Certified Tester

Advanced Level Syllabus
Agile Technical Tester (ATT)

Version 1.1

International Software Testing Qualifications Board

Copyright Notice

Revision History

Version	Date	Remarks
Syllabus v0.1	11 January 2017	Standalone sections
Syllabus v0.2	24 May 2017	WG review comments on v01 incorporated
Syllabus v0.3		WG review comments on v02 incorporated
Syllabus v0.7		Alpha review comments on v03 incorporated
Syllabus v0.71		Working group updates on v07
Syllabus v0.9	30 December 2017	Alpha candidate
Syllabus v0.91	6 January 2018	Alpha release
Syllabus v0.98	12 January 2019	Beta candidate
Syllabus v0.99		Beta release
Syllabus 2018		GA version
Syllabus 2019	12 February 2019	Minor changes in Text Added several LOs (K1, K2 levels) Added Copyright information Added learning time in all chapters
Syllabus 2019	14 March 2019	Changed LOs to adapt to official short name “ATT” Modified “Acknowledgements”
Syllabus 2019	06 May 2019	Some minor changes
Syllabus 2019	07 Jun 2019	Minor changes
Syllabus 2019	08 Jul 2019	Redactional changes
Syllabus 2019	10 Jul 2019	Preparation for Beta Review
Syllabus 2019	14 Sep 2019	Enhancements, corrections post Beta Review
Syllabus 2019	14 November 2019	Release date
Syllabus 2019	09 December 2019	Changed the ISTQB logo ® Changed the K-level training duration (Chapter 0.6)

Table of Contents

Acknowledgements

This document was produced by a team from the International Software Testing Qualifications Board Agile Working Group lead by Rex Black (chair), Michaël Pilaeten (Vice-chair and chair a.i.) and Renzo Cerquozzi (Product Owner).

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

At the time the Advanced Level Agile Technical Tester syllabus was completed, the Working Group had the following membership: Michaël Pilaeten (Chair a.i.), Renzo Cerquozzi (Product Owner), Alon Linetzki (Marketing workgroup), Leanne Howard (Glossary workgroup) and Klaus Skafte (Exam workgroup).

Authors: Leo van der Aalst, Renzo Cerquozzi, Bertrand Cornanguer, István Forgács, Jani Haukinen, Noam Kfir, Sammy Kolluru, Alon Linetzki, Tilo Linz, Michaël Pilaeten, Marie Walsh.

Internal Reviewers: Michael Arefi, Vojtěch Barta, Renzo Cerquozzi. Graham Bath, Laurent Bouhier, Anders Claesson, Alessandro Collino, David Janota, David Evans, Leanne Howard, Matthias Hamburg, Kari Kakkonen, Tor Kjetil Moseng, Meile Posthuma, Salvatore Reale, Marko Rytkönen, Sarah Savoy, Klaus Skafte, Mike Smith, Chris van Bael.

Creation and Review of Sample Questions: Armin Born, Renzo Cerquozzi, Alon Linetzki, Tilo Linz, Jamie Mitchell, Jani Haukinen, Tobias Horn, Chris van Bael, Suruchi Varshney.

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: Adam Roman, Armin Beer, Beata Karpinska, Chris Van Bael, Erwin Engelsma, Giancarlo Tomasig, Gary Mogyorodi, Ingvar Nordström, Jana Gierloff, Jörn Münzel, Jurian van de Laar, Kari Kakkonen, Laurent Bouhier, Marko Rytkönen, Martin Klonk, Matthias Hamburg, Meile Posthuma, Paul Weymouth, R. Green, Richard Seidl, Rik Marselis, Stephanie Ulrich, Stephanie van Dijck, Tal Pe'er, Tilo Linz, Veronica Seghieri and Wim Decoutere.

A special thanks to Galit Zucker, ISTQB® general secretary, for the Guidance and Support.

This document was formally approved for release by the General Assembly of the ISTQB® on October 18th, 2019.

0 Introduction to this Syllabus

0.1 Purpose of this Document

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

- To member boards, to translate into their local language and to accredit training providers. Member boards may adapt the syllabus to their particular language needs and add references to adapt to their local publications.
- To certification bodies, to derive examination questions in their local language adapted to the learning objectives for this syllabus.
- To training providers, to produce courseware and determine appropriate teaching methods.
- To certification candidates, to prepare for the certification exam (either 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 from the ISTQB®.

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 Advanced Level Agile Technical Tester Overview document [ISTQB_ATT_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 software and systems testing, and as a basis for books and articles.

0.3 Examinable Learning Objectives

The Learning Objectives support the Business Outcomes and are used to create the examination for achieving the Advanced Agile Technical 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 learning objectives at K2, K3 and K4 levels are shown at the beginning of the pertinent chapter.

0.4 The Advanced Level Agile Technical Tester Certificate Exam

The Advanced Level Agile Technical Tester Certificate exam will be based on this syllabus. Answers to exam questions may require the use of material based on more than one section of this syllabus. All sections of the syllabus are examinable, except for the Introduction and Appendices. Standards, books, and other ISTQB® syllabi are included as references, but their content is not examinable,

beyond what is summarized in this syllabus itself from such standards, books, and other ISTQB® syllabi.

The format of the exam is multiple choice.

Exams may be taken as part of an accredited training course or taken independently (e.g., at an exam centre or in a public exam). Completion of an accredited training course is not a pre-requisite for the exam.

0.5 Accreditation

An ISTQB® Member Board may accredit training providers whose course material follows this syllabus.
Training providers should obtain accreditation guidelines from the Member Board or body that performs the accreditation. An accredited course is recognized as conforming to this syllabus, and is allowed to have an ISTQB® exam as part of the course.

0.6 How this Syllabus is organized

There are four chapters with examinable content.

The top-level heading for each chapter specifies the total time for the chapter; timing is not provided below chapter level.

For accredited training courses, the syllabus requires a minimum of **16** hours of instruction, distributed across the chapters as follows:

- Chapter 1: 180 minutes Requirements Engineering
- Chapter 2: 540 minutes Testing in Agile
- Chapter 3: 135 minutes Test Automation
- Chapter 4: 105 minutes Deployment and Delivery

given a minimum time duration to cover each learning objective as follow:
K2 : 15 minutes
K3 : 60 minutes
K4 : 75 minutes

1 Requirements Engineering 180 minutes

Keywords
Acceptance criteria, epic, user story

Learning Objectives for Requirements Engineering

ATT-1.x (K1) Keywords

1.1 Requirements Engineering Techniques

ATT-1.1.1-1 (K4) Analyze user stories and epics using requirements engineering techniques

ATT-1.1.1-2 (K2) Describe the requirements engineering techniques and how they can help testers

ATT-1.1.2-1 (K4) Create and evaluate testable acceptance criteria for a given user story using requirements engineering and test techniques

ATT-1.1.2-2 (K2) Describe the elicitation techniques

1.1 Requirements Engineering Techniques

Applying requirements engineering techniques allows Agile teams to polish the user stories (see [AgileFoundationExt]) and epics (see [AgileFoundationExt]), add context, consider impacts and dependencies, and identify any gaps, such as missing non-functional requirements.
Although the majority of the requirements engineering techniques discussed in this section come from traditional development approaches, they are also effective in Agile development.
Generally, in traditional projects, requirements engineering activities and techniques are formalized, performed sequentially, and are the responsibility of designated people such as Business Analysts, Functional Analysts, Technical Architects, Enterprise Architects, and Process Analysts. In contrast, in Agile projects, requirements engineering techniques are applied throughout the project and during each iteration via a less formal approach. These requirements engineering techniques are performed more frequently, using continuous feedback loops, by all Agile team members, not just the dedicated business analyst or product owner of the team.

