Certified Tester
Advanced Level
Agile Tester (CTAL-AT)
Syllabus

v2.0

International Software Testing Qualifications Board

Copyright Notice

Revision History

Version	Date	Remarks
CT-AT v1.0 (2014)	2014-05-31	First version (superseded)
CTAL-AT v2.0 (2026)	2026-04-17	Revamp & repositioning

Table of Contents

Acknowledgments

This document was formally released by the General Assembly of the ISTQB® on 17 April 2026.

It was produced by a team from the International Software Testing Qualifications Board: Wim Decoutere, Michaël Pilaeten, Adam Roman, Murian Song, Nele Van Asch.

The team thanks Dr. Stuart Reid and Mark Rutz for technical review and the review team and the Member Boards for their suggestions and input.

The following persons participated in the reviewing, commenting, and balloting of this syllabus: Mariam Abdellatif, Hatim Abuazab, Bíró Ádám, Tom Adams, Gergely Agnecz, Laura Albert, Tobias Anderson, Chris Van Bael, André Baumann, Jürgen Beniermann, Menno van den Berg, Ruud van Berkum, Armin Born, Earl Burba, Simeone Chiumarulo, Marco Ciarlitto, Alessandro Collino, Walter Eder, Michael Fischlein, Valeriya Gagarina, Attila Gyuri, Ferenc Hamori, Andrea Horvath, Sagar Joshi, Mattijs Kemmink, Jędrzej Kwapiński, Thomas Letzkus, Ine Lutterman, Niranjan Maharajh, Judy McKay, Imre Meszaros, Gary Mogyorodi, Thaer Mustafa, Frank Neirynck, Andreas Neumeister, Melanie Neumueller, Ingvar Nordström, Tauhida Parveen, Tal Pe'er, Christoph Peters, Lukas Piska, Horst Pohlmann, Andrew Pollner, Nishan Portoyan, Meile Posthuma, Randall Rice, Jean François Riverin, Nicola De Rosa, Mark Rutz, Sarno Salvatore, Márton Siska, Klaus Skafte, Péter Sótér, Michael Stahl, Lucjan Stapp, Ioannis Symeonidis, Reifa Tangon, Richard Taylor, Benjamin Timmermans, Giancarlo Tomasig, Stephanie Ulrich, Csilla Varga, Linda Vreeswijk, and Jana Zientková.

International Software Testing Qualifications Board Working Group Foundation Level at the time the Foundation Level Agile Extension Syllabus v1.0 (2014) was completed: 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 of Foundation Level Agile Extension Syllabus v1.0 (2014): 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 of Foundation Level Agile Extension Syllabus v1.0 (2014): 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 Level Agile Extension Syllabus v1.0 (2014): 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.

0 Introduction

0.1 Purpose of this Syllabus

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

1. 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 modify the references to adapt to their local publications.
2. To certification bodies, to derive examination questions in their local language, adapted to the learning objectives for this syllabus.
3. To training providers, to produce training materials and determine appropriate teaching methods.
4. To certification candidates, to prepare for the certification exam (either as part of a training course or independently).
5. To the international software and systems engineering community, to advance the software and systems testing profession, and as a basis for books and articles.

0.2 The Certified Tester Agile Tester in Software Testing

The ISTQB® Certified Tester Agile Tester (CTAL-AT) qualification is intended for anyone involved in software testing on Agile projects. This includes people in roles such as testers, test analysts, test engineers, test consultants, test managers, user acceptance testers, and software developers. This qualification is also appropriate for anyone seeking a better understanding of Agile software testing, including product owners, Scrum masters, quality managers, software development managers, business analysts, IT directors, and management consultants.

0.3 Career Path for Testers

The ISTQB® scheme provides support for testing professionals at all stages of their careers, offering both breadth and depth of knowledge. Individuals who achieve the ISTQB® CTAL - Agile Tester certification may also be interested in other specialist syllabi, with a special focus on Agile Test Leadership at Scale.

Please visit *www.istqb.org* for the latest information on ISTQB´s Certified Tester Scheme.

0.4 Business Outcomes

This section lists the Business Outcomes expected of a candidate who has achieved the CTAL - Agile Tester certification.

A Certified Agile Tester can:

Code	Description
CTAL-AT-BO1	Collaborate in cross-functional teams, being familiar with principles and basic practices of Agile software development
CTAL-AT-BO2	Adapt existing testing experience and knowledge to Agile values and principles
CTAL-AT-BO3	Support the Agile team in test planning
CTAL-AT-BO4	Apply relevant Agile software development approaches and test techniques to ensure tests that provide adequate coverage
CTAL-AT-BO5	Assist business stakeholders in defining understandable and testable user stories, scenarios, requirements, and acceptance criteria, as appropriate
CTAL-AT-BO6	Create and implement various Agile software development test approaches
CTAL-AT-BO7	Support and contribute to test automation in an Agile software development project
CTAL-AT-BO8	Work with - and share information with - other team members using effective communication styles and channels

0.5 Learning Objectives and Cognitive Level of Knowledge

Learning objectives support the business outcomes and are used to create the CTAL - Agile Tester certification exams.

In general, all contents of this syllabus are examinable, except for the Introductions and Appendices. The exam questions will assess knowledge of keywords at the K2 level (see below) or learning objectives at the respective level of knowledge.

The specific learning objectives and their levels of knowledge are shown at the beginning of each chapter, and classified as follows:

- K1: Remember (K1 Learning Objectives are not present in this syllabus)
- K2: Understand
- K3: Apply
- K4: Analyze

Further details and examples of learning objectives are given in *Section 11*.

For all terms listed as keywords just below chapter headings, the correct name and definition from the ISTQB® glossary shall be remembered (K1), and the definition from the ISTQB® Glossary shall be understood (K2), even if not explicitly mentioned in the learning objective.

0.6 The CTAL - Agile Tester Certification Exam

The CTAL - Agile Tester certification 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 and books are included as references, but their content is not examinable, beyond what is summarized in the syllabus itself from such standards and books.

Refer to "Exam Structures and Rules" for CTAL - Agile Tester for further details.

The entry criteria for taking the CTAL - Agile Tester exam are:

- That candidates are interested in software testing, especially in Agile environments.
- That candidates have obtained the ISTQB® Foundation Level certificate.

However, it is strongly recommended that candidates also:

- Have at least a minimal background in either Agile software development or software testing, such as six months of experience as a system or user acceptance tester or as an Agile team member
- Take a course that has been accredited to ISTQB® standards (by one of the ISTQB-recognized member boards).

0.7 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.

The accreditation guidelines for this syllabus follow the general Accreditation Guidelines published by the Processes Management and Compliance Working Group.

