many of the testing tasks. However, given those testers' position within the team, there is a risk of loss of independence and objective evaluation.

Other Agile teams retain fully independent, separate test teams, and assign testers on-demand during the final days of each sprint. This can preserve independence, and these testers can provide an objective, unbiased evaluation of the software. However, time pressures, lack of understanding of the new features in the product, and relationship issues with business stakeholders and developers often lead to problems with this approach.

A third option is to have an independent, separate test team where testers are assigned to Agile teams on a long-term basis, at the beginning of the project, allowing them to maintain their independence while gaining a good understanding of the product and strong relationships with other team members. In addition, the independent test team can have specialized testers outside of the Agile teams to work on long-term and/or iteration-independent activities, such as developing automated test tools, carrying out non-functional testing, creating and supporting test environments and data, and carrying out test levels that might not fit well within a sprint (e.g., system integration testing).

2.2 Status of Testing in Agile Projects

Change takes place rapidly in Agile projects. This change means that test status, test progress, and product quality constantly evolve, and testers must devise ways to get that information to the team so that they can make decisions to stay on track for successful completion of each iteration. In addition, change can affect existing features from previous iterations. Therefore, manual and automated tests must be updated to deal effectively with regression risk.

2.2.1 Communicating Test Status, Progress, and Product Quality

Agile teams progress by having working software at the end of each iteration. To determine when the team will have working software, they need to monitor the progress of all work items in the iteration and release. Testers in Agile teams utilize various methods to record test progress and status, including test automation results, progression of test tasks and stories on the Agile task board, and burndown charts showing the team's progress. These can then be communicated to the rest of the team using media such as wiki dashboards and dashboard-style emails, as well as verbally during stand-up meetings. Agile teams may use tools that automatically generate status reports based on test results and task progress, which in turn update wiki-style dashboards and emails. This method of communication also gathers metrics from the testing process, which can be used in process improvement. Communicating test status in such an automated manner also frees testers' time to focus on designing and executing more test cases.

Teams may use burndown charts to track progress across the entire release and within each iteration. A burndown chart [Crispin08] represents the amount of work left to be done against time allocated to the release or iteration.

To provide an instant, detailed visual representation of the whole team's current status, including the status of testing, teams may use Agile task boards. The story cards, development tasks, test tasks, and other tasks created during iteration planning (see Section 1.2.5) are captured on the task board, often using color-coordinated cards to determine the task type. During the iteration, progress is managed via the movement of these tasks across the task board into columns such as *to do*, *work in progress*, *verify*, and *done*. Agile teams may use tools to maintain their story cards and Agile task boards, which can automate dashboards and status updates.

Testing tasks on the task board relate to the acceptance criteria defined for the user stories. As test automation scripts, manual tests, and exploratory tests for a test task achieve a passing status, the task moves into the *done* column of the task board. The whole team reviews the status of the task board regularly, often during the daily stand-up meetings, to ensure tasks are moving across the board at an acceptable rate. If any tasks (including testing tasks) are not moving or are moving too slowly, the team reviews and addresses any issues that may be blocking the progress of those tasks.

The daily stand-up meeting includes all members of the Agile team including testers. At this meeting, they communicate their current status. The agenda for each member is [Agile Alliance Guide]:

- What have you completed since the last meeting?
- What do you plan to complete by the next meeting?
- What is getting in your way?

Any issues that may block test progress are communicated during the daily stand-up meetings, so the whole team is aware of the issues and can resolve them accordingly.

To improve the overall product quality, many Agile teams perform customer satisfaction surveys to receive feedback on whether the product meets customer expectations. Teams may use other metrics similar to those captured in traditional development methodologies, such as test pass/fail rates, defect discovery rates, confirmation and regression test results, defect density, defects found and fixed, requirements coverage, risk coverage, code coverage, and code churn to improve the product quality.

As with any lifecycle, the metrics captured and reported should be relevant and aid decision-making. Metrics should not be used to reward, punish, or isolate any team members.

2.2.2 Managing Regression Risk with Evolving Manual and Automated Test Cases

In an Agile project, as each iteration completes, the product grows. Therefore, the scope of testing also increases. Along with testing the code changes made in the current iteration, testers also need to verify no regression has been introduced on features that were developed and tested in previous iterations. The risk of introducing regression in Agile development is high due to extensive code churn (lines of code added, modified, or deleted from one version to another). Since responding to change is a key Agile principle, changes can also be made to previously delivered features to meet business needs. In order to maintain velocity without incurring a large amount of technical debt, it is critical that teams invest in test automation at all test levels as early as possible. It is also critical that all test assets such as automated tests, manual test cases, test data, and other testing artifacts are kept up-to-date with each iteration. It is highly recommended that all test assets be maintained in a configuration management tool in order to enable version control, to ensure ease of access by all team members, and to support making changes as required due to changing functionality while still preserving the historic information of the test assets.

Because complete repetition of all tests is seldom possible, especially in tight-timeline Agile projects, testers need to allocate time in each iteration to review manual and automated test cases from previous and current iterations to select test cases that may be candidates for the regression test suite, and to retire test cases that are no longer relevant. Tests written in earlier iterations to verify specific features may have little value in later iterations due to feature changes or new features which alter the way those earlier features behave.

While reviewing test cases, testers should consider suitability for automation. The team needs to automate as many tests as possible from previous and current iterations. This allows automated regression tests to reduce regression risk with less effort than manual regression testing would require. This reduced regression test effort frees the testers to more thoroughly test new features and functions in the current iteration.

It is critical that testers have the ability to quickly identify and update test cases from previous iterations and/or releases that are affected by the changes made in the current iteration. Defining how the team designs, writes, and stores test cases should occur during release planning. Good practices for test design and implementation need to be adopted early and applied consistently. The shorter timeframes for testing and the constant change in each iteration will increase the impact of poor test design and implementation practices.

Use of test automation, at all test levels, allows Agile teams to provide rapid feedback on product quality. Well-written automated tests provide a living document of system functionality [Crispin08]. By checking the automated tests and their corresponding test results into the configuration management system, aligned with the versioning of the product builds, Agile teams can review the functionality tested and the test results for any given build at any given point in time.