1.1.1 Analyze user stories and epics using requirements engineering techniques

As a tester, to be able to assist in clarifying (and possibly improving) user stories, epics, and other Agile requirements, it is necessary to know, understand, select and use the various requirements engineering techniques that support this.
Example of such techniques are storyboards, story mapping, personas, diagrams, and use cases.

- **Storyboards**: A Storyboard (not to be confused with Agile task board or Agile user story board) provides a visual representation of the system. Storyboards help testers to:
  - See the thought process behind the user stories, and the overall “story”, providing context, making it possible to quickly see the functional flow of the system and identify any gaps in the logic.
  - Visualize groups of user stories related to a common area of the system (Themes) which can be considered for inclusion in the same iteration, as they will likely be touching the same piece of code.
  - Assist in Story Mapping and prioritization of epics and related user stories in the product backlog.
  - Assist with identifying acceptance criteria for user stories and epics.
  - Assist with selecting the right test approach based on the visual aspect on the system design.
  - Along with Story Mapping, assist in prioritizing tests and in identifying needs for stubs, drivers and/or mocks.

- **Story Mapping:** Story Mapping (or user story Mapping) is a technique which consists of using 2 independent dimensions to order user stories. The horizontal axis of the map represents the order of priority of each of the user stories, whilst the vertical axis represents the sophistication of the implementation. Use of Story Mapping can help testers to:
  - Determine the most basic functionalities of a system to distillate a smoke test.
  - Identify the order of functionalities to determine test priorities.
  - Visualize the scope of the system.
  - Determine the risk level of each user story.

- **Personas**: Personas are used to define fictional characters or archetypes that illustrate how typical users will interact with the system. Use of personas can help testers to:
  - Identify gaps in user stories by identifying different types of users that may use the system.
  - Identify inconsistencies in user stories on how a particular type of user may use the system compared to others.

- Elicitate user story acceptance criteria
- Discover additional test paths during exploratory testing
- Reveal test conditions, especially those related to particular user groups, thus helping to ensure sufficient user group coverage and testing of the differences between user groups.

- **Diagrams**: Diagrams such as entity relationship diagrams, class diagrams, and (other) UML diagrams can show the structure or flow of data and the functional attributes or behavior of the system, and may be used to identify gaps in system functionality.

- **Use Cases**: Use Cases (diagrams and specifications) [Foundation] can assist testers to:
  - Ensure user stories are testable and properly sized.
  - Determine whether the user stories need to be refined or decomposed.
  - Reveal forgotten stakeholders.
  - Identify interfaces and integration points, which should be considered during test design.
  - See the relationships between epics and user stories, to check that the epic doesn't have any missing user stories.

1.1.2 Identifying acceptance criteria using requirements engineering and test techniques

Requirements engineering is a process consisting of the following steps:

- **Elicitation**: The process of discovering, understanding, documenting, and reviewing the users' needs and constraints for the system. Elicitation techniques should be used to derive, evaluate and enhance the acceptance criteria.
- **Documentation**: The process of documenting the users' needs and constraints clearly and precisely. User stories and acceptance criteria should be documented to level corresponding to the team adherence over the principles of the Agile manifesto. The type of documentation depends on the team's and stakeholders' approach. Acceptance criteria can be documented using natural language, using models (e.g., state transition diagrams) or using examples.
- **Negotiation and Validation:** For every user story, multiple stakeholders might have other insights or preferences. As these insights and preferences might be inconsistent or even conflicting, so can each stakeholder's acceptance criteria. Each of these conflicts should be identified, negotiated, and resolved between all impacted stakeholders. Every forgotten or non-resolved conflict might endanger the success of the project. At the end of this step, the content of each user story is validated by the impacted stakeholders (e.g. as a Definition of Ready).
- **Management**: As projects progress, opinions and circumstances might change. Even though acceptance criteria were properly elicited, documented, negotiated, and validated, acceptance criteria are still subject to change. Because of the potential for changes, user stories should be managed by using good configuration and change management processes.

For identifying acceptance criteria, multiple elicitation techniques are at the disposal of the tester, including the following:

- **Quantitative Questionnaires**: Using quantitative data culled from closed-ended questions is an excellent way of making clear comparisons between various data points. This will often provide numerical data that can be included in a numerical conclusion for an acceptance criterion. The quantitative questionnaire can be used as an elicitation technique for large number of stakeholders and specifically for non-functional acceptance criteria.

- **Qualitative Questionnaires:** Open-ended questions are an extremely effective way to add more quality to quantitative research. Open-ends are best used as a follow-up to key questions. This could generate additional information for which new User Stories have to created or has to be added to existing ones. The qualitative questionnaire can be used as an elicitation technique for smaller number of stakeholders – as the processing takes more time – and is suitable for functional acceptance criteria.
- **Qualitative Interview:** The qualitative interview is more flexible than a quantitative query and is mainly used to acquire information about backgrounds, contexts and causes. It is unlikely to return hard data, but acceptance criteria can be derived from the responses regarding the context of a user story. The qualitative interview can be the follow-up of any type of questionnaire, to deepen the derived acceptance criteria.

Multiple other elicitation techniques exist, in the range of observation techniques (e.g. apprenticing), creativity techniques (e.g. 6 Thinking hats) and supporting techniques (e.g. low-fi prototyping). The technique toolset of the tester will influence the quality of the elicitated acceptance criteria.

INVEST and SMART [INVEST], along with test techniques such as equivalence partitioning, boundary value analysis, decision tables, and state transition testing [Foundation] can be used to identify and evaluate acceptance criteria as well.

2 Testing in Agile 540 minutes

Keywords
Test-driven development , behavior-driven development, acceptance test-driven development, specification by example, test charter

Learning Objectives for Testing in Agile

ATT-2.x (K1) Keywords

2.1 Agile development and test techniques

ATT-2.1.1-1 (K3) Apply test-driven development (TDD) in the context of a given example in an Agile project

ATT-2.1.1-2 (K2) Understand the characteristics of a Unit test

ATT-2.1.1-3 (K2) Understand the meaning of the mnemonic word FIRST

ATT-2.1.2-1 (K3) Apply behavior-driven development (BDD) in the context of a given user story in an Agile project

ATT-2.1.2-2 (K2) Understand how to manage guidelines for a formulation of a scenario

ATT-2.1.3 (K4) Analyze a product backlog in an Agile project to determine a way to introduce acceptance test-driven development (ATDD)

2.2 Experience-based testing in Agile

ATT-2.2.1-1 (K4) Analyze the creation of a test approach using test automation, experience-based tests and back-box tests, created using other approaches (including risk-based testing) for a given scenario in an Agile project

ATT-2.2.1-2 (K2) Explain differences between Mission critical and non-critical

ATT-2.2.2-1 (K4) Analyze user stories and epics to create test charters

ATT-2.2.2-2 (K2) Understand the use Experienced-based techniques

2.3 Aspects of code quality

ATT-2.3.1-1 (K2) Understand the importance of refactoring test cases in Agile projects

ATT-2.3.1-2 (K2) Understand practical task-list for Refactoring Test cases

ATT-2.3.2-1 (K4) Analyze code as part of a code review to identify defects and technical debt

ATT-2.3.2-2 (K2) Understand Static code Analysis

2.1 Agile Software Development and Test Techniques

In software development, defects can occur from poorly-written code and from failure to meet customer needs. Agile development techniques handle these problems by applying the concepts of test-driven development (TDD), behavior-driven development (BDD), and acceptance test-driven development (ATDD). TDD is a technique to improve software's product quality, while BDD and ATDD help improve its quality in use (features and functionality).