0.8 Handling of Standards

International standardization organizations such as IEEE and ISO have issued standards related to quality characteristics and software testing. Such standards are referenced in this syllabus. The purpose of these references is to provide a framework (as in the references to ISO 25010 regarding quality characteristics) or to provide a source of additional information if desired by the reader. Please note that ISTQB® syllabi use the standards documents as references. Standards documents are not intended for examination. Refer to *Section 7* for more information on Standards.

0.9 Level of Detail

The level of detail in this syllabus allows internationally consistent courses and exams. In order to achieve this goal, the syllabus consists of:

- General instructional objectives describing the intention of the CTAL - Agile Tester syllabus
- A list of terms that students must be able to recall
- Learning objectives for each knowledge area, describing the cognitive learning outcome to be achieved
- A description of the key concepts, including references to sources such as accepted literature or standards

The syllabus content does not describe the entire knowledge area of software testing; it reflects the level of detail to be covered in CTAL - Agile Tester training courses. It focuses on test approaches and test techniques that can be applied to all software projects following Agile software development.

The syllabus uses the terminology (i.e. the names and meanings) of the software testing and quality assurance terms according to the *ISTQB® Glossary*.

For the terminology in related disciplines, please refer to the respective glossaries: IREB-CPRE (*IREB-CPRE for Requirements Engineering*, [n.d.]), IEEE Pascal (*IEEE/Pascal for software engineering*, [n.d.]), and Agile Alliance for agile terminology (*Agile Alliance*, [n.d.]).

0.10 How this Syllabus is Organized

There are six chapters with examinable content. The top-level heading for each chapter specifies the time for the chapter. For accredited training courses, the syllabus requires a minimum of 13 hours of instruction divided over at least two days. The minimum training time is distributed across the six chapters as follows:

- Chapter 1: 60 minutes, Test Strategy and Test Approach Challenges
- Chapter 2: 60 minutes, People and Teams
- Chapter 3: 210 minutes, Test Management and Test Process Improvement
- Chapter 4: 135 minutes, Shift Left
- Chapter 5: 285 minutes, Agile Test Approaches and Test Techniques
- Chapter 6: 30 minutes, Test Automation and Test Tools

1 Test Strategy and Test Approach Challenges - 60 minutes

Keywords

acceptance test-driven development, bug bash, edge case, end-to-end testing, formal testing, holistic testing, regression testing, test type, test-driven development

Domain Specific Keywords

feature toggle, hardening iteration, iteration, user journey

Learning Objectives:

1.1 Test Types

CTAL-AT-1.1.1 (K2) Compare test types to be performed during and after an iteration

1.2 End-to-End Testing

CTAL-AT-1.2.1 (K2) Explain when end-to-end testing should be performed

1.3 Formal Testing and Holistic Testing

CTAL-AT-1.3.1 (K2) Compare the benefits and drawbacks of formal testing and holistic testing

1.4 Regression Test Approaches

CTAL-AT-1.4.1 (K2) Differentiate among regression test approaches

Introduction to Agile

Whilst basic Agile foundations are already described in the Foundation Level syllabus, this extensive introduction fills in essential concepts not covered there but critical to understanding the practices, roles, and mindset addressed throughout this syllabus.

Agile software development has become one of the most widely adopted approaches in software development today. Its core objectives are to adapt quickly to changes and to consistently deliver value to customers. To achieve these goals, Agile software development takes a distinct approach from sequential development models, leading to the emergence of Agile testing. Agile testing goes beyond simply testing software; it is a collaborative, integrated method that embeds quality throughout the development process and highlights the whole team's responsibility for quality.

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 (Beck; Beedle, et al., 2001). The Agile Manifesto defines the following values:

- Individuals and interactions over processes and tools. Agile software 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 over comprehensive documentation. 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, even with reduced functionality, is delivered much earlier in the development lifecycle, Agile software development can offer a significant time-to-market advantage. Agile software development is especially useful in rapidly changing business environments where the problems or solutions are unclear or where the business wishes to innovate in new problem domains.
- Customer collaboration over contract negotiation. Customers often find it difficult to specify the system they require. Direct collaboration with the customer improves the likelihood of understanding exactly what the customer requires. While having contracts with customers may be important, working in regular, close collaboration with them is likely to bring greater success to the project.
- Responding to change over following a plan. Change is inevitable in software projects. The environment in which the business operates, including legislation, competitor activity, technological advances, and other factors, can significantly impact the project and its objectives. These factors must be considered in the development process. As such, having flexibility in work practices to embrace change is more important than adhering rigidly to a plan.

Agile principles. The twelve principles of the Agile Manifesto capture its core values, emphasizing the delivery of valuable software early and often, embracing changing requirements, frequent releases, and daily collaboration between business stakeholders and developers. They prioritize motivated, supported individuals, face-to-face communication, working software as the key measure of progress, and sustainable development at a steady pace. They stress technical excellence, simplicity, self-organizing teams, and regular reflection to continuously improve effectiveness and adaptability.

Agile software development practices. Agile teams use various prescriptive practices to put these values and principles into action. Common practices across most Agile organizations include collaborative user story creation, retrospectives, continuous integration, and planning for each iteration as well as for the overall release.

Agile methodologies. Several Agile software development methodologies are in use by organizations. The most representative are:

- Scrum – a lightweight framework that organizes work into fixed-length iterations called sprints, emphasizing defined roles, ceremonies, and a prioritized product backlog (Sutherland et al., 2020)
- Extreme Programming (XP) – a methodology focused on engineering practices like test-driven development, continuous integration, and close collaboration to improve software quality and responsiveness (Beck; Andres, 2004)
- Kanban – a visual workflow management method that limits work in progress, emphasizes continuous delivery, and uses boards to track and optimize flow (D. Anderson et al., 2010).

Defining Agile Testing. Janet Gregory and Lisa Crispin, prominent voices in the Agile community, define Agile testing as "collaborative testing practices that occur from inception to delivery, supporting frequent delivery of quality products that add business value for our customers. Testing activities focus on defect prevention rather than defect detection, and work to strengthen and support the idea of whole team responsibility for quality" (Gregory; Crispin, 2019).

This definition clearly articulates the basic principles of Agile testing. It is not confined to the final stages of a project; instead, testing is an ongoing activity from the very beginning. Furthermore, the testing role is a shared one, and shifts from merely finding defects to proactively preventing them. This embodies the whole team approach, where everyone involved in the development process shares ownership of quality.

Agile testing encompasses a wide range of activities, including - but not limited to - guiding development with concrete examples, challenging ideas and assumptions with insightful questions, automating tests to improve efficiency, performing exploratory testing to uncover defects, and evaluating quality characteristics such as performance efficiency, reliability, and security. These activities illustrate that in Agile software development, testing roles are not just limited to test execution; they are quality moderators (facilitators) and collaborators.