Automated unit tests are run before source code is checked into the mainline of the configuration management system to ensure the code changes do not break the software build. To reduce build breaks, which can slow down the progress of the whole team, code should not be checked in unless all automated unit tests pass. Automated unit test results provide immediate feedback on code and build quality, but not on product quality.

Automated acceptance tests are run regularly as part of the continuous integration full system build. These tests are run against a complete system build at least daily, but are generally not run with each code check-in as they take longer to run than automated unit tests and could slow down code check-

ins. The test results from automated acceptance tests provide feedback on product quality with respect to regression since the last build, but they do not provide status of overall product quality.

Automated tests can be run continuously against the system. An initial subset of automated tests to cover critical system functionality and integration points should be created immediately after a new build is deployed into the test environment. These tests are commonly known as build verification tests. Results from the build verification tests will provide instant feedback on the software after deployment, so teams don't waste time testing an unstable build.

Automated tests contained in the regression test set are generally run as part of the daily main build in the continuous integration environment, and again when a new build is deployed into the test environment. As soon as an automated regression test fails, the team stops and investigates the reasons for the failing test. The test may have failed due to legitimate functional changes in the current iteration, in which case the test and/or user story may need to be updated to reflect the new acceptance criteria. Alternatively, the test may need to be retired if another test has been built to cover the changes. However, if the test failed due to a defect, it is a good practice for the team to fix the defect prior to progressing with new features.

In addition to test automation, the following testing tasks may also be automated:

- Test data generation
- Loading test data into systems
- Deployment of builds into the test environments
- Restoration of a test environment (e.g., the database or website data files) to a baseline
- Comparison of data outputs

Automation of these tasks reduces the overhead and allows the team to spend time developing and testing new features.

2.3 Role and Skills of a Tester in an Agile Team

In an Agile team, testers must closely collaborate with all other team members and with business stakeholders. This has a number of implications in terms of the skills a tester must have and the activities they perform within an Agile team.

2.3.1 Agile Tester Skills

Agile testers should have all the skills mentioned in the Foundation Level syllabus [ISTQB_FL_SYL]. In addition to these skills, a tester in an Agile team should be competent in test automation, test-driven development, acceptance test-driven development, white-box, black-box, and experience-based testing.

As Agile methodologies depend heavily on collaboration, communication, and interaction between the team members as well as stakeholders outside the team, testers in an Agile team should have good interpersonal skills. Testers in Agile teams should:

- Be positive and solution-oriented with team members and stakeholders
- Display critical, quality-oriented, skeptical thinking about the product
- Actively acquire information from stakeholders (rather than relying entirely on written specifications)
- Accurately evaluate and report test results, test progress, and product quality
- Work effectively to define testable user stories, especially acceptance criteria, with customer representatives and stakeholders
- Collaborate within the team, working in pairs with programmers and other team members
- Respond to change quickly, including changing, adding, or improving test cases
- Plan and organize their own work

Continuous skills growth, including interpersonal skills growth, is essential for all testers, including those on Agile teams.

2.3.2 The Role of a Tester in an Agile Team

The role of a tester in an Agile team includes activities that generate and provide feedback not only on test status, test progress, and product quality, but also on process quality. In addition to the activities described elsewhere in this syllabus, these activities include:

- Understanding, implementing, and updating the test strategy
- Measuring and reporting test coverage across all applicable coverage dimensions
- Ensuring proper use of testing tools
- Configuring, using, and managing test environments and test data
- Reporting defects and working with the team to resolve them
- Coaching other team members in relevant aspects of testing
- Ensuring the appropriate testing tasks are scheduled during release and iteration planning
- Actively collaborating with developers and business stakeholders to clarify requirements, especially in terms of testability, consistency, and completeness
- Participating proactively in team retrospectives, suggesting and implementing improvements

Within an Agile team, each team member is responsible for product quality and plays a role in performing test-related tasks.

Agile organizations may encounter some test-related organizational risks:

- Testers work so closely to developers that they lose the appropriate tester mindset
- Testers become tolerant of or silent about inefficient, ineffective, or low-quality practices within the team
- Testers cannot keep pace with the incoming changes in time-constrained iterations

To mitigate these risks, organizations may consider variations for preserving independence discussed in Section 2.1.5.

3. Agile Testing Methods, Techniques, and Tools – 480 mins.

Keywords
acceptance criteria, exploratory testing, performance testing, product risk, quality risk, regression testing, test approach, test charter, test estimation, test execution automation, test strategy, test-driven development, unit test framework

Learning Objectives for Agile Testing Methods, Techniques, and Tools

3.1 Agile Testing Methods

FA-3.1.1 (K1) Recall the concepts of test-driven development, acceptance test-driven development, and behavior-driven development
FA-3.1.2 (K1) Recall the concepts of the test pyramid
FA-3.1.3 (K2) Summarize the testing quadrants and their relationships with testing levels and testing types
FA-3.1.4 (K3) For a given Agile project, practice the role of a tester in a Scrum team

3.2 Assessing Quality Risks and Estimating Test Effort

FA-3.2.1 (K3) Assess quality risks within an Agile project
FA-3.2.2 (K3) Estimate testing effort based on iteration content and quality risks

3.3 Techniques in Agile Projects

FA-3.3.1 (K3) Interpret relevant information to support testing activities
FA-3.3.2 (K2) Explain to business stakeholders how to define testable acceptance criteria
FA-3.3.3 (K3) Given a user story, write acceptance test-driven development test cases
FA-3.3.4 (K3) For both functional and non-functional behavior, write test cases using black box test design techniques based on given user stories
FA-3.3.5 (K3) Perform exploratory testing to support the testing of an Agile project

3.4 Tools in Agile Projects

FA-3.4.1 (K1) Recall different tools available to testers according to their purpose and to activities in Agile projects

3.1 Agile Testing Methods

There are certain testing practices that can be followed in every development project (agile or not) to produce quality products. These include writing tests in advance to express proper behavior, focusing on early defect prevention, detection, and removal, and ensuring that the right test types are run at the right time and as part of the right test level. Agile practitioners aim to introduce these practices early. Testers in Agile projects play a key role in guiding the use of these testing practices throughout the lifecycle.