2.1.1 Test-driven development (TDD)

Test-driven development is a software development method that combines design, testing, and coding in a rapid iterative process. TDD adopts a disciplined test-first approach of creating a test that expresses the intended functionality of the code before coding that functionality. The test is run before the code itself is written, in order to check that the test fails. Once the code is written, the test is executed again and should pass.

TDD is generally considered to be the first major test-first programming methodology from which other methodologies, such as behavior-driven development (BDD), acceptance test-driven development (ATDD) and specification by example (SBE), are derived.

TDD offers an evolutionary approach to software development that treats design as a continuous, ongoing process. Each iteration constitutes a small change to the production code and provides an opportunity for the developer to slightly improve the design. As the changes and carefully considered design decisions accumulate step by step, a robust and well-tested design emerges.

TDD practitioners use a number of practices and techniques to write high-quality code and avoid the accumulation of technical debt, including:

- Unit tests and a prescriptive test-first approach
- Short iterative cycles
- Immediate and frequent feedback
- Effective use of tooling, source control, and continuous integration
- The guided application of programming principles and design patterns

TDD practitioners write unit tests to check the expected behavior of the production code. A unit test executes a function under specific conditions and checks whether it produces an expected result or not. Expectations are described with assertions that check that the system state changes as expected, or that specific behavior is exhibited. Assertions can check, for example:

- That a function performs a calculation and returns an expected result
- That a function modifies the state of the system in a particular way
- That a function calls another function in a specific way

Unit tests should be easy for developers to implement and maintain. They are automated and are often written using the same language as the production code. Unit tests should have the following characteristics:

- Deterministic – Every time a unit test is run under the same conditions, it should produce the same results.
- Atomic – The unit test should only test the functionality related to it
- Isolated – A unit test should strive to exercise only the specific code for which it was initially intended. Unit tests should not depend on each other and should avoid dependencies in the production code whenever possible.
- Fast – Unit tests should be small and fast so they can provide immediate feedback. It should be possible to run many unit tests in a short period of time.

Another mnemonic to describe a good unit test is FIRST: Fast, Isolated, Repeatable, Self-Validating, Thorough

There are many unit test frameworks available for many different languages. They differ in their APIs, approaches and terminology, but all share some common elements. Regardless of the language or unit test framework, a unit test typically follows this three-steps pattern:

1. Arrange/Setup – prepare the execution environment. This step may instantiate objects, initialize system state, and inject data as necessary.
2. Act – execute the operation being tested, then check the result produced by the operation. This mandatory step may consists of just one line of code that triggers the operation being tested.
3. Assert - perform the actual check of expected outputs or other post-conditions.

The iterative process is the cornerstone of TDD. The purpose of each iteration is to make a small, focused, carefully considered change to the program. Following the color-coded convention, the iterative cycle is often called the red-green-refactor cycle:

- Red – Write a failing test that describes an unimplemented expectation. Run the test to ensure that it fails.
- Green – Write production code that satisfies only the expectation described by the test and causes it to pass. Run all the relevant tests to ensure that they all pass. If any fail, make the changes necessary to cause all the tests to pass.
- Refactor – Improve the design and structure of both the test code and the production code without changing functionality, while ensuring that all the relevant tests continue to pass. Developers may apply a sequence of refactoring to change the code without changing the behavior until the code is optimized. Most modern Integrated Development Environments (IDE's) and many code editors can apply refactoring automatically and safely.

2.1.2 Behavior Driven Development (BDD)

According to the ISTQB® Agile Foundation syllabus, behavior-driven development (BDD) is a technique in which developers, testers, and business representatives work together to analyze the requirements of a software system, formulate them using a shared language, and verify them automatically.

BDD is strongly influenced by two separate disciplines: test-driven development (TDD) and domain-driven design (DDD) [Evans03]. It incorporates many of the core principles of both disciplines, including collaboration, design, ubiquitous language, automated test execution, short feedback cycles, and more. TDD relies on unit tests to verify implementation details whereas BDD relies on executable scenarios to verify behaviours and more. BDD generally follows a prescribed path:

- Create user stories collaboratively
- Formulate user stories as executable scenarios and verifiable behaviours
- Implement behaviours and execute the scenarios to verify them

Teams that apply BDD extract one or more scenario from each user story and then formulate them as automated tests. A scenario represents a single behavior under specific conditions.
Scenarios are typically based on user story's acceptance criteria, examples, and use cases. BDD scenarios should be written using language that can be understood by all team members, whether technical or non-technical.
Therefore, a strong preference is given for using natural language, such as English, for expressing scenarios. Furthermore, teams that apply BDD establish a ubiquitous language [Evans03], which essentially constitutes terminology that is clear and unambiguous to everyone on the team and is used everywhere.

BDD scenarios are typically composed of three main sections [Gherkin]:

1. Given – describes the state of the environment (preconditions) before the behavior is triggered
2. When – describes the actions that trigger the behavior
3. Then – describes the expected results of the behavior

Extracting scenarios from user stories involves:

- Identifying all acceptance criteria specified by a user story and writing scenarios for each criterion. Some may require multiple scenarios.
- Identifying functional use cases and examples and writing scenarios for each of them. Many use cases and examples are conditional, so it's crucial to identify each of the conditions.
- Creating scenarios while testing exploratory, which can help identifying related existing behaviors, potential conflicting behaviors, minimal states, alternative flows, etc.
- Looking for repeated use of steps or groups of steps to avoid repetitive work.
- Identifying areas that require random or synthetic data.
- Identifying steps that require mocks, stubs or drivers to maintain isolation and to avoid executing integrations or possibly costly processes.
- Ensuring scenarios are atomic and do not affect each other's state (isolated).
- Deciding whether to limit the When sections to one step in accordance with the principle that every test checks just one thing, or to optimize for other considerations, such as test execution speed.

Recommended guidelines for formulating scenarios include:

- The scenario should describe a specific behavior that the system supports from the perspective of a specific user.
- The scenario should use the third person when describing the steps (Given, When, Then) to describe the state and interactions from the perspective of the user.
- The scenarios should be isolated and atomic so that they can be run in any order and not affect each other or rely on each other. The Given steps should place the system in the state necessary for the When steps to consistently execute as expected.
- The When steps should describe the semantic actions that a user performs rather than the specific technical actions, unless there is a particular need to test a specific action. For example, "The user confirms the order" (a semantic action) is generally preferable to "The user clicks the Confirm button" (a technical action) unless the button itself has to be tested.
- The Then steps should describe specific observations or states. They should not specify generic success or error states.

2.1.3 Acceptance test-driven development (ATDD)

Acceptance test-driven development (ATDD) supports the coordination of software projects to address the delivery based on the customer needs. Acceptance tests are specifications for the desired behavior and functionality of a system. A user story represents a piece of customer-valued functionality. Acceptance tests verify that this functionality is implemented correctly. This user story is split by the developers into a set of tasks required to implement that functionality. Developers can implement these tasks using TDD. When a given task is completed, the developers move on to the next task, and so forth, until the user story is completed, which is indicated by successfully executed acceptance tests. Both BDD and ATDD are customer focused whereas TDD is developer focused. BDD and ATDD are similar in that they both produce the same result: a shared understanding of what is to be built and how to build the thing right. BDD is a structured way of writing test cases (e.g. acceptance tests) using the Given/When/Then syntax discussed earlier.