The Agile Testing Manifesto. Karen Greaves and Samantha Laing's Agile Testing Manifesto distills the values and principles that guide Agile testing into five concise statements. This manifesto highlights the fundamental differences from testing in sequential software development and outlines the mindset crucial for testing roles in an Agile setting (Greaves et al., 2019):

- Testing throughout over testing at the end. In sequential software development, testing often begins only after development is complete. Agile, however, emphasizes continuous testing throughout the development lifecycle, from requirements analysis and design to implementation and deployment. This approach helps to identify and fix defects early, significantly reducing costs and effort. By embracing the shift-left approach, the whole team builds quality in from the start, making testing a continuous learning process that strengthens confidence in the software under test.
- Preventing defects over finding defects. The ultimate goal of Agile testing is not just to discover defects but to prevent them from occurring in the first place. To achieve this, testing roles actively participate in early discussions, such as clarifying requirements and reviewing designs, thereby proactively identifying and mitigating potential problems.
- Testing understanding over checking functionality. Beyond simply verifying that software performs its stated functions, it is vital to test the underlying understanding of the software's purpose and user needs. This involves evaluating the system from the user's perspective, often through user stories and scenarios, to ensure it aligns with business value.

- Building the best system over breaking the system. A testing role's primary function is not to exploit system weaknesses or intentionally break it. Instead, testing roles collaborate with developers to help build the highest quality system. This includes providing constructive feedback, proposing testable designs, and sharing ideas for continuous quality improvement.
- Team responsibility for quality over tester responsibility. Quality is not the sole responsibility of an individual or a dedicated test team. Instead, it is a shared commitment among everyone involved in the development process, including developers, business representatives, testers, and other stakeholders. Agile testing fosters this culture of shared ownership, encouraging every team member to actively participate in quality-related activities.

The Importance and Impact of Agile Testing. These core principles of Agile testing are pivotal in ensuring software quality within Agile software development, which prioritizes responsiveness and rapid feedback. In sequential software development, testing often becomes a project bottleneck or a final "quality gate" just before release. However, in Agile, testing is seamlessly integrated as a fundamental and indispensable part of the development workflow. This integration empowers teams to consistently deliver high-quality software.

Agile testing also significantly strengthens collaboration and communication within the team. Testing roles work hand-in-hand with developers and business stakeholders to clarify requirements, identify potential defects early, and encourage everyone to share a common objective for quality. This collaborative environment not only contributes to defect prevention but also plays a vital role in boosting team productivity and overall efficiency. The testing role is not necessarily linked to a specific position within the organization, it can be performed by anyone at any time.

In essence, Agile testing is more than just a collection of test techniques; it represents a profound shift in software development culture and mindset. It is essential to focus on customer value, adapt flexibly to change, and continuously deliver high-quality products under the collective responsibility of the entire team.

1.1 Test Types

Test types are selected based on the specific test objectives and context of the iteration, including user story acceptance criteria, quality risks, and feedback needs. All test types can be used at any stage of the project, but their use may differ during and after the iteration. Any decision made should be reflected in the Definition of Done and the test approach.

During the iteration, functional testing verifies that the implemented features meet the related functional requirements and acceptance criteria. After the iteration, expanded functional testing can include functional testing of the integration of different features, including those developed by other teams.

During the iteration, non-functional testing is typically limited to testing basic performance efficiency, usability, or security on new features. After the iteration, full non-functional testing can be performed on integrated or pre-production (staging) environments. Further exploratory testing and usability testing with a broader group may also be performed post-iteration to validate the product's fitness for purpose.

During the iteration, white-box testing is used before or during coding, for example as part of test-driven development (TDD). This supports early defect detection and helps maintain a stable codebase. It is rarely performed as a separate post-iteration-activity - however, code coverage can be measured during black-box testing to provide additional insight into the effectiveness and completeness of the black-box tests.

During the iteration, black-box testing focuses on verifying new or modified features against specifications defined typically as user stories' acceptance criteria. This is consistent with functional testing using acceptance test-driven development (ATDD), helping to ensure that the delivered functionality meets stakeholder expectations. After the iteration, broader, system level black-box testing may be performed, including regression testing and end-to-end testing.

During the iteration, exploratory testing plays a critical role. This specific approach to functional testing complements both white-box testing and black-box testing by uncovering unknown risks and emergent behavior through structured, time-boxed sessions.

For information regarding the link between test types and testing quadrants, see *Section 3.1.2*.

1.2 End-to-end Testing

In Agile software development, end-to-end testing is performed when it adds value by verifying integration across systems and user workflows. This includes testing the entire application and, if relevant, multiple systems or services. However, as a mitigation measure, it should be balanced against the risk involved in the E2E chain, with the particular user story, because it is costly to maintain, time consuming to run, and provides limited diagnostic feedback. End-to-end testing is typically performed closer to a release, during hardening iterations, or when system-wide integration points are newly introduced or at high risk of failure. While hardening iterations are used in some organizations, they may indicate insufficient attention to quality during regular iterations. End-to-end testing is especially useful in distributed systems, complex user journeys, or scenarios with third-party dependencies that cannot be reliably tested in isolation. Its purpose is to validate critical user flows in a pre-production environment, not to detect low-level defects.

Several factors influence the decision to include end-to-end tests:

- Risk. The risk impact and risk likelihood of a system level failure justify targeted end-to-end testing, even if confidence is built through lower-level tests.
- Feedback time. Relying heavily on end-to-end tests slows down delivery. Teams often favor faster tests at lower test levels, and select only a few key flows for end-to-end coverage.
- Maturity of the Continuous Integration/Continuous Delivery (CI/CD) pipeline. When teams have good test automation, observability, and monitoring in production, confidence can shift toward production feedback, feature toggles, or A/B testing as alternatives to, or to complement, extensive end-to-end testing.

From a test strategy perspective, the test pyramid remains valid: broad, fast feedback at the base (component testing), fewer service-level tests, and minimal end-to-end tests at the top. Deviating from this structure may increase costs and risks without proportionate value.

In microservices and distributed architectures, contract testing offers an alternative to extensive end-to-end testing. Contract testing verifies that services communicate correctly by validating agreed-upon interfaces (contracts) between service consumers and providers. Unlike end-to-end tests that require all services to be running, contract tests can be executed independently, providing faster feedback and easier failure diagnosis. Consumer-driven contract testing, in which consumers define their expectations and providers verify that they meet them, is particularly effective in Agile environments where services evolve independently. Contract testing complements, rather than replaces, a small set of end-to-end tests that validate critical user journeys.

Integration testing should preferably be performed at the interface level unless system level behaviors are explicitly impacted. End-to-end testing should align with stakeholder-defined quality objectives and not be used indiscriminately (*ISO 29119-6 Software and systems engineering – Software testing*, 2021).