3.1.1 Test-Driven Development, Acceptance Test-Driven Development, and Behavior-Driven Development

Test-driven development, acceptance test-driven development, and behavior-driven development are three complementary techniques in use among Agile teams to carry out testing across the various test levels. Each technique is an example of a fundamental principle of testing, the benefit of early testing and QA activities, since the tests are defined before the code is written.

Test-Driven Development
Test-driven development (TDD) is used to develop code guided by automated test cases. The process for test-driven development is:

- Add a test that captures the programmer's concept of the desired functioning of a small piece of code
- Run the test, which should fail since the code doesn't exist
- Write the code and run the test in a tight loop until the test passes
- Refactor the code after the test is passed, re-running the test to ensure it continues to pass against the refactored code
- Repeat this process for the next small piece of code, running the previous tests as well as the added tests

The tests written are primarily unit level and are code-focused, though tests may also be written at the integration or system levels. Test-driven development gained its popularity through Extreme Programming [Beck02], but is also used in other Agile methodologies and sometimes in sequential lifecycles. It helps developers focus on clearly-defined expected results. The tests are automated and are used in continuous integration.

Acceptance Test-Driven Development
Acceptance test-driven development [Adzic09] defines acceptance criteria and tests during the creation of user stories (see Section 1.2.2). Acceptance test-driven development is a collaborative approach that allows every stakeholder to understand how the software component has to behave and what the developers, testers, and business representatives need to ensure this behavior. The process of acceptance test-driven development is explained in Section 3.3.2.

Acceptance test-driven development creates reusable tests for regression testing. Specific tools support creation and execution of such tests, often within the continuous integration process. These tools can connect to data and service layers of the application, which allows tests to be executed at the system or acceptance level. Acceptance test-driven development allows quick resolution of defects and validation of feature behavior. It helps determine if the acceptance criteria are met for the feature.

Behavior-Driven Development
Behavior-driven development [Chelimsky10] allows a developer to focus on testing the code based on the expected behavior of the software. Because the tests are based on the exhibited behavior of the software, the tests are generally easier for other team members and stakeholders to understand.

Specific behavior-driven development frameworks can be used to define acceptance criteria based on the given/when/then format:

Given some initial context,
When an event occurs,
Then ensure some outcomes.

From these requirements, the behavior-driven development framework generates code that can be used by developers to create test cases. Behavior-driven development helps the developer collaborate with other stakeholders, including testers, to define accurate unit tests focused on business needs.

3.1.2 The Test Pyramid

A software system may be tested at different levels. Typical test levels are, from the base of the pyramid to the top, unit, integration, system, and acceptance (see [ISTQB_FL_SYL], Section 2.2). The test pyramid emphasizes having a large number of tests at the lower levels (bottom of the pyramid) and, as development moves to the upper levels, the number of tests decreases (top of the pyramid). Usually unit and integration level tests are automated and are created using API-based tools. At the system and acceptance levels, the automated tests are created using GUI-based tools. The test pyramid concept is based on the testing principle of early QA and testing (i.e., eliminating defects as early as possible in the lifecycle).

3.1.3 Testing Quadrants, Test Levels, and Testing Types

Testing quadrants, defined by Brian Marick [Crispin08], align the test levels with the appropriate test types in the Agile methodology. The testing quadrants model, and its variants, helps to ensure that all important test types and test levels are included in the development lifecycle. This model also provides a way to differentiate and describe the types of tests to all stakeholders, including developers, testers, and business representatives.

In the testing quadrants, tests can be business (user) or technology (developer) facing. Some tests support the work done by the Agile team and confirm software behavior. Other tests can verify the product. Tests can be fully manual, fully automated, a combination of manual and automated, or manual but supported by tools. The four quadrants are as follows:

- Quadrant Q1 is unit level, technology facing, and supports the developers. This quadrant contains unit tests. These tests should be automated and included in the continuous integration process.
- Quadrant Q2 is system level, business facing, and confirms product behavior. This quadrant contains functional tests, examples, story tests, user experience prototypes, and simulations. These tests check the acceptance criteria and can be manual or automated. They are often created during the user story development and thus improve the quality of the stories. They are useful when creating automated regression test suites.
- Quadrant Q3 is system or user acceptance level, business facing, and contains tests that critique the product, using realistic scenarios and data. This quadrant contains exploratory testing, scenarios, process flows, usability testing, user acceptance testing, alpha testing, and beta testing. These tests are often manual and are user-oriented.
- Quadrant Q4 is system or operational acceptance level, technology facing, and contains tests that critique the product. This quadrant contains performance, load, stress, and scalability tests, security tests, maintainability, memory management, compatibility and interoperability, data migration, infrastructure, and recovery testing. These tests are often automated.

During any given iteration, tests from any or all quadrants may be required. The testing quadrants apply to dynamic testing rather than static testing.

3.1.4 The Role of a Tester

Throughout this syllabus, general reference has been made to Agile methods and techniques, and the role of a tester within various Agile lifecycles. This subsection looks specifically at the role of a tester in a project following a Scrum lifecycle [Aalst13].

Teamwork
Teamwork is a fundamental principle in Agile development. Agile emphasizes the whole-team approach consisting of developers, testers, and business representatives working together. The following are organizational and behavioral best practices in Scrum teams:

- Cross-functional: Each team member brings a different set of skills to the team. The team works together on test strategy, test planning, test specification, test execution, test evaluation, and test results reporting.
- Self-organizing: The team may consist only of developers, but, as noted in Section 2.1.5, ideally there would be one or more testers.
- Co-located: Testers sit together with the developers and the product owner.
- Collaborative: Testers collaborate with their team members, other teams, the stakeholders, the product owner, and the Scrum Master.
- Empowered: Technical decisions regarding design and testing are made by the team as a whole (developers, testers, and Scrum Master), in collaboration with the product owner and other teams if needed.
- Committed: The tester is committed to question and evaluate the product's behavior and characteristics with respect to the expectations and needs of the customers and users.
- Transparent: Development and testing progress is visible on the Agile task board (see Section 2.2.1).
- Credible: The tester must ensure the credibility of the strategy for testing, its implementation, and execution, otherwise the stakeholders will not trust the test results. This is often done by providing information to the stakeholders about the testing process.
- Open to feedback: Feedback is an important aspect of being successful in any project, especially in Agile projects. Retrospectives allow teams to learn from successes and from failures.
- Resilient: Testing must be able to respond to change, like all other activities in Agile projects.