ATDD separates the test intention, which is expressed in a human readable format such as plain text, from the test implementation, which is preferably automated. This important separation means that business-facing and other non-technical team members such as product owners, analysts, and testers can take an active role in describing testable examples (or acceptance tests) to drive development. Stakeholders with different roles and perspectives, such as customer, developer, and tester come together to collaboratively discuss a potential user story (or other form of requirement) to reach a shared understanding of the problem to be solved, to ask each other open questions regarding the functionality and to explore concrete, testable examples of required behavior.

The agreed examples (and the discussions that lead to them) are valuable outputs in themselves, so it is important to remember that although they may also be automated as acceptance tests, it is not always necessary or cost-effective to do so. This emphasis on the idea of valuable *examples* rather than *tests* is intentional, and is an important trick for encouraging all members of a team to be engaged in the discovery process.

This illustrates an important role for the Agile tester in this situation. During the discussions, the tester may be thinking in terms of test techniques, such as equivalence partitioning, boundary value analysis, and so on. They can use this analytical approach to encourage questions to be asked to the product owner about where the interesting behaviors and edge cases can be found. The intention should always be to encourage the whole group to consider and find the *key examples*. Posing an interesting combination of inputs and asking if the group is in agreement about the expected outputs is a good way to explore potential areas of uncertainty or incomplete analysis.

Specification by example (or SBE) refers to a collection of useful patterns that allow an Agile team to discover, discuss, and confirm the important outcomes required by business stakeholders and the software behaviors that are needed to reach those outcomes. The term has been widely used to include and broaden the meaning of the term acceptance test-driven development (ATDD).

2.2 Experience-based testing in Agile

The various characteristics of Agile projects such as the approach, length of iteration, applicable test level(s), risk level of project and product, quality of requirements, level of experience/expertise of the team members, project organization, etc. will influence the balance of automated tests, exploratory tests, and manual black-box tests in an Agile project.

2.2.1 Combining experience-based techniques and black-box tests

During risk analysis, the risk levels (e.g. high, medium, low) are determined for the individual system features and functionality. The next step is to find the right mix and balance of automated tests, exploratory tests, and manual black-box tests for a specific risk level. Below is a table describing this idea. In this table, the risk levels are listed vertically and the three test approaches horizontally. The following symbols are used to describe:

++	(highly recommended)
+	(recommended)
o	(neutral)
-	(not recommended)
--	(not to be used)

The table below is an example of a mix of different testing techniques (exploratory, automated, and manual) that can be used in a safety critical system. This table can be adapted to other specific projects.

Risk Level	Automated Tests	Exploratory Tests	Black-box Tests
High	++	+	++
Medium	+	+	+
Low	o	++	+

Table 1: Mission and/or Safety Critical Systems

When looking at the first row of Table 1, it suggests that in this situation, a combination of automated test and black-box test would be highly recommended in addition to an exploratory testing approach. The decision to automate (or not) will also be influenced by many other factors.

The following is an example of a combination of different testing techniques when used in a non-safety critical system.

Risk Level	Automated Tests	Exploratory Tests	Black-box Tests
High	+	++	+
Medium	o	++	o
Low	--	++	--

Table 2: Non-Mission and/or Non-Safety Critical Systems.

When looking at the last row of the Table 2 above, it suggests that in this situation, an exploratory testing approach is highly recommended, while other approaches might not be used.
In any situation (safety critical or not) the specific mixture will depend on the project characteristics.

2.2.2 Creating test charters and interpreting their results

Before an appropriate test charter can be created, the existing epics and user stories should be evaluated first. See chapter 1 for techniques

When analyzing epics or user stories to create a test charter the following things should be considered:

- Who are the users in this epic or user story?
- What is the main functionality of the epic or user story?
- What are the actions a user can perform? (This can be obtained from the acceptance criteria list, which is defined for a user story)
- Is the goal of the user story realized once the feature or functionality is complete? (Or are there other testing tasks affecting the Definition of Done)

The granularity of a test charter is important. It should not be too small, as it must explore an area around an identified problem (re-active, regression) or an area around a user story or an epic (pro-active, revealing).

It shouldn't be too big, as it should fit within a 60 to 120 minutes time box. The goal of running an exploratory testing session is to help us make a good quality decision, regarding a phenomenon, bug clustered area, etc.. The results of that exploration should give enough information to make that decision.

Test charters can be created using flipcharts, spreadsheets, documents, existing test management system, personas, mind mapping and a whole team approach. Exploratory testers use heuristics to drive their creativity in writing and performing exploratory testing sessions. These can also be used for creating test charters and in thinking creatively when analyzing user stories and epics. Examples of heuristics can be found in [Whittaker09] and [Hendrickson13].

All the findings found during exploratory testing should be documented. The results of exploratory testing should provide insights into better test design, ideas for testing the product, and ideas for any further improvement. Findings that should be documented during exploratory testing include defects, ideas, questions, improvement suggestions, etc..

Tools can be used for documenting the exploratory testing sessions. These include video capture and logging tools, planning tools, etc. The documentation should include the expected result. In some cases, pen and paper is sufficient, depending on the volume of information to collect.

When summarizing the exploratory testing session, during the debriefing meeting the information is collected and aggregated to present a status of progress, coverage, and efficiency of the session. This summary information can be used as management report, or used in retrospective meetings on any level and any scale (single team, multiple teams, and large scale Agile implementation). However, it can be quite challenging to determine accurate test metrics related to exploratory testing sessions.

2.3 Aspect of Code Quality

The control of technical debt, especially in terms of maintaining high levels of code quality throughout the release, is very important in Agile projects. Various techniques are used to achieve this goal.

2.3.1 Refactoring

Refactoring is a way to clean up code in an efficient and controlled manner, by clarifying and simplifying the design of existing code and test cases, without changing its behavior. In Agile projects, the iterations are short, which creates a short feedback loop for all team members. Short iterations also create a challenge for testers trying to achieve adequate coverage. Due to the nature of the iterations, the fact that functionality is growing, and features are added and enhanced over time, test cases that have been written for a feature in an earlier iteration, often need maintenance or even complete redesign in later iterations. Using an evolutionary test design approach, updating and refactoring of the tests can compensate for the changes of features, and ensure that tests remain aligned with the product's functionality.

After user stories are understood and acceptance criteria are written for each of them, the impact of the functionality of the current iteration on the existing regression tests (manual and automated) can be analyzed, and refactoring and/or enhancing of the tests may be required. Teams are maintaining and extending their code extensively from iteration to iteration, and without continuous refactoring, this is hard to do.

The refactoring of the test cases can be done as follows:

- **Identification**: Identifying existing tests that require refactoring by reviewing or causal analysis.
- **Analysis**: Analyzing the impact of the changed tests on the overall regression test set.

- **Refactor**: Making changes to the internal structure of the tests to make it easier to understand and cheaper to modify without changing its observable behavior.
- **Re-run**: Re-running the tests, checking their results and documenting defects where relevant. The refactoring should not have influenced the result of test execution.
- **Evaluate**: Checking the results of the re-run tests, ending this phase when the tests have passed a quality threshold defined and accepted by the team.

2.3.2 Code reviews and static code analysis to identify defects and technical debt

A code review is a systematic examination of code by two or more people (one of whom is usually the author). Static code analysis is the systematic examination of code by a tool. Both are effective and widespread practices for exposing and identifying issues that affect code quality. Code reviews and static code analysis provide constructive feedback that helps identify defects and manage technical debt.

Impediments such as resource constraints, higher technical complexity than expected, rapidly changing priorities, and technical limitations may hinder efforts to write quality code and force programmers to make compromises that will decrease the quality of the code in favour of more immediate results. These compromises can introduce defects and incur technical debt.