Scripted end-to-end tests may miss unexpected behaviors that arise from real usage. Holistic testing (see *Section 1.3*) and exploratory testing (see *Section 5.1.5*) complement these tests by allowing testers to examine quality from multiple perspectives and use experience-based testing to help uncover unexpected behaviors, usability defects, and edge cases that predefined scripts may miss.

1.3 Formal Testing and Holistic Testing

Formal testing emphasizes precision and formalization. It involves deriving test cases from explicitly defined requirements. It is typically used in contexts that require strict compliance or risk control (e.g., regression testing, when regression is considered a risk).

Benefits of formal testing include:

- Traceability: traceability is ensured between requirements and test cases.
- Alignment: test cases are aligned with coverage metrics.
- Verification: formal testing assists in verification, testing whether the system has been built right.
- Conformance: test cases can be inspected to ensure conformance with standards and guidelines.
- Repeatability: since test cases are formally documented, repeatability is ensured.
- Test Automation: formalization and precision smoothen the translation to test automation.

Challenges of formal testing include:

- Excessive rigidity: It tends to restrict exploration and responsiveness to change.
- Over-reliance on documented requirements: The test basis is typically incomplete, inconsistent, or incorrect. When test cases are created solely from poor or flawed explicit requirements, this results in incomplete coverage, misaligned test cases, and an increased likelihood of undetected defects.
- Missing the unexpected: Focusing only on expected results, it may overlook emergent risks and failures.
- Delayed feedback: It is often executed late in the test cycle, which defers discovery of critical problems.
- Low cognitive engagement: It encourages mechanical test execution instead of thoughtful analysis.

Holistic testing (Gregory, 2021) extends beyond planned test activities to encompass a system-wide view of quality. It incorporates continuous testing throughout the software development lifecycle and includes a wide range of perspectives: customer, team, and organizational. It focuses on collaboration, context awareness, systems thinking (see (ISTQB-ATLAS, v1.0), section 3.2.1), and exploration.

Benefits of holistic testing include:

- Addressing different perspectives: combining process, product, and user-based points of view simultaneously.
- First Time Right (FTR): ensuring that quality is built in, not just tested after the fact.

- Continuous Testing: combines verification (building the system right) and validation (building the right system).

Challenges of holistic testing include:

- Lack of focus: Without a disciplined structure, teams may test widely but miss critical risks or regressions.
- Difficult to measure: Evaluation of quality is context-sensitive and lacks fixed metrics, challenging stakeholders' trust.
- Requires culture change: It relies on strong collaboration and systems thinking, and without supportive leadership and team maturity, adoption may fail.
- Ambiguity in responsibility: "Whole team" ownership can dilute accountability if not carefully managed.
- Heavily context-dependent: It is not universally applicable, and it may not satisfy regulatory or contractual constraints that require more formal verification.
- Test Automation issues: the lack of formality endangers proper test automation.

To maximize the benefits of these two approaches, they can be used in a complementary way. Formal testing should focus on critical components, high-risk areas, or regulatory and legal requirements, and be automated where possible to minimize repetitive effort. Holistic testing should be used for general quality control, early feedback, and exploring complex scenarios, prioritizing areas with the highest user impact.

1.4 Regression Test Approaches

In Agile software development, regression testing must be organized to support continuous delivery and rapid change. Different regression test approaches can be used individually or in combination, depending on the context, team capabilities, risk profile, and test automation maturity:

- Incremental regression testing: Relies on an automated test suite that is maintained and run continuously. Instead of re-running a full regression test suite only at the end of an iteration or release, a subset of regression tests is triggered automatically with each code commit or integration event, selected based on risk impact, risk likelihood, location, and dependencies of the code change. This allows early detection of regression defects and supports continuous feedback loops.
- Risk-based regression testing: Uses frequent risk assessment workshops or team discussions to identify areas where changes are likely to cause defects, and guides both automated and manual regression efforts. This method helps decide where test automation is unnecessary and where human investigation adds more value.
- DevOps-oriented regression testing: Weaves regression testing into the CI/CD pipeline, with smoke tests acting as quality gates and high-priority regression tests executed automatically post-deployment in pre-production or even production environments; feature toggles and canary releases allow selective exposure, while observability and monitoring can, in some cases, replace traditional regression testing.
- Exploratory regression testing: Is applied continuously or at regular intervals, supplementing test automation by using testers' insights to explore interactions across features and test for unexpected defects.

- Collaborative regression testing: Includes test activities such as bug bashes or whole team walkthroughs of critical flows, increasing team awareness of the product's current state and risk areas. It can be time-boxed and used when test automation is not yet mature, or as a learning and risk discovery exercise.

Traceability between user stories, requirements, and acceptance criteria and their corresponding test cases plays a key role in regression testing. It enables teams to quickly identify which tests must be updated or rerun when changes are made, thereby reducing unnecessary effort and focusing regression testing on impacted functionality. It also highlights coverage gaps, helping teams balance automated testing, exploratory testing, and collaborative regression testing. Lightweight traceability (e.g., tagging tests to user stories in a backlog tool) enables teams to maintain confidence in frequent releases without slowing delivery.

2 People and Teams - 60 minutes

Keywords

acceptance test-driven development, behavior-driven development, tissue tester

Learning Objectives:

2.1 Whole Team Approach

CTAL-AT-2.1.1 (K2) Compare generalization and specialization within a team

CTAL-AT-2.1.2 (K2) Give examples of how to motivate business representatives to perform test activities

CTAL-AT-2.1.3 (K2) Summarize how the whole team approach can assist the development team

2.2 Tissue Testers

CTAL-AT-2.2.1 (K2) Explain how and when to use tissue testers

Introduction

In Agile software development projects, people and tools form the foundation for delivering quality at speed. The whole team approach emphasizes that quality is a shared responsibility among developers, testers, and business representatives who collaborate closely. Typically, teams are composed of multi-disciplinary or cross-functional members. While specialists bring deep expertise, generalists adapt across roles, enabling greater flexibility and shared workload. Activating and motivating business representatives to contribute to testing through early involvement in defining acceptance criteria, reviewing test results, and participating in exploratory testing, enhances product quality, and alignment with business goals. This collaboration also benefits developers, because sharing testing insights reduces defects and accelerates feedback. Agile teams may also engage tissue testers – external testers who provide a "fresh eyes" perspective to uncover defects early. Knowing how and when to use such testers helps provide balanced, unbiased feedback without slowing delivery.

2.1 Whole Team Approach

2.1.1 Generalization and Specialization within a Team

The essence of the whole team approach lies in testers, developers, and business representatives working together as an Agile team in every step of the development process. Generalization and specialization of roles within an Agile team directly impact how test activities are distributed, owned, and improved. Agile promotes a whole team approach to quality, where everyone owns quality regardless of title or position.