These best practices maximize the likelihood of successful testing in Scrum projects.

Sprint Zero
Sprint zero is the first iteration of the project where many preparation activities take place (see Section 1.2.5). The tester collaborates with the team on the following activities during this iteration:

- Identify the scope of the project (i.e., the product backlog)
- Create an initial system architecture and high-level prototypes
- Plan, acquire, and install needed tools (e.g., for test management, defect management, test automation, and continuous integration)
- Create an initial test strategy for all test levels, addressing (among other topics) test scope, technical risks, test types (see Section 3.1.3), and coverage goals
- Perform an initial quality risk analysis (see Section 3.2.1)
- Define test metrics to measure the test process, the progress of testing in the project, and product quality
- Specify the definition of "done"
- Create the task board (see Section 2.2.1)
- Define when to continue or stop testing before delivering the system to the customer

Sprint zero sets the direction for what testing needs to achieve and how testing needs to achieve it throughout the sprints.

Integration
In Agile projects, the objective is to deliver customer value on a continuous basis (preferably in every sprint). To enable this, the integration strategy should consider both design and testing. To enable a continuous testing strategy for the delivered functionality and characteristics, it is important to identify all dependencies between underlying functions and features.

Test Planning
Since testing is fully integrated into the Agile team, test planning should start during the release planning session and be updated during each sprint. Test planning for the release and each sprint should address the issues discussed in Section 1.2.5.

Sprint planning results in a set of tasks to put on the task board, where each task should have a length of one or two days of work. In addition, any testing issues should be tracked to keep a steady flow of testing.

Agile Testing Practices
Many practices may be useful for testers in a scrum team, some of which include:

- Pairing: Two team members (e.g., a tester and a developer, two testers, or a tester and a product owner) sit together at one workstation to perform a testing or other sprint task.
- Incremental test design: Test cases and charters are gradually built from user stories and other test bases, starting with simple tests and moving toward more complex ones.
- Ming mapping: Mind mapping is a useful tool when testing [Crispin08]. For example, testers can use mind mapping to identify which test sessions to perform, to show test strategies, and to describe test data.

These practices are in addition to other practices discussed in this syllabus and in Chapter 4 of the Foundation Level syllabus [ISTQB_FL_SYL].

3.2 Assessing Quality Risks and Estimating Test Effort

A typical objective of testing in all projects, Agile or traditional, is to reduce the risk of product quality problems to an acceptable level prior to release. Testers in Agile projects can use the same types of techniques used in traditional projects to identify quality risks (or product risks), assess the associated level of risk, estimate the effort required to reduce those risks sufficiently, and then mitigate those risks through test design, implementation, and execution. However, given the short iterations and rate of change in Agile projects, some adaptations of those techniques are required.

3.2.1 Assessing Quality Risks in Agile Projects

One of the many challenges in testing is the proper selection, allocation, and prioritization of test conditions. This includes determining the appropriate amount of effort to allocate in order to cover each condition with tests, and sequencing the resulting tests in a way that optimizes the effectiveness and efficiency of the testing work to be done. Risk identification, analysis, and risk mitigation strategies can be used by the testers in Agile teams to help determine an acceptable number of test cases to execute, although many interacting constraints and variables may require compromises.

Risk is the possibility of a negative or undesirable outcome or event. The level of risk is found by assessing the likelihood of occurrence of the risk and the impact of the risk. When the primary effect of the potential problem is on product quality, potential problems are referred to as quality risks or product risks. When the primary effect of the potential problem is on project success, potential problems are referred to as project risks or planning risks [Black07] [vanVeenendaal12].

In Agile projects, quality risk analysis takes place at two places.

- Release planning: business representatives who know the features in the release provide a high-level overview of the risks, and the whole team, including the tester(s), may assist in the risk identification and assessment.
- Iteration planning: the whole team identifies and assesses the quality risks.

Examples of quality risks for a system include:
- Incorrect calculations in reports (a functional risk related to accuracy)
- Slow response to user input (a non-functional risk related to efficiency and response time)
- Difficulty in understanding screens and fields (a non-functional risk related to usability and understandability)

As mentioned earlier, an iteration starts with iteration planning, which culminates in estimated tasks on a task board. These tasks can be prioritized in part based on the level of quality risk associated with them. Tasks associated with higher risks should start earlier and involve more testing effort. Tasks associated with lower risks should start later and involve less testing effort.

An example of how the quality risk analysis process in an Agile project may be carried out during iteration planning is outlined in the following steps:
1. Gather the Agile team members together, including the tester(s)
2. List all the backlog items for the current iteration (e.g., on a task board)
3. Identify the quality risks associated with each item, considering all relevant quality characteristics
4. Assess each identified risk, which includes two activities: categorizing the risk and determining its level of risk based on the impact and the likelihood of defects
5. Determine the extent of testing proportional to the level of risk.
6. Select the appropriate test technique(s) to mitigate each risk, based on the risk, the level of risk, and the relevant quality characteristic.

The tester then designs, implements, and executes tests to mitigate the risks. This includes the totality of features, behaviors, quality characteristics, and attributes that affect customer, user, and stakeholder satisfaction.

Throughout the project, the team should remain aware of additional information that may change the set of risks and/or the level of risk associated with known quality risks. Periodic adjustment of the quality risk analysis, which results in adjustments to the tests, should occur. Adjustments include identifying new risks, re-assessing the level of existing risks, and evaluating the effectiveness of risk mitigation activities.

Quality risks can also be mitigated before test execution starts. For example, if problems with the user stories are found during risk identification, the project team can thoroughly review user stories as a mitigating strategy.