Technical debt refers to the increased effort that will be required to implement a better solution (including removing latent defects) in the future as a consequence of choosing an inferior but easier-to-implement solution now. Technical debt is often incurred unintentionally, by subtle compromises or a gradual accumulation of small or unnoticed changes as the software evolves. Code reviews and static code analysis help identify different causes of technical debt, such as increased complexity, circular dependencies, conflicts between different code modules, poor code coverage, insecure code, and so forth. Other types of technical debt can also occur on e.g. testing artifacts, infrastructure and the CI pipeline.

When technical debt is incurred deliberately (as a consequence of other decisions, or as a compromise) or identified by code reviews or static code analysis, an effort should be made to reduce the technical debt. Dealing with it immediately is preferable. If it cannot be dealt with immediately, tasks to deal with the technical debt should be added to the (product) backlog.

The trade-off between, on the one hand, taking the additional time necessary to analyze and review the code and, on the other hand, incurring technical debt, almost always favors code analysis and reviews. When code with defects and technical debt spreads, it becomes increasingly difficult, costly, and time-consuming to correct without harmfully affecting other parts of the system. Code analysis and reviews can improve the quality of the code and likely reduce the overall time expenditure.

Besides helping identify defects and manage technical debt, code analysis and reviews offer additional benefits:

- Training and sharing knowledge
- Improving the robustness, maintainability, and readability of the code
- Providing oversight and maintaining consistent coding standards

Code Reviews

By participating in code reviews, testers can use their unique perspective to make valuable contributions to code quality by collaboratively working with programmers to identify potential defects and avoid technical debt at a very early stage. Testers should be competent to read the programming language used in the code they review, but they don't need to have significant coding skills to participate effectively in code reviews. They can bring their expertise to bear in many ways, such as by

asking questions about the behavior of the code, suggesting use cases that may not have been considered, or monitoring code metrics that could indicate quality issues. Code reviews also provide an opportunity to share knowledge between programmers and testers. One of the main challenges of code reviews in Agile projects is short iterations and time needed to do the code reviews. It is important to plan for code reviews and set aside the time needed to do such reviews during each iteration.

Code reviews are manual activities, possibly supported by tools, that are performed by or with other people (in addition to the author). Usually team leaders or more experienced programmers on the team will perform the code review, though they can be done with other team members as well. It is often beneficial for testers and other non-programmers to participate in code reviews.

Different approaches to code reviews vary in their level of formality and rigor [Wiegers02]. More formal and rigorous approaches tend to be more thorough but are also more time-consuming. Less formal and rigorous approaches are a bit less thorough but can be much faster. There are different types of peer reviews, and Agile teams tend to prefer quicker reviews performed frequently, usually before integration.

Code reviews may be performed with the reviewer and the author/developer sitting side-by-side. This mode, which is common in ad-hoc reviews and pair programming, facilitates excellent communication and encourages deeper analyses and better knowledge sharing. It can also contribute to team cohesion and morale.

On distributed teams or teams that prefer a more disconnected approach, the code review process is facilitated by the configuration management system. The process is usually partially automated as part of the continuous integration process. These processes may support code reviews with a single reviewer in each round of review, or collaborative team reviews.

Static Code Analysis

In static code analysis, a tool analyzes the code and searches for specific issues without executing the code. The results of static code analysis may point at clear issues in the code or provide indirect indicators that require further assessment.

Many development tools, especially integrated development environments (IDEs), can perform static code analysis while writing the code. This provides the benefit of immediate feedback, though it may only be possible to apply just a subset of the analysis performed during continuous integration.

3 Test Automation 135 minutes

Keywords

data-driven testing, keyword-driven testing, test procedure, test approach

Learning Objectives for Testing Process

ATT-3.x (K1) Keywords

3.1 Test Automation Techniques

ATT-3.1.1 (K3) Apply data-driven and keyword-driven test techniques to develop automated test scripts

ATT-3.1.2 (K2) Understand how to apply test automation to a given test approach in an Agile environment

ATT-3.1.3-1 (K2) Understand the test automation

ATT-3.1.3-2 (K2) Understand differences between various test approaches

3.2 Level of Automation

ATT-3.2.1-1 (K2) Understand the factors to consider when determining the level of test automation needed to keep up with the speed of deployment

ATT-3.2.1-2 (K2) Understand the challenges of test automation in agile settings

3.1 Test Automation Techniques

3.1.1 Data-Driven Testing

Motivation

Data-driven testing is a test automation technique which minimizes the efforts needed for developing and maintaining test cases which have identical test steps but have differing combinations of test data inputs. Data-driven testing is a well-established technique which is not specific for testing in Agile projects Because of its capability to lower test automation development and maintenance efforts, this technique should be considered as part of the test automation strategy in every Agile project.

Concept

The basic idea of data-driven testing is to separate the test logic from test data. The test procedure instead names test data variables which reference test data values that are provided in a separate test data list/table. The test procedure can then be repeatedly executed using different sets of test data. Further details and examples can be found in [AdvancedTestAutomationEngineer].

Benefits to Agile Teams

- Agile teams can quickly adjust to a product's changing/increasing functionality from iteration to iteration, since changing/adding new data combinations is simple and has little or no impact on existing automation.
- Agile teams can easily scale the necessary test coverage up or down by adding, changing, or removing entries from the test data table. This also helps Agile teams to control test execution times to meet continuous deployment constraints.
- The technique supports the philosophy of working cross–functionally, because test data tables are easier to understand than test scripts and therefore enable more effective participation from less-technical team members.
- As test data tables are more easily understood, this technique supports early feedback on test cases and acceptance criteria from less-/non-technical team members and customers/users.
- This technique helps Agile teams to complete test automation tasks more efficiently because not only does it lower the effort to develop new tests, it also reduces the maintenance of existing data-driven tests.

Limitations to Agile Teams

While there are no specific limitations to using this technique in an Agile context, Agile teams should be aware of the general limitations to using this approach. Further details can be found in [AdvancedTestAutomationEngineer].

Tools

- Editing, storing and managing the test data is usually done by utilizing spreadsheets or text files.
- Most test automation tools/languages offer built-in commands to read test data from spreadsheets, or text files.

3.1.2 Keyword-Driven Testing

Motivation

One downside of test automation is that automated test scripts are much more difficult to understand than manual test procedures described in natural language. Applying a keyword-driven test automation technique helps to improve readability, understandability, and maintainability of automated test scripts.

Concept

The basic idea of keyword-driven testing is to define a set of keywords taken from a product's use cases and/or the customer's business domain and use them as vocabulary to phrase test procedures. Because of their semi-natural language, the resulting automated test scripts are easier to understand than plain program/script language text.
Further details and examples can be found in [AdvancedTestAutomationEngineer].

There are different styles how a test procedure can be written using keywords (see [Linz 14], Chapter 6.4.2 Keyword-driven Testing):

- List-style: A test procedure is a plain sequence/list of keywords;
- BDD-style: A test procedure (or scenario) is written as a natural-language like sentence using keywords as 'executable' parts of the sentence (see section 2.1.2 (BDD));
- DSL-style: A formal grammar based on the "domain-specific language" (DSL) concept (see [Fowler/Parsons10]) defines how the keywords (and potentially additional language elements) can be combined.