Agile testers are typically generalizing T-shaped specialists. They maintain deep expertise in testing and broad knowledge across multiple disciplines like coding, business analysis, and DevOps. This supports cross-functional collaboration and enables flexibility in team composition.

Generalization allows multiple team members to perform a certain activity. This supports a sustainable pace, reduces bottlenecks, and enhances team resilience. It also encourages shared ownership of quality, which is essential in continuous delivery environments. Role specialization still exists, particularly for test tasks that require specific expertise, such as test techniques, exploratory testing, test automation, or usability testing. However, Agile teams strive to minimize silos, where individuals work within narrow role boundaries, by encouraging knowledge sharing and pairing.

Mob programming and mob testing (see *Section 5.2.1*) blur the roles within the team. These approaches bring the whole team together, including business representatives, developers, and testers, to work on a single task simultaneously, rather than dividing the work strictly by specialized roles.

Testers often take the lead in helping others apply quality-focused practices such as BDD, exploratory testing, and test automation. At the same time, they may learn from developers about implementation details to better assess risks and design tests. In this way, testers both preserve their specialization (deep knowledge of testing) while also developing broader skills that contribute to generalization within the team.

The balance between specialization and generalization should be context-driven. Complex domains or regulated environments may require more specialized roles, while lean startups may benefit from broadly skilled generalists. Regardless of structure, clear communication and mutual respect among team members are essential. Testers should contribute to conversations from the beginning of the project to ensure that quality is built in rather than only assessed later.

In Agile environments with multiple teams (e.g., SAFe, LeSS), the whole team approach extends beyond

individual teams. Cross-team coordination on quality standards, integration testing, and defect ownership at team boundaries becomes essential. Testers often participate in communities of practice to share knowledge across teams, see (ISTQB-ATLAS, v1.0).

2.1.2 Motivating Business Representatives to Participate in Testing

Quality is a shared responsibility across the whole team. Business representatives define product quality, but should also participate in test activities. Motivating them to test, begins by involving business representatives early in the test process, especially in defining acceptance criteria and formulating business-facing tests. Their engagement in these early discussions ensures that what gets delivered aligns with customer value and stakeholder intent.

Testing should not be presented as a separate phase, but rather as a collaborative, continuous activity. Practices such as acceptance test-driven development, behavior-driven development and specification by example help anchor the business representative's role in clarifying behavior through concrete, testable examples. These practices make testing visible and relevant to business concerns, inviting business representatives into meaningful dialogue with team members and other stakeholders.

Embedding testability into user stories during backlog refinement ensures that acceptance criteria are concrete rather than abstract, tied to real, verifiable test results. Working closely with testers and developers in these sessions allows business representatives to see how unclear requirements lead to ambiguity and rework, and how well-formed examples lead to faster, more predictable delivery.

Visibility is a key driver of motivation. Dashboards showing coverage, defect trends, and metrics related to quality risks make it easier for business representatives to understand the current product quality and where testing adds value. Business metrics, such as customer satisfaction, support costs, or feature adoption, help business representatives view testing as part of product validation, not just verification. These insights help link testing directly to product decisions and release readiness.

Shared ownership is reinforced in daily collaboration. During iteration planning and refinement, business representatives should actively help shape what "done" means for a particular user story, including coverage and acceptance criteria. Their presence during test activities like bug bashes or usability testing ensures that feedback loops remain short and grounded in business value, while also increasing their motivation as they witness the fruits of their labor.

Finally, recognizing the business representative's contribution to testing during retrospectives and demos reinforces their role in quality and keeps them engaged. With clear expectations, access to relevant information, and continuous opportunities to influence outcomes, business representatives are more likely to see testing as a core part of their responsibilities and a powerful lever for delivering successful products.

2.1.3 Supporting the Developers

The whole team approach assists developers by embedding quality-focused collaboration throughout the software development lifecycle. Continuous interaction between developers, testers, and business representatives builds a shared understanding of quality goals and potential risks, leading to better technical and business decisions. Developers benefit from early test feedback, which reduces defect costs and supports rapid corrective action, especially when practices such as acceptance test-driven development and behavior-driven development are applied.

Testers working closely with developers help clarify acceptance criteria through examples, leading to improved specification by example and better alignment between the code and business expectations.

Testers can coach developers to write better component tests and component integration tests to achieve higher levels of code coverage. As testing becomes a continuous activity rather than a late-phase process, developers gain confidence to refactor and evolve the codebase safely. Automating tests early in the software development lifecycle enables faster detection of integration defects and facilitates implementing CI/CD pipelines.

Furthermore, the whole team mindset breaks down role silos, creating an environment in which testers contribute to code reviews from a quality and risk perspective, and developers participate in exploratory test sessions. This mutual learning environment not only elevates the team's overall testing competence but also fosters a collective responsibility for product quality. Developers also gain insights into real user behavior by participating in exploratory testing, usability test sessions, or analyzing production analytics, which helps them build more user-aligned software.

Another way the whole team approach supports developers is by involving operations, security specialists, and testers early to address deployment, monitoring, and compliance considerations throughout development. This collaboration helps developers design solutions that are easier to deploy, observe, and secure in production, reducing late-phase rework and operational problems. By embedding these perspectives from the start, developers can make architectural choices that balance functionality with quality characteristics. This will ultimately speed up delivery while increasing confidence in the system's readiness for real-world use.

2.2 Tissue Testers

Tissue testers (also known as fresh-eyes testers) are temporary testers, external to the team, who provide fast, informal feedback on a product or feature in development, typically used in volatile sectors such as the video game industry. They are most effective when used in the early phases of design or feature implementation, before teams commit to expensive development or infrastructure decisions. The idea is to briefly expose a new idea, interface, or interaction to someone unfamiliar with it in order to detect usability defects, confusing behavior, or misleading flows.

Tissue testers are typically people from outside the team or project (for example, a colleague from another team) and should be used with the understanding that their feedback may vary significantly between individuals, and that their involvement is short-term. After they have seen the product once, they are no longer truly unbiased, so their special value as "fresh eyes" diminishes, see (see *Section 4.1.4*). This metaphor underscores the practice's lightweight, transient nature. A particular tissue tester should not be used once they have seen the design under test, because their reactions will be influenced by prior knowledge. At that point, they lose the essential fresh perspective.

Tissue testers are most appropriate for exploratory testing, usability testing, or assumption-checking, where quick feedback is valuable but it is too early for rigorous validation. For example, before refining an interaction or screen flow, a developer or tester might walk a tissue tester through it to identify potential issues. It is important not to explain too much and thus introducing bias, since this will destroy their fresh-eyed view of the product.