3.2.2 Estimating Testing Effort Based on Content and Risk

During release planning, the Agile team estimates the effort required to complete the release. The estimate addresses the testing effort as well. A common estimation technique used in Agile projects is planning poker, a consensus-based technique. The product owner or customer reads a user story to the estimators. Each estimator has a deck of cards with values similar to the Fibonacci sequence (i.e., 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …) or any other progression of choice (e.g., shirt sizes ranging from extra-small to extra-extra-large). The values represent the number of story points, effort days, or other units in which the team estimates. The Fibonacci sequence is recommended because the numbers in the sequence reflect that uncertainty grows proportionally with the size of the story. A high estimate usually means that the story is not well understood or should be broken down into multiple smaller stories.

The estimators discuss the feature, and ask questions of the product owner as needed. Aspects such as development and testing effort, complexity of the story, and scope of testing play a role in the estimation. Therefore, it is advisable to include the risk level of a backlog item, in addition to the priority specified by the product owner, before the planning poker session is initiated. When the feature has been fully discussed, each estimator privately selects one card to represent his or her estimate. All cards are then revealed at the same time. If all estimators selected the same value, that becomes the estimate. If not, the estimators discuss the differences in estimates after which the poker round is repeated until agreement is reached, either by consensus or by applying rules (e.g., use the median, use the highest score) to limit the number of poker rounds. These discussions ensure a reliable estimate of the effort needed to complete product backlog items requested by the product owner and help improve collective knowledge of what has to be done [Cohn04].

3.3 Techniques in Agile Projects

Many of the test techniques and testing levels that apply to traditional projects can also be applied to Agile projects. However, for Agile projects, there are some specific considerations and variances in test techniques, terminologies, and documentation that should be considered.

3.3.1 Acceptance Criteria, Adequate Coverage, and Other Information for Testing

Agile projects outline initial requirements as user stories in a prioritized backlog at the start of the project. Initial requirements are short and usually follow a predefined format (see Section 1.2.2). Non-functional requirements, such as usability and performance, are also important and can be specified as unique user stories or connected to other functional user stories. Non-functional requirements may follow a predefined format or standard, such as [ISO25000], or an industry specific standard.

The user stories serve as an important test basis. Other possible test bases include:

- Experience from previous projects
- Existing functions, features, and quality characteristics of the system
- Code, architecture, and design
- User profiles (context, system configurations, and user behavior)
- Information on defects from existing and previous projects
- A categorization of defects in a defect taxonomy
- Applicable standards (e.g., [DO-178B] for avionics software)
- Quality risks (see Section 3.2.1)

During each iteration, developers create code which implements the functions and features described in the user stories, with the relevant quality characteristics, and this code is verified and validated via acceptance testing. To be testable, acceptance criteria should address the following topics where relevant [Wiegers13]:

- Functional behavior: The externally observable behavior with user actions as input operating under certain configurations.
- Quality characteristics: How the system performs the specified behavior. The characteristics may also be referred to as quality attributes or non-functional requirements. Common quality characteristics are performance, reliability, usability, etc.
- Scenarios (use cases): A sequence of actions between an external actor (often a user) and the system, in order to accomplish a specific goal or business task.
- Business rules: Activities that can only be performed in the system under certain conditions defined by outside procedures and constraints (e.g., the procedures used by an insurance company to handle insurance claims).
- External interfaces: Descriptions of the connections between the system to be developed and the outside world. External interfaces can be divided into different types (user interface, interface to other systems, etc.).

- Constraints: Any design and implementation constraint that will restrict the options for the developer. Devices with embedded software must often respect physical constraints such as size, weight, and interface connections.
- Data definitions: The customer may describe the format, data type, allowed values, and default values for a data item in the composition of a complex business data structure (e.g., the ZIP code in a U.S. mail address).

In addition to the user stories and their associated acceptance criteria, other information is relevant for the tester, including:

- How the system is supposed to work and be used
- The system interfaces that can be used/accessed to test the system
- Whether current tool support is sufficient
- Whether the tester has enough knowledge and skill to perform the necessary tests

Testers will often discover the need for additional information (e.g., code coverage) throughout the iterations and should work collaboratively with the rest of the Agile team members to obtain that information. Relevant information plays a part in determining whether a particular activity can be considered done. This concept of the definition of done is critical in Agile projects and applies in a number of different ways as discussed in the following sub-subsections.

Test Levels

Each test level has its own definition of done. The following list gives examples that may be relevant for the different test levels.

- Unit testing
  - 100% decision coverage where possible, with careful reviews of any infeasible paths
  - Static analysis performed on all code
  - No unresolved major defects (ranked based on priority and severity)
  - No known unacceptable technical debt remaining in the design and the code [Jones11]
  - All code, unit tests, and unit test results reviewed
  - All unit tests automated
  - Important characteristics are within agreed limits (e.g., performance)
- Integration testing
  - All functional requirements tested, including both positive and negative tests, with the number of tests based on size, complexity, and risks
  - All interfaces between units tested
  - All quality risks covered according to the agreed extent of testing
  - No unresolved major defects (prioritized according to risk and importance)
  - All defects found are reported
  - All regression tests automated, where possible, with all automated tests stored in a common repository
- System testing
  - End-to-end tests of user stories, features, and functions
  - All user personas covered
  - The most important quality characteristics of the system covered (e.g., performance, robustness, reliability)
  - Testing done in a production-like environment(s), including all hardware and software for all supported configurations, to the extent possible
  - All quality risks covered according to the agreed extent of testing
  - All regression tests automated, where possible, with all automated tests stored in a common repository
  - All defects found are reported and possibly fixed
  - No unresolved major defects (prioritized according to risk and importance)

User Story
The definition of done for user stories may be determined by the following criteria:

- The user stories selected for the iteration are complete, understood by the team, and have detailed, testable acceptance criteria
- All the elements of the user story are specified and reviewed, including the user story acceptance tests, have been completed
- Tasks necessary to implement and test the selected user stories have been identified and estimated by the team

Feature
The definition of done for features, which may span multiple user stories or epics, may include:

- All constituent user stories, with acceptance criteria, are defined and approved by the customer
- The design is complete, with no known technical debt
- The code is complete, with no known technical debt or unfinished refactoring
- Unit tests have been performed and have achieved the defined level of coverage
- Integration tests and system tests for the feature have been performed according to the defined coverage criteria
- No major defects remain to be corrected
- Feature documentation is complete, which may include release notes, user manuals, and on-line help functions

Iteration
The definition of done for the iteration may include the following:

- All features for the iteration are ready and individually tested according to the feature level criteria
- Any non-critical defects that cannot be fixed within the constraints of the iteration added to the product backlog and prioritized
- Integration of all features for the iteration completed and tested
- Documentation written, reviewed, and approved

At this point, the software is potentially releasable because the iteration has been successfully completed, but not all iterations result in a release.