Benefits to Agile Teams

- By defining keywords or a DSL, an Agile team creates and standardizes its domain-related team vocabulary, which helps team members to communicate more clearly and precisely, helping to avoid misunderstandings.
- Agile teams can quickly gather more qualified customer/user feedback. Test procedures composed by keywords and/or written in BDD/DSL-style can be better understood by customers than plain program language test code. Informal acceptance criteria can be formalized without losing 'natural-language readability'. This can help gain an understanding of the business logic itself and thus on the interpretation of acceptance criteria.
- The way the test cases are created attributes to the creation and management of living documentation.
- The technique helps Agile teams to accomplish its test automation tasks in a cross-functional fashion by including the more non-technical members, since composing test procedures from existing keywords does not require programming skills
- Changing the behavior of a defined keyword requires much less effort than changing the same behavior across multiple test procedures. This can greatly reduce maintenance efforts and frees up resources to spend time on the implementation of new tests.
- An Agile team can apply the technique (and thus realize its benefits) across the whole testing pyramid because keywords can be implemented to drive the test object by arbitrary interfaces, from interface level-testing level down to API-level (e.g. via API calls, REST calls, Soap calls etc). This means that the concept is also applicable on unit and integration test cases and not just limited to system testing via the user interface, as it is often assumed. However, it can add an unnecessary abstraction level to the unit tests.

Limitations to Agile Teams

Besides the general limitations of this approach as described in [AdvancedTestAutomationEngineer], Agile teams should also be aware of the following limitations and potential pitfalls:

- To execute keyword-driven test procedures an appropriate execution framework (e.g. including a keyword interpreter) is needed. It's not recommended reinventing the framework from scratch. The team will gain higher velocity by using an existing framework or tool which supports keyword-driven testing. This is especially the case if the team follows a BDD- or DSL-style.
- Implementing a new keyword in a stable, maintainable way is difficult and requires experience and good programming skills. Keyword implementation tasks also compete against product coding tasks. The resulting test automation velocity therefore might be lower than expected.
- The set of keywords (naming, abstraction level) and/or the DSL (grammar rules) must be well designed. Otherwise understandability and/or scalability will not meet expectations.
- The set of keywords also must be properly managed. If this is not done, keyword implementation will not pay off because multiple implementations of synonyms may occur, or keywords are not or are seldom used by test cases. To avoid these pitfalls, the team should make a team member responsible for managing the keyword vocabulary.
- The team should be aware that applying keyword-driven testing requires some upfront investment (e.g. for defining the keywords/domain language, selecting an appropriate framework, implementing the first set of keywords) and at the beginning of a project, test automation velocity might be reduced.

Tools

- As in data-driven testing, a common but limited approach is to use spreadsheets, text or flat files for editing, storing and managing keyword-driven tests.
- Several test frameworks, test execution tools and test management tools offer built-in support for keyword-driven testing (named "keyword-driven testing," "interaction based testing," "business process testing" or "behavior-driven testing," depending on the tool vendor or framework). Available tools can be found in [ToolList].

3.1.3 Applying Test Automation to a Given Test Approach

Test automation is not a test objective. Test automation is a strategy which, when pursued appropriately, can promote larger strategic test objectives by increasing testing efficiency, making testing effective against certain types of defects (e.g., performance and reliability defects), or allowing earlier discovery of defects. The strategy aligns with the context and is thus continuously evolving.

Test automation often takes on many different forms and involves many different tools, depending on the context and needs of the team. In large projects, there is usually not just one solution that fits all needs, and so multiple test automation strategies are relevant. The application of test automation must be appropriate to the test strategy of the organization and to the test approach on a given project.

Test automation is more than just the automation of test execution. Test automation can play an important role in test environment configuration, test release acquisition, test data management and test result comparison, to give just a few examples. When planning and designing the use of such tools, consider the test approach, the implications of the implemented Agile lifecycle, the capabilities of these test automation tools, and the integration of various other tools with these test automation tools (e.g., test automation within the continuous integration framework).

In some cases, test automation directly serves the goals of an iteration; i.e., building new functionality. In other cases, test automation supports these goals indirectly; e.g., by reducing the regression risk associated with changes to the system. As discussed in the Agile Foundation syllabus, some organizations choose to put these supportive test automation efforts into teams outside of the iteration teams themselves. In such organizations, you may for example see a separate team providing

regression test automation framework creation and maintenance as a service to multiple iteration teams. This approach can be successful if the outside team provides useful services to the iteration teams that help those iteration teams focus on their immediate iteration goals. Depending on external teams can influence the commitment of the team, as they transfer (part of) the control over their commitment.

The following are examples of test automation considerations for the major test approaches as stated in the ISTQB® Foundation, Advanced Test Manager, and Expert Test Manager syllabi:

Analytical: Behavior-driven development (BDD) and acceptance test-driven development (ATDD) are techniques which can be used within an analytical approach in an Agile context, e.g. applied to of test automation. BDD and ATDD can be used to produce automated tests in parallel with (or even before) the implementation of the user story.

Model-based: Model-based testing of functional behavior can support automated creation of tests during user story implementation, providing a quick source of efficient tests. Model-based testing can also be used for the creation of user stories, as models can be used to test requirements and aid in static reviews. Models are often used for testing of non-functional behavior such as reliability and performance, which are important characteristics for many systems and which are emergent properties of the system as a whole.

Methodical: Since there are short, multiple iterations in Agile projects, automated test checklists (with all activities) can be used as a methodical approach for the efficient execution of a stable set of tests.

Process-compliant: On projects that must comply with externally defined standards or regulations, these standards or regulations can influence how automated tests are used or how automated test results are captured. For example, on FDA-regulated (Food & Drug Administration; high risk) projects, automated tests and their results must be traceable back to requirements, and the results must contain sufficient details to prove that the test passed.

Reactive (or heuristic): Reactive testing plays an important validation role in Agile testing, while most automated tests primarily play a verification role. While reactive strategies are primarily manual (e.g., exploratory testing, error guessing, etc.), an increased proportion of automated test coverage often leads to a greater degree of manual testing that follows reactive strategies since many of the tests that can be prepared upfront will be automated. Furthermore, the remaining manual tests can cover riskier areas.

Directed (or consultative): When the test coverage is specified by outside stakeholders and test automation is to be used, the ability of the test team to respond to the request is important. Therefore, test teams following a directed test strategy should consider both the time and the skills necessary to complete their tasks within the iterations.

Regression-averse: In Agile projects, a primary characteristic of regression-averse strategy is a large, stable, and growing set of automated regression tests. Adequate coverage, maintainability, and efficient results analysis are critical, especially as the number of regression tests grows. Rather than focusing on an ever-growing set of regression tests, a successful regression-averse approach focuses on continuous improvement and refactoring of created tests.

3.2 Level of Automation

3.2.1 Understand the level of test automation needed

Automation is an important element in Agile projects because it not only covers test automation but also the automation of the deployment process (cf. Chapter 4). Continuous deployment is the automatic deployment of a new version into the production environment. Continuous deployment takes place at regular and short intervals.

Since continuous deployment is an automated process, the automated tests must be sufficient to maintain the required level of code quality. Merely executing automated unit tests is not enough to achieve sufficient test coverage. An automated test suite, executed as part of the deployment process, must also include integration and system level tests. In practice, some manual testing may still be needed.

The following are some challenges faced with test automation in an Agile setting:

- **Test suite volume**: Each iteration (besides maintenance or hardening iterations) implements additional features. To keep the test suite in line with the product's additional functionality, the Agile team must enhance its test suite within each iteration. This means that the number of test cases within the test suite increases by default from iteration to iteration. It requires careful and deliberate efforts to refactor the tests to increase coverage without significantly increasing size. Either way, the effort and time needed to maintain, prepare, and execute the complete test suite will increase over time.
- **Test development time**: The tests needed to verify the product's new or changed functionality have to be designed and implemented. This includes creating or updating the necessary test data and preparing any updates to the test environment. Test maintainability also affects the development time.
- **Test execution time**: Increasing test suite volume will lead to an increasing amount of time needed for executing the tests.
- **Staff availability**: The staff needed to create, maintain and execute the test suite must be available for each deployment. It may be difficult or impossible to ensure this over the course of the project, especially during holidays, weekends, or if deployments occur at off-hours.