Tissue testing should not replace structured testing. Rather, it should precede or complement it. Its primary value is in revealing early misalignments between design intent and actual user comprehension. Because they are quick to run and do not require recruiting external users, tissue tests are useful in Agile software development and DevOps contexts.

3 Test Management and Test Process Improvement - 210 minutes

Keywords

Agile software development, test control, test monitoring, test planning, test process improvement, test reporting, test strategy, testing quadrants

Domain Specific Keywords

canary release, community of practice, dark launch, Definition of Done, feature toggle, Goal-Question-Metric, metric

Learning Objectives:

3.1 Test Planning

CTAL-AT-3.1.1 (K2) Summarize how to perform test planning in Agile software development

CTAL-AT-3.1.2 (K4) Outline a project test strategy using testing quadrants

3.2 Test Monitoring and Test Control

CTAL-AT-3.2.1 (K2) Explain how to perform test monitoring and test control in Agile software development

3.3 Test Reporting

CTAL-AT-3.3.1 (K2) Compare the different types of coverage that can be used for test reporting in Agile software development

3.4 Test Process Improvement

CTAL-AT-3.4.1 (K4) Select appropriate test process improvement measures based on metrics in Agile software development

CTAL-AT-3.4.2 (K2) Explain how to perform test process improvement in Agile software development

Introduction

In Agile software development projects, test activities are integrated into short, iterative test cycles, requiring adaptive test planning that evolves with the product and team priorities. A project's test strategy is supported by the testing quadrants (Gregory; Crispin, 2019), which are used to select appropriate test types, test levels, test approaches, and test techniques to achieve balanced coverage of business-facing and technology-facing needs.

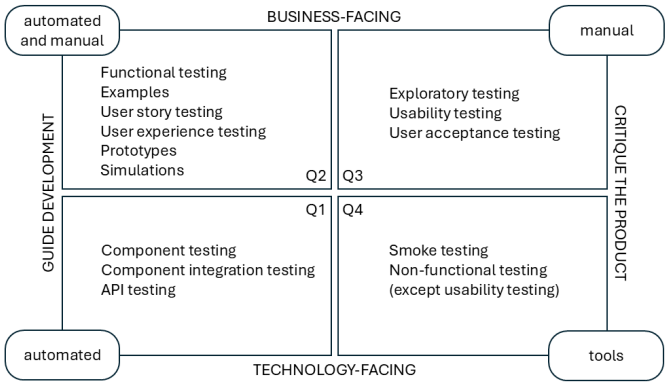

Fig. 1. Testing quadrants model

Quadrant 1 contains technology-facing tests that help guide development. They verify small pieces of code. Quadrant 2 contains business-facing tests that help guide development. They validate that the product behaves as the business expects. Quadrant 3 contains business-facing tests that critique the product. They validate how well the product works for users. Quadrant 4 contains technology-facing tests that critique the product. They verify the product's quality characteristics.

Agile principles further shape the test strategy by promoting adaptive planning, early feedback, and continuous risk-based prioritization to align testing with business value. Test monitoring and test control are ongoing, using lightweight, real-time metrics to track progress, detect risks, and adjust priorities. Test reporting delivers meaningful, context-driven coverage metrics, such as requirements, code, and risk coverage, to provide stakeholders with straightforward, unambiguous insights into quality status. Continuous test process improvement is embedded in Agile, leveraging metrics to identify bottlenecks, refine practices, and enhance efficiency. By initiating improvements early and often, teams can respond to change, sustain quality, and maximize the value of testing within each iteration.

3.1 Test Planning

3.1.1 Test Planning in Agile Software Development

In Agile, test planning typically happens at two main levels: iteration planning and release planning. The project test strategy impacts these planning activities.

Iteration Planning

During iteration planning, the team reviews the product backlog and selects user stories for the iteration. Testers can contribute by facilitating risk-storming sessions with business representatives and developers

to identify risks and prioritize test activities, define high-level test conditions, and refine acceptance criteria.

Iteration planning includes selecting appropriate test types, test levels, test approaches, and test techniques for the given iteration. Testers use their knowledge and experience to help the team in making the right choices. The test quadrants can be used to support this task.

Iteration planning must consider the test environment and the test data needs, to achieve iteration readiness. Since testing is not a separate phase in Agile and can occur at any point during the iteration, test environments and test data should be stable and available from the start of the iteration. The test plan typically includes continuous integration and continuous testing practices. Manual testing complements test automation by addressing areas that require human judgment, such as usability testing and exploratory testing.

Iteration planning includes agreeing on or revisiting the team-level Definition of Done, detailing the exit criteria for stories and other backlog items within the iteration. It also clarifies how test execution, test results, and coverage information contribute to meeting the Definition of Done for each item and for the iteration outcome.

Release Planning

During release planning, test planning operates at a broader level than in iteration planning. The team identifies high-level business objectives and evaluates the product backlog to determine the scope and sequencing of features to be delivered. From a testing perspective, this involves assessing quality risks across the release scope, defining release-level test objectives, and determining the appropriate test levels, test types, test approaches, and test techniques needed to mitigate those risks.

Testers contribute by identifying dependencies between features, constraints on test environments and data, and integration points that require specific validation. They help estimate the overall test effort using metrics-based and expert-based techniques, which support the team's ability to plan capacity across multiple iterations.

The team outlines a release-level test strategy that covers at least both functional and non-functional testing, how these will be incorporated across the iterations, and how regression testing will be managed. It ensures early planning for experience-based testing and identifies where external stakeholders, such as customers or compliance experts, need to be involved. The tester, often acting as a moderator, can guide the team through collaborative exercises (e.g., mind-mapping workshop or risk-storming sessions) to design the strategy.

Release planning includes agreement on the Definition of Done at the release level, including acceptance criteria for the integrated product. The team also defines how test results like coverage information will be reviewed and communicated to support release readiness assessments.

3.1.2 Project Test Strategy

A project test strategy is shaped by several contextual factors, each influencing how test techniques, test levels, test approaches, and test types are assigned to the testing quadrants. The product's domain complexity and business criticality form the foundation of the test strategy. In domains with legal or safety implications (e.g., finance, healthcare, aviation), there is increased emphasis on methods from Quadrants 2 and 4, where traceable, structured approaches are favored to provide auditable evidence of compliance and coverage (e.g., non-functional testing such as performance testing, security testing, compatibility testing, or reliability testing).

Organizational structure and team composition

They influence the depth and breadth of testing. A cross-functional team with embedded test specialties in UX, security, or performance enables deeper testing across all quadrants. In contrast, teams that rely on external or siloed experts may underutilize exploratory testing and non-functional testing unless they are explicitly planned and supported. Similarly, when teams adopt a strong whole team quality ownership mindset, experience-based test techniques, such as exploratory testing from Quadrant 3, are applied continuously and integrated into development rather than confined to the end of iterations.