Release
The definition of done for a release, which may span multiple iterations, may include the following areas:

- Coverage: All relevant test basis elements for all contents of the release have been covered by testing. The adequacy of the coverage is determined by what is new or changed, its complexity and size, and the associated risks of failure.
- Quality: The defect intensity (e.g., how many defects are found per day or per transaction), the defect density (e.g., the number of defects found compared to the number of user stories, effort, and/or quality attributes), estimated number of remaining defects are within acceptable limits, the consequences of unresolved and remaining defects (e.g., the severity and priority) are understood and acceptable, the residual level of risk associated with each identified quality risk is understood and acceptable.
- Time: If the pre-determined delivery date has been reached, the business considerations associated with releasing and not releasing need to be considered.
- Cost: The estimated lifecycle cost should be used to calculate the return on investment for the delivered system (i.e., the calculated development and maintenance cost should be considerably lower than the expected total sales of the product). The main part of the lifecycle

cost often comes from maintenance after the product has been released, due to the number of defects escaping to production.

3.3.2 Applying Acceptance Test-Driven Development

Acceptance test-driven development is a test-first approach. Test cases are created prior to implementing the user story. The test cases are created by the Agile team, including the developer, the tester, and the business representatives [Adzic09] and may be manual or automated. The first step is a specification workshop where the user story is analyzed, discussed, and written by developers, testers, and business representatives. Any incompleteness, ambiguities, or errors in the user story are fixed during this process.

The next step is to create the tests. This can be done by the team together or by the tester individually. In any case, an independent person such as a business representative validates the tests. The tests are examples that describe the specific characteristics of the user story. These examples will help the team implement the user story correctly. Since examples and tests are the same, these terms are often used interchangeably. The work starts with basic examples and open questions.

Typically, the first tests are the positive tests, confirming the correct behavior without exception or error conditions, comprising the sequence of activities executed if everything goes as expected. After the positive path tests are done, the team should write negative path tests and cover non-functional attributes as well (e.g., performance, usability). Tests are expressed in a way that every stakeholder is able to understand, containing sentences in natural language involving the necessary preconditions, if any, the inputs, and the related outputs.

The examples must cover all the characteristics of the user story and should not add to the story. This means that an example should not exist which describes an aspect of the user story not documented in the story itself. In addition, no two examples should describe the same characteristics of the user story.

3.3.3 Functional and Non-Functional Black Box Test Design

In Agile testing, many tests are created by testers concurrently with the developers' programming activities. Just as the developers are programming based on the user stories and acceptance criteria, so are the testers creating tests based on user stories and their acceptance criteria. (Some tests, such as exploratory tests and some other experience-based tests, are created later, during test execution, as explained in Section 3.3.4.) Testers can apply traditional black box test design techniques such as equivalence partitioning, boundary value analysis, decision tables, and state transition testing to create these tests. For example, boundary value analysis could be used to select test values when a customer is limited in the number of items they may select for purchase.

In many situations, non-functional requirements can be documented as user stories. Black box test design techniques (such as boundary value analysis) can also be used to create tests for non-functional quality characteristics. The user story might contain performance or reliability requirements. For example, a given execution cannot exceed a time limit or a number of operations may fail less than a certain number of times.

For more information about the use of black box test design techniques, see the Foundation Level syllabus [ISTQB_FL_SYL] and the Advanced Level Test Analyst syllabus [ISTQB_ALTA_SYL].

3.3.4 Exploratory Testing and Agile Testing

Exploratory testing is important in Agile projects due to the limited time available for test analysis and the limited details of the user stories. In order to achieve the best results, exploratory testing should be combined with other experience-based techniques as part of a reactive testing strategy, blended with

other testing strategies such as analytical risk-based testing, analytical requirements-based testing, model-based testing, and regression-averse testing. Test strategies and test strategy blending is discussed in the Foundation Level syllabus [ISTQB_FL_SYL].

In exploratory testing, test design and test execution occur at the same time, guided by a prepared test charter. A test charter provides the test conditions to cover during a time-boxed testing session. During exploratory testing, the results of the most recent tests guide the next test. The same white box and black box techniques can be used to design the tests as when performing pre-designed testing.

A test charter may include the following information:

- Actor: intended user of the system
- Purpose: the theme of the charter including what particular objective the actor wants to achieve, i.e., the test conditions
- Setup: what needs to be in place in order to start the test execution
- Priority: relative importance of this charter, based on the priority of the associated user story or the risk level
- Reference: specifications (e.g., user story), risks, or other information sources
- Data: whatever data is needed to carry out the charter
- Activities: a list of ideas of what the actor may want to do with the system (e.g., “Log on to the system as a super user”) and what would be interesting to test (both positive and negative tests)
- Oracle notes: how to evaluate the product to determine correct results (e.g., to capture what happens on the screen and compare to what is written in the user’s manual)
- Variations: alternative actions and evaluations to complement the ideas described under activities

To manage exploratory testing, a method called session-based test management can be used. A session is defined as an uninterrupted period of testing which could last from 60 to 120 minutes. Test sessions include the following:

- Survey session (to learn how it works)
- Analysis session (evaluation of the functionality or characteristics)

Deep coverage (corner cases, scenarios, interactions)The quality of the tests depends on the testers ability to ask relevant questions about what to test. Examples include the following:

- What is most important to find out about the system?
- In what way may the system fail?
- What happens if.....?
- What should happen when.....?
- Are customer needs, requirements, and expectations fulfilled?
- Is the system possible to install (and remove if necessary) in all supported upgrade paths?