One strategy to address these challenges is to minimize the test suite by selecting, preparing, and executing only a subset of tests by prioritizing the tests and/or using risk analysis. This strategy has the disadvantage to increase the risks because it implies a reduced number of executed tests.

Automation of as many tests as possible should increase the frequency and/or the speed of deployments. Another strategy to keep up with (or even increase) the frequency and/or the speed of deployment is to automate as many tests as possible.

Test automation is an operative instrument to keep up with or increase the speed of deployment in any project and is the premise for continuous deployment. To find the right amount of test automation that can keep up with the speed of deployment, the following benefits and limitations should be analyzed and balanced:

Benefits

- Test automation can guarantee a defined and repeatable level of test coverage for each deployment cycle.
- Test automation can decrease test execution time and help to increase the speed of deployment.
- Test automation can diminish limits for achieving higher deployment frequency.
- Continuous Deployment supports shortening of time to market and user feedback loops.

- Test automation can occur more frequently, which allows all tests to be executed with every build, offering a stable mainline in CI, where software is merged to mainline as often as possible.

Limitations

- Test automation requires test development and maintenance efforts, which themselves may lengthen the duration of development and thus decrease deployment frequency.
- Tests at the system test level and especially load or performance tests (and possibly other non-functional tests) might need a long time for execution even if automated.
- Automated tests can fail due to many factors. A passed automated test case may not be reliable (false negative). A failed automated test case may fail due to an error not related to the products quality or due to a false positive.

The release frequency should match the product users' needs. Too many delivered versions in too short time might displease the customer more than a slower frequency. Therefore, in situations where it is technically possible to further increase deployment frequency, it may not be of additional benefit from the customer's point of view.

4 Deployment and Delivery 105 minutes

Keywords

Service virtualization, continuous testing

Learning Objectives for Deployment and Delivery

ATT-4.x (K1) Keywords

4.1 Continuous Integration, continuous testing, and continuous delivery

ATT-4.1.1 (K3) Apply continuous integration (CI) and summarize its impact on testing activities

ATT-4.1.2 (K2) Understand the role of continuous testing in continuous delivery and continuous deployment (CD)

4.2 Virtualization

ATT-4.2.1-1 (K2) Understand the concept of service virtualization and its role in Agile projects

ATT-4.2.1-2 (K2) Understand the benefits of service virtualization

4.1 Continuous Integration, continuous testing and continuous delivery

4.1.1 Continuous integration and its impact on testing

The aim of continuous integration (CI) is to provide fast feedback so if defects are introduced into the code, they are found and fixed as soon as possible. Agile testers should contribute to the design, implementation, and maintenance of an effective and efficient CI process, not only in terms of creating and maintaining automated tests that fit within the CI framework, but also in terms of prioritizing the tests, the necessary environments, configurations, and so forth.

In an ideal situation with CI, once the code is built, all automated tests are then run to verify that the software continues to behave as defined and has not been damaged by the code changes. However, there are two conflicting goals:

1. Execute the CI process frequently to obtain immediate feedback on the code
2. Verify the code as thoroughly as possible after each build

When adequate care is not taken in the design, implementation, and maintenance of the automated tests (as discussed in the previous chapter), executing all the automated tests will take too long for the CI process to be completed multiple times per day. Even with careful test automation, it can be the case that complete execution of all automated tests across all test levels would slow down the CI process excessively. Different organizations have different priorities and different projects need different solutions to find the right balance between the goals mentioned above. For example, if a system is stable and changes less frequently, then fewer CI cycles may be needed. If the system is being updated constantly, then more CI cycles are most likely needed.

There are solutions that support both goals, which are complementary and can be used in parallel.

The first is to prioritize testing so that the basic and most important tests are always executed by using a risk-based testing approach.

The second solution is to enable different test configurations in the CI process to be used for different types of CI cycles. For the daily build and test process, only the basic tests selected based on upfront prioritization are executed. For the nightly CI process, a larger number or possibly all of the functional tests which do not require the pre-production environment are executed, Before release, more thorough functional and non-functional testing is done in a pre-production environment with real user inputs, including integration tests with databases, different systems and/or platforms.

The third solution is to speed up test execution by decreasing the amount of User Interface (UI) testing. Usually, there are no time issues completing the execution of unit/integration tests, which generally run very quickly. A situation might arise where so many unit tests exist that they cannot be completely executed during the CI process. The real issues with time generally are related to the use of end-to-end UI test cases as part of the CI process. The solution is to increase the amount of API, command line, data-layer, service-layer, and other non-UI business logic testing and decrease UI testing, pushing the automation effort down on the test automation pyramid. This ensures more maintainable tests but also reduces the test execution time.

The fourth solution can be used when test executions are very frequent in a CI system and it is impossible to run all the test cases. Based on the code modifications and knowledge of the execution trace of the existing test cases, a developer or tester can select and execute only those test cases affected by the changes (i.e., the use of impact analysis to select tests). Since only a small fraction of the whole code base is modified during a short cycle, relatively few test cases have to be executed. However, significant regression defects could be missed.

A fifth solution is to split the test suite into equally sized chunks and run them in parallel on multiple environments (agents, build farm, cloud). This is very commonly applied in companies using CI, because they already need a lot of build server capacity.

Continuous integration works on a variety of technology platforms. As development and deployment tools have moved toward the cloud, so have CI products. While most CI products are still designed to download and run in local environments, the cloud has created a new breed of products that provide CI services on hosted platforms that teams can begin to use quickly. Now teams can avoid the superfluous costs and time of creating a new environment, downloading, installing and configuring the CI software. By moving to the cloud, the team can configure and start working. When practical, the cloud is a flexible way to speed up test build and test execution when necessary.

Current CI tools support not only continuous integration, but continuous delivery and continuous deployment as well. Running automated tests in an environment that doesn't completely replicate production can lead to false negatives, but cloning the production environment can be cost excessive. Solutions include using cloud testing environments that replicate production environments on an as-needed basis, or creating a test environment that is a scaled-down but realistic version of production.

Good CI systems need to be able to automatically deploy on more complex environments and different platforms. The testers' task is to plan which test cases to include, prioritize which have to be executed on some or all platforms in order to maintain good coverage, and to design test cases to efficiently validate the software in a production-like environment.

4.1.2 The role of continuous testing in continuous delivery and deployment (CD)

Continuous testing is an approach that involves a process of testing early, testing often, test everywhere, and automate to obtain feedback on the business risks associated with a software release candidate as rapidly as possible. In continuous testing, a modification made to the system triggers the required tests (i.e., those tests that cover the change) to be executed automatically, giving the developer prompt feedback. Continuous testing can be applied in different situations (e.g., in the IDE, in CI, in continuous delivery and continuous deployment, etc.), however the concept is the same, i.e., to automatically execute tests as early as possible.

Continuous delivery requires continuous integration. Continuous delivery extends continuous integration by deploying all code changes to a testing or pre-production environment and/or a production-like environment after the build stage. This staging process makes it possible to execute functional tests by applying real user inputs and non-functional tests such as load, stress, performance and portability tests. Since every change embedded is delivered to the staging environment using complete automation, the Agile team can have confidence the system can be deployed to production with a push of a button when the definition of done is achieved.

Continuous deployment takes continuous delivery a step further with the notion that every change is automatically deployed to production. Continuous deployment aims to minimize the time elapsed between the development task of writing new code and having it be used by real users, in production. For further information, see [Farley].