Delivery frequency and release strategy

They determine how much time is available for testing and directly affect the degree of test automation. Short release cycles and continuous delivery models demand extensive test automation in Quadrants 1 and 4. Here, the selection of methods leans toward maintainable, low-cost feedback options (e.g., component testing, API testing, smoke testing). Conversely, in longer-cycle or milestone-based releases, there may be more opportunity to test manually in Quadrant 3 (e.g., exploratory testing, in-depth usability testing, and user acceptance testing).

Observability

If observability is low, relying on implicit passive monitoring becomes difficult, increasing the need for Quadrant 2 elements such as simulations and prototypes. If testability is high, teams can introduce instrumentation and test-only interfaces to enable more targeted testing across all quadrants.

Product maturity and change frequency

In early-stage products or systems undergoing rapid prototyping, the test strategy typically favors low-overhead, high-value feedback mechanisms. Exploratory testing from Quadrant 3 takes precedence over formal modeling or exhaustive coverage. As the product matures, regression risk increases, requiring greater investment in maintainable test automation and structured testing (e.g., component testing from Quadrant 1 or functional tests from Quadrant 2 designed with black-box test techniques) to mitigate the growing cost of change. Quadrants 1 and 2 increasingly depend on coverage-based test techniques and data-driven test automation to sustain velocity.

Tooling constraints and infrastructure maturity

Where modern CI/CD pipelines, test doubles, and containerized environments are available, test automation and virtualization can be fully exploited using Quadrants 1 and 2, while tools also enable the application of non-functional testing from Quadrant 4. Without these capabilities, the test strategy may be limited to manual testing (e.g., exploratory testing or user acceptance testing from Quadrant 3) or rely on isolated test environments. This introduces risk and necessitates compensatory measures such as increased exploratory coverage and feature toggles to manage uncertainty in production.

Risk appetite and leadership culture

A risk-averse or regulatory-driven organization may enforce process-heavy test approaches, prioritizing documented coverage, extensive test conditions, and quantitative metrics. Teams in these contexts tend to apply black-box test techniques for functional testing from Quadrant 2 (e.g., boundary value analysis, state transition testing, decision table testing). A culture that values adaptive learning and experimentation supports exploratory testing and user acceptance testing from Quadrant 3, as well as simulations from Quadrant 2. Effective testing requires balancing all aspects – Quadrants 1 and 2 typically ensure repeatability and confidence, while Quadrants 3 and 4 foster innovation and learning.

Customer feedback mechanisms and production observability

They can extend the test strategy beyond delivery. Test approaches such as A/B testing, customer usage telemetry, and failure pattern analysis support ongoing validation of business value and operational fault tolerance. These test practices extend Quadrants 3 and 4 into production environments, enabling short feedback loops.

The test strategy should be revisited periodically, e.g., based on retrospective outcomes. As a product and its context evolve, the test strategy should adapt its focus by shifting across the quadrants, continually rebalancing the mix of scripted and unscripted testing, as well as manual and automated testing. The more dynamic and continuous the delivery model, the more often this reevaluation becomes necessary.

3.2 Test Monitoring and Test Control

In Agile software development, test monitoring uses lightweight metrics displayed via burn-down charts, cumulative flow diagrams, automated dashboards, and similar visualizations to continuously track test progress and product quality. These visualizations are updated frequently and are accessible to all team members to enable shared situational awareness and immediate action when deviations occur. Typically, metrics focus on delivered value and on providing actionable, real-time feedback rather than reporting only periodically. The emphasis is on understanding trends in defect discovery, test automation coverage, exploratory testing findings, and production monitoring data rather than tracking compliance with predefined test plans.

In Agile teams, test control is exercised collaboratively during daily stand-ups, iteration reviews, and retrospectives, where data from CI/CD pipelines drives decisions. Adjustments are made in real time, often at the user story or feature level, including prioritizing exploratory test sessions, refining acceptance criteria, and temporarily focusing on high-risk areas. These activities are carried out by the whole team, not just testers, ensuring that changes to scope, schedule, and quality objectives are discussed and agreed upon immediately. Test automation feedback, production monitoring signals, and customer feedback loops are treated as critical triggers for test control decisions.

Test monitoring and test control should be integrated into the daily workflow, using cross-functional collaboration and continuous feedback from both pre-production and production environments. The decisions are frequently reassessed based on evolving business priorities and the current state of delivered increments, supporting the Agile principle of responding to change quickly while still protecting quality goals.

3.3 Test Reporting

Test reporting includes various activities, but this section focuses on various coverage metrics to reflect test progress, support decision-making, and enhance transparency. These coverage types should align with team goals, support rapid feedback cycles, and be lightweight enough to be useful without causing unnecessary overhead. Agile teams prefer coverage indicators that foster shared understanding of system quality and risks.

Requirements coverage is common, especially in the context of ATDD or BDD. By linking automated acceptance tests to user stories or examples, teams can demonstrate which functional aspects have been specified, developed, and verified. These acceptance criteria become traceable units of coverage, helping the team communicate user story readiness and completeness.

Code coverage can be used selectively, especially in highly automated test environments. It should be

interpreted cautiously and should not be treated as a proxy for quality. While useful for identifying untested logic paths, high code coverage alone does not guarantee meaningful testing. Agile teams may combine this with fault seeding or assertion quality reviews to gain deeper insights into test effectiveness.

Exploratory testing related coverage can be reported using session-based test management or test charter completion rates. This is particularly valuable for business-facing exploratory testing, usability testing, or testing in risk areas that lack formal specifications. Visual dashboards can be used to represent what has been explored and which scenarios or personas have been exercised.

Infrastructure and environment coverage, particularly in DevOps and continuous delivery contexts, may be tracked to show which platforms, configurations, or APIs have been exercised. Monitoring and observability tools also enhance operational coverage reporting by revealing usage patterns and anomalies in production.

In all cases, coverage metrics in Agile should be discussed collaboratively, not just consumed as abstract indicators. They must support informed decision-making and adapt to organizational, project, and team contexts, rather than being imposed as prescriptive metrics. Test metrics are tools for communication and improvement in iterative and adaptive software development lifecycles. Teams should resist the illusion of completeness created by metrics and instead provide tailored information to meet the needs of different stakeholders.

3.4 Test Process Improvement

3.4.1 Metrics for Test Process Improvements

Metrics-based test process improvement in Agile software development requires careful interpretation of both product and process indicators, aligning with continuous improvement principles and drawing on team capabilities and tools. Except for genuine key performance indicators, metrics should serve as feedback mechanisms rather than performance targets, to avoid misuse that could distort behavior and lead to local optimization.