During test execution, the tester uses creativity, intuition, cognition, and skill to find possible problems with the product. The tester also needs to have good knowledge and understanding of the software under test, the business domain, how the software is used, and how to determine when the system fails.

A set of heuristics can be applied when testing. A heuristic can guide the tester in how to perform the testing and to evaluate the results [Hendrickson]. Examples include:

- Boundaries
- CRUD (Create, Read, Update, Delete)
- Configuration variations
- Interruptions (e.g., log off, shut down, or reboot)

It is important for the tester to document the process as much as possible. Otherwise, it would be difficult to go back and see how a problem in the system was discovered. The following list provides examples of information that may be useful to document:

- Test coverage: what input data have been used, how much has been covered, and how much remains to be tested
- Evaluation notes*:* observations during testing, do the system and feature under test seem to be stable, were any defects found, what is planned as the next step according to the current observations, and any other list of ideas
- Risk/strategy list*:* which risks have been covered and which ones remain among the most important ones, will the initial strategy be followed, does it need any changes
- Issues, questions, and anomalies*:* any unexpected behavior, any questions regarding the efficiency of the approach, any concerns about the ideas/test attempts, test environment, test data, misunderstanding of the function, test script or the system under test
- Actual behavior: recording of actual behavior of the system that needs to be saved (e.g., video, screen captures, output data files)

The information logged should be captured and/or summarized into some form of status management tools (e.g., test management tools, task management tools, the task board), in a way that makes it easy for stakeholders to understand the current status for all testing that was performed.

3.4 Tools in Agile Projects

Tools described in the Foundation Level syllabus [ISTQB_FL_SYL] are relevant and used by testers on Agile teams. Not all tools are used the same way and some tools have more relevance for Agile projects than they have in traditional projects. For example, although the test management tools, requirements management tools, and incident management tools (defect tracking tools) can be used by Agile teams, some Agile teams opt for an all-inclusive tool (e.g., application lifecycle management or task management) that provides features relevant to Agile development, such as task boards, burndown charts, and user stories. Configuration management tools are important to testers in Agile teams due to the high number of automated tests at all levels and the need to store and manage the associated automated test artifacts.

In addition to the tools described in the Foundation Level syllabus [ISTQB_FL_SYL], testers on Agile projects may also utilize the tools described in the following subsections. These tools are used by the whole team to ensure team collaboration and information sharing, which are key to Agile practices.

3.4.1 Task Management and Tracking Tools

In some cases, Agile teams use physical story/task boards (e.g., whiteboard, corkboard) to manage and track user stories, tests, and other tasks throughout each sprint. Other teams will use application lifecycle management and task management software, including electronic task boards. These tools serve the following purposes:

- Record stories and their relevant development and test tasks, to ensure that nothing gets lost during a sprint
- Capture team members' estimates on their tasks and automatically calculate the effort required to implement a story, to support efficient iteration planning sessions
- Associate development tasks and test tasks with the same story, to provide a complete picture of the team's effort required to implement the story
- Aggregate developer and tester updates to the task status as they complete their work, automatically providing a current calculated snapshot of the status of each story, the iteration, and the overall release

- Provide a visual representation (via metrics, charts, and dashboards) of the current state of each user story, the iteration, and the release, allowing all stakeholders, including people on geographically distributed teams, to quickly check status
- Integrate with configuration management tools, which can allow automated recording of code check-ins and builds against tasks, and, in some cases, automated status updates for tasks

3.4.2 Communication and Information Sharing Tools

In addition to e-mail, documents, and verbal communication, Agile teams often use three additional types of tools to support communication and information sharing: wikis, instant messaging, and desktop sharing.

Wikis allow teams to build and share an online knowledge base on various aspects of the project, including the following:

- Product feature diagrams, feature discussions, prototype diagrams, photos of whiteboard discussions, and other information
- Tools and/or techniques for developing and testing found to be useful by other members of the team
- Metrics, charts, and dashboards on product status, which is especially useful when the wiki is integrated with other tools such as the build server and task management system, since the tool can update product status automatically
- Conversations between team members, similar to instant messaging and email, but in a way that is shared with everyone else on the team

Instant messaging, audio teleconferencing, and video chat tools provide the following benefits:

- Allow real time direct communication between team members, especially distributed teams
- Involve distributed teams in standup meetings
- Reduce telephone bills by use of voice-over-IP technology, removing cost constraints that could reduce team member communication in distributed settings

Desktop sharing and capturing tools provide the following benefits:

- In distributed teams, product demonstrations, code reviews, and even pairing can occur
- Capturing product demonstrations at the end of each iteration, which can be posted to the team's wiki

These tools should be used to complement and extend, not replace, face-to-face communication in Agile teams.

3.4.3 Software Build and Distribution Tools

As discussed earlier in this syllabus, daily build and deployment of software is a key practice in Agile teams. This requires the use of continuous integration tools and build distribution tools. The uses, benefits, and risks of these tools was described earlier in Section 1.2.4.

3.4.4 Configuration Management Tools

On Agile teams, configuration management tools may be used not only to store source code and automated tests, but manual tests and other test work products are often stored in the same repository as the product source code. This provides traceability between which versions of the software were tested with which particular versions of the tests, and allows for rapid change without losing historical information. The main types of version control systems include centralized source control systems and distributed version control systems. The team size, structure, location, and requirements to integrate with other tools will determine which version control system is right for a particular Agile project.

3.4.5 Test Design, Implementation, and Execution Tools

Some tools are useful to Agile testers at specific points in the software testing process. While most of these tools are not new or specific to Agile, they provide important capabilities given the rapid change of Agile projects.