4.2 Service Virtualization

Service virtualization refers to the process of creating a shareable testing service (a virtual service) that simulates the relevant behavior, data, and performance of a connected system or service. This virtual service enables development and test teams to execute their tasks even if the actual service is still under development or not available.

Testing early in the development cycle is difficult to accomplish with today's highly connected and interdependent development teams and systems. By decoupling teams and systems using service virtualization, teams can test their software earlier in the development lifecycle using more realistic use cases and loads.

While stubs, drivers and mocks are valuable in allowing for early testing, manually created stubs are usually stateless and provide simple responses to requests with fixed response times. Virtual services can support stateful transactions and maintain context of dynamic elements (session / customer IDs, dates & times etc.) as well as having variable response times that model the message flow through multiple systems.

Virtual services are created with the help of service virtualization tool, often in one of the following ways:

- By interpreting from data: XML file, historical data from server logs or simply sample spreadsheet of data
- By monitoring network traffic: exercising the SUT (System under test) will trigger relevant behavior of dependent system which is captured and modelled by the service virtualization tool
- By capturing from agents: server-side or internal logic may not be visible on the communication messages forcing relevant data and messages to be pushed from dependent server to be re-created as a virtual service

If none of the above applies, then the virtual service needs to be created within the project team, based on the appropriate communication protocol.

Benefits of service virtualization include:

- Parallel development and test activities for the service under development
- Earlier testing of services and API's
- Earlier test data setup
- Parallel compilation, continuous integration, and test automation
- Earlier discovery of defects
- Reduced over-utilization of shared resources (COTS system, mainframe, data warehouses)
- Reduced testing costs by reducing the investment in infrastructure.
- Enabling early non-functional testing of SUT
- Simplifying test environment management
- Less maintenance work for test environments (no need to maintain middleware)
- Lower risk for data management (which helps in GDPR compliance)

Note that the virtual service does not need to include all of the functionality and data of the actual service, only the parts needed for testing of the SUT.

Introducing a service virtualization could be a complex and potentially expensive endeavour. The introduction of a service virtualization tool should be treated similarly to introducing any new test tool to the team and organization. This topic is covered in the ISTQB® Foundation Syllabus and the ISTQB® Advanced Test Manager Syllabus.

5 References

[AgileFoundationExt] Certified Tester Foundation Level Extension Syllabus – Agile Tester, 2014.

[Foundation] Certified Tester Foundation Level Syllabus, 2018.

[AdvancedTestAutomationEngineer] Certified Tester Advanced Level Test Automation Engineer Syllabus, 2016

[ISO 29119-5], ISO/IEC/IEEE 29119-5 Software and systems engineering -- Software testing -- Part 5: Keyword-Driven Testing

Books

[Adzic09] Adzic, Gojko. “Bridging the Communication Gap” Neuri Ltd, 2009

[Adzic11] Adzic, Gojko. “Specification by Example” Manning, 2011.

[Beck02] Kent Beck, “Test Driven Development: By Example”, 2002

[Beck99] Beck, Kent. “Extreme Programming Explained” Addison Wesley, 1999.

[Carkenord08] Barbara A. Carkenord, “Seven Steps to Mastering Business Analysis”, J. Ross Publishing, 2008.

[Cohn 09] Mike Cohn: Succeeding with Agile: Software Development Using Scrum. Addison-Wesley Professional, 2009;

[Crispin08] Crispin, L. and Gregory, J. (2008) Agile Testing: A Practical Guide for Testers and Agile Teams. Crawfordsville : Addison-Wesley Professional

[Elfriede99] Dustin Elfriede, Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance, Addison-Wesley Professional, 1999

[Evans03] Eric Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software”, 2003

[Fewster12] Mark Fewster, Dorothy Graham, Experiences of Test Automation: Case Studies of Software Test Automation, Addison-Wesley Professional, 2012

[Fowler/Parsons 10], Martin Fowler, Rebecca Parsons, Domain-Specific Languages, Addison-Wesley Signature, Series, 2010

[Hendrickson13] Elisabeth Hendrickson, “Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing”, Pragmatic Bookshelf, 2013

[Jeffries00] Ron Jeffries, Ann Anderson, and Chet Hendrickson, “Extreme Programming Installed,” Addison-Wesley Professional, 2000

[Jorgensen13] Paul C. Jorgensen, Software Testing: A Craftsman’s Approach, Auerbach Publications; 4th edition, 2013

[Linz14] Tilo Linz, Testing in Scrum, A Guide for Software Quality Assurance in the Agile World, Rocky Nook, 2014

[Meszaros07] Gerard Meszaros, “xUnit Test Patterns: Refactoring Test Code”, Addison-Wesley, 1st Edition, Apress, 2007

[Michelsen12] John Michelsen and Jason English, “Service Virtualization: Reality is Overrated”, Apress, 1st Edition, 2012.

[Osherove09] Roy Osherove, “The Art of Unit Testing”, 2009

[Paskal15] Greg Paskal, “Test Automation in the Real World: Practical Lessons for Automated Testing”, MissionWares, 2015

[Smart15]; Smart, John Ferguson; "BDD in action", Manning, 2015

[Wein89] Weinberg, Gerald & Gause, Donald. “Exploring Requirements: Quality Before Design” Dorset House, 1989.

[Whittaker09] James A Whittaker, “Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design”, Addison-Wesley Professional, 2009

[Wiegers02] Karl Wiegers, “Peer Reviews in Software: A Practical Guide (Paperback)”, 2002.

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: https://www.agilealliance.org/agile101/agile-glossary/
In case of conflicting definitions, the ISTQB® Glossary is the leading source.

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.

[Cohn09] The Forgotten Layer of the Test Automation Pyramid, https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

[CyclomaticComplexity] https://en.wikipedia.org/wiki/Cyclomatic_complexity

[Farley] http://www.davefarley.net/?cat=5

[DSL] https://en.wikipedia.org/wiki/Domain-specific_language

[Fowler07] https://martinfowler.com/articles/mocksArentStubs.html

[Fowler04] Fowler, Martin. https://martinfowler.com/bliki/SpecificationByExample.html

[Fowler03] Martin Fowler, https://martinfowler.com/bliki/TechnicalDebt.html

[Gherkin] https://docs.cucumber.io/gherkin/

[INVEST] Bill Wake, “INVEST in Good Stories, and SMART Tasks”, http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

[IQBBA] International Qualifications Board for Business Analysts, http://www.iqbba.org/

[IREB] International Requirements Engineering Board, https://www.ireb.org/en

[IIBA] International Institute of Business Analysts (IIBA), https://www.iiba.org/

[Marick01] Marick, Brian, http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1

[Marick03] Marick, Brian. http://www.exampler.com/old-blog/2003/08/22.1.html

[North06] Dan North, “Introducing BDD”, blog post, 2006

[TESTDOUBLES] Wojciech Bulaty, Bill Wake, “Stubbing, Mocking and Service Virtualization Differences for Test and Development Teams”, https://www.infoq.com/articles/stubbing-mocking-service-virtualization-differences

[ToolList] Test tool review, information platform on the international market of software testing tools, www.testtoolreview.de/en/

[xUnit] https://en.wikipedia.org/wiki/XUnit, https://en.wikipedia.org/wiki/Unit_testing

6 Appendices

6.1 Glossary of Agile Technical Tester specific Terms

Glossary Term	Definition
Persona	A fictional character representing a certain type of users and how they will interact with the system.
Storyboard	A visual representation of the system in which user stories are represented in context for the purpose of understanding the business processes.
Story Mapping	A technique ordering user stories on two dimensions, the horizontal axis representing their execution order, and the vertical axis representing the sophistication of the implemented product.