- Test design tools: Use of tools such as mind maps have become more popular to quickly design and define tests for a new feature.
- Test case management tools: The type of test case management tools used in Agile may be part of the whole team's application lifecycle management or task management tool.
- Test data preparation and generation tools: Tools that generate data to populate an application's database are very beneficial when a lot of data and combinations of data are necessary to test the application. These tools can also help re-define the database structure as the product undergoes changes during an Agile project and refactor the scripts to generate the data. This allows quick updating of test data as changes occur. Some test data preparation tools use production data sources as a raw material and use scripts to remove or anonymize sensitive data. Other test data preparation tools can help with validating large data inputs or outputs.
- Test data load tools: After data has been generated for testing, it needs to be loaded into the application. Manual data entry is often time consuming and error prone, but data load tools are available to make the process reliable and efficient. In fact, many of the data generator tools include an integrated data load component. In other cases, bulk-loading using the database management systems is also possible.
- Automated test execution tools: There are test execution tools which are more aligned to Agile testing. Specific tools are available via both commercial and open source avenues to support test first approaches, such as behavior-driven development, test-driven development, and acceptance test-driven development. These tools allow testers and business staff to express the expected system behavior in tables or natural language using keywords.
- Exploratory test tools: Tools that capture and log activities performed on an application during an exploratory test session are beneficial to the tester and developer, as they record the actions taken. This is useful when a defect is found, as the actions taken before the failure occurred have been captured and can be used to report the defect to the developers. Logging steps performed in an exploratory test session may prove to be beneficial if the test is ultimately included in the automated regression test suite.

3.4.6 Cloud Computing and Virtualization Tools

Virtualization allows a single physical resource (server) to operate as many separate, smaller resources. When virtual machines or cloud instances are used, teams have a greater number of servers available to them for development and testing. This can help to avoid delays associated with waiting for physical servers. Provisioning a new server or restoring a server is more efficient with snapshot capabilities built into most virtualization tools. Some test management tools now utilize virtualization technologies to snapshot servers at the point when a fault is detected, allowing testers to share the snapshot with the developers investigating the fault.

4. References

4.1 Standards

- [DO-178B] RTCA/FAA DO-178B, Software Considerations in Airborne Systems and Equipment Certification, 1992.
- [ISO25000] ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE), 2005.

4.2 ISTQB Documents

- [ISTQB_ALTA_SYL] ISTQB Advanced Level Test Analyst Syllabus, Version 2012
- [ISTQB_ALTM_SYL] ISTQB Advanced Level Test Manager Syllabus, Version 2012
- [ISTQB_FA_OVIEW] ISTQB Foundation Level Agile Tester Overview, Version 1.0
- [ISTQB_FL_SYL] ISTQB Foundation Level Syllabus, Version 2011

4.3 Books

[Aalst13] Leo van der Aalst and Cecile Davis, “TMap NEXT® in Scrum,” ICT-Books.com, 2013.
[Adzic09] Gojko Adzic, “Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing,” Neuri Limited, 2009.
[Anderson13] David Anderson, “Kanban: Successful Evolutionary Change for Your Technology Business,” Blue Hole Press, 2010.
[Beck02] Kent Beck, “Test-driven Development: By Example,” Addison-Wesley Professional, 2002.
[Beck04] Kent Beck and Cynthia Andres, “Extreme Programming Explained: Embrace Change, 2e” Addison-Wesley Professional, 2004.
[Black07] Rex Black, “Pragmatic Software Testing,” John Wiley and Sons, 2007.
[Black09] Rex Black, “Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, 3e,” Wiley, 2009.
[Chelimsky10] David Chelimsky et al, “The RSpec Book: Behavior Driven Development with Rspec, Cucumber, and Friends,” Pragmatic Bookshelf, 2010.
[Cohn04] Mike Cohn, “User Stories Applied: For Agile Software Development,” Addison-Wesley Professional, 2004.
[Crispin08] Lisa Crispin and Janet Gregory, “Agile Testing: A Practical Guide for Testers and Agile Teams,” Addison-Wesley Professional, 2008.
[Goucher09] Adam Goucher and Tim Reilly, editors, “Beautiful Testing: Leading Professionals Reveal How They Improve Software,” O'Reilly Media, 2009.
[Jeffries00] Ron Jeffries, Ann Anderson, and Chet Hendrickson, “Extreme Programming Installed,” Addison-Wesley Professional, 2000.
[Jones11] Capers Jones and Olivier Bonsignour, “The Economics of Software Quality,” Addison-Wesley Professional, 2011.
[Linz14] Tilo Linz, “Testing in Scrum: A Guide for Software Quality Assurance in the Agile World,” Rocky Nook, 2014.
[Schwaber01] Ken Schwaber and Mike Beedle, “Agile Software Development with Scrum,” Prentice Hall, 2001.
[vanVeenendaal12] Erik van Veenendaal, “The PRISMA approach”, Uitgeverij Tutein Nolthenius, 2012.
[Wiegers13] Karl Wiegers and Joy Beatty, “Software Requirements, 3e,” Microsoft Press, 2013.

4.4 Agile Terminology

Keywords which are found in the ISTQB Glossary are identified at the beginning of each chapter. For common Agile terms, we have relied on the following well-accepted Internet resources which provide definitions.

http://guide.Agilealliance.org/
http://whatis.techtargetcom/glossary
http://www.scrumalliance.org/

We encourage readers to check these sites if they find unfamiliar Agile-related terms in this document. These links were active at the time of release of this document.

4.5 Other References

The following references point to information available on the Internet and elsewhere. Even though these references were checked at the time of publication of this syllabus, the ISTQB cannot be held responsible if the references are not available anymore.

- [Agile Alliance Guide] Various contributors, http://guide.Agilealliance.org/.
- [Agilemanifesto] Various contributors, www.agilemanifesto.org.
- [Hendrickson]: Elisabeth Hendrickson, “Acceptance Test-driven Development,” testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview.
- [INVEST] Bill Wake, “INVEST in Good Stories, and SMART Tasks,” xp123.com/articles/invest-in-good-stories-and-smart-tasks.
- [Kubaczkowski] Greg Kubaczkowski and Rex Black, “Mission Made Possible,” www.rbcs-us.com/images/documents/Mission-Made-Possible.pdf.

5. Index