When the defect detection percentage declines unexpectedly, a root cause analysis may reveal ineffective exploratory testing or insufficient test automation. In response, the team may consider refining their test chartering techniques, improving session-based test management, or investing in tooling for observability or exploratory testing support.

If tests repeatedly fail due to environmental instability, investing in test environment management, service virtualization, or containerized solutions may improve stability and confidence in test results.

High cycle times for resolving defects might indicate weak collaboration across roles or siloed responsibilities. A possible improvement could involve adopting whole team approaches to defect triage, involving testers earlier in refinement sessions, and embedding Agile test leaders to reinforce shared ownership of quality. Pairing metrics such as mean time to failure and mean time to repair with retrospectives enables teams to recognize bottlenecks in feedback loops and adapt accordingly.

Low coverage on critical paths, as reported by code coverage tools, may prompt efforts to strengthen automated coverage at the component testing or component integration testing levels, potentially rebalancing the test pyramid toward more granular, fast-executing tests. However, teams must assess whether these tests provide meaningful insights or simply inflate metrics.

Redundant or fragile tests indicated by high failure rates with low defect yields should be refactored or replaced using design patterns such as page objects or test data builders. For more information on design

patterns in test automation, see (ISTQB-TAE, v2.0).

A declining participation in retrospectives or quality discussions may reflect a cultural issue. Coaching may be required to facilitate psychological safety and reinforce testing as a shared responsibility. In such a context, qualitative metrics, such as perceived trust and openness in collaboration, can be just as important as numerical indicators.

When test debt accumulates, as evidenced by an increasing backlog of flaky tests or manual regression test time, a test strategy may schedule time in each iteration to maintain tests, rotate test ownership, and apply test automation refactoring practices. Measurement of test automation effectiveness should include not just pass/fail rates, but also maintenance effort and test execution lead times.

Monitoring-based metrics, such as mean time to recovery or production defect escape rates, might initiate test process improvements. A high MTTR may indicate missing system alerts, resource constraints, or insufficient exploratory testing of operational behaviors. This could lead to incorporating chaos testing or software development techniques such as canary releases, feature toggles, or dark launches.

For people development, metrics such as pair testing frequency, defect learning session counts, or cross-functional contribution ratios may indicate whether knowledge sharing and capability growth are occurring. A lack of cross-functional collaboration across roles may necessitate mentoring, role rotation, or structured learning pathways such as community of practice initiatives.

Tool-related improvements can also be informed by lead time metrics from code commit to feedback, flaky test analysis, and integration failure rates. Tools that enable short feedback loops, such as dashboards, automated root cause analysis, or smart test selection, can directly support team learning and enable faster responses.

Any improvement action should be measurable and validated. For this, retrospectives can play a vital role. Metrics must be traceable to test objectives and risk priorities, rather than arbitrary benchmarks. Models such as GQM (Goal-Question-Metric) are helpful for structuring the use of metrics in a goal-oriented manner. Finally, teams should maintain critical thinking and contextual sensitivity to prevent metrics from becoming disconnected from the core test mission.

3.4.2 Test Process Improvement in Agile Software Development

For test process improvement in Agile software development, teams can adopt structured yet lightweight approaches that emphasize collaboration, context-awareness, and iterative experimentation:

Conduct focused retrospectives. Use lightweight, collaborative practices, such as test-specific retrospectives, to identify testing challenges and opportunities for enhancement. Improvement efforts must be grounded in the team's current context, not imposed from outside. Retrospectives focused specifically on testing, distinct from general Agile retrospectives, can provide focused insight into the effectiveness of testing practices and their alignment with business and technical needs (ISTQB-ATLAS, v1.0), and help build shared ownership of quality goals.

Establish a baseline assessment. Perform structured Agile test assessments to identify gaps in the current test approach. Tools like self-assessments or checklists can prompt reflection on core Agile testing dimensions such as collaboration, feedback loops, and continuous integration. From there, targeted improvement can follow a gradual, iterative path.

Align via joint risk workshops. Include business, testing, and development representatives to build a shared understanding of critical product risks and where test effort is best focused. This ensures that improvements support both business and team goals.

Strengthen CI/CD feedback loops. Integrate practices such as test-first approaches and pair testing to enable early, actionable feedback. Use exploratory testing not just to find defects, but to uncover unknown risks and test gaps, supporting adaptive test process refinement.

Run small experiments. Adopt techniques, such as session-based test management, to explore test process changes incrementally. Results of these experiments should be discussed openly within the team, using retrospectives to adjust the test strategy as needed.

Leverage visible metrics with care. Monitor test automation health, defect trends, and coverage across CI/CD pipelines, but resist the temptation to benchmark teams against each other. Instead, focus on metrics that help teams understand their own effectiveness and make their work more visible.

Teams can use practices such as Value Stream Mapping, PDCA cyle or casual loop diagrams (see (ISTQB-ATLAS, v1.0)). Improvements must account for sociotechnical factors, such as organizational culture and communication practices. Resistance to change (see: (Prosci, 2026)) should be addressed through inclusive conversations that draw on change facilitation strategies.

Above all, continuous improvement in Agile testing must be rooted in critical thinking, ongoing learning, and whole team collaboration. Applying fixed models or processes should be adapted carefully to the context. Each improvement should act as a hypothesis to be tested through action, learning, and adaptation.

4 Shift Left - 135 minutes

Keywords

acceptance-driven development, behavior-driven development, quality, shift left, test basis, testboarding, testware

Domain Specific Keywords

bias, example mapping, specification by example, storyboarding, structured conversation, test session sheet, user journey, user story, user story slicing

Learning Objectives:

4.1 Using Shift Left to Improve Test Basis Quality

CTAL-AT-4.1.1 (K2) Give examples of how testware can be used as a form of requirements

CTAL-AT-4.1.2 (K2) Explain how storyboarding and testboarding can be used to increase the quality of the test basis

CTAL-AT-4.1.3 (K2) Explain how example mapping can be used to increase the quality of the test basis

CTAL-AT-4.1.4 (K2) Give examples of how biases can negatively affect the product quality

CTAL-AT-4.1.5 (K3) Apply user story slicing to achieve testable user stories

4.2 Shift Left and Requirements Engineering

CTAL-AT-4.2.1 (K2) Explain how requirements engineering supports shift left

Introduction

Shift left emphasizes bringing quality activities as early as possible in the software development lifecycle, focusing on improving the test basis from the outset. Testware, such as acceptance criteria, checklists, and automated test scripts, can act as living requirements or support and document requirements, ensuring clarity and shared understanding. Techniques like storyboarding and testboarding help visualize workflows and align perspectives, while example mapping refines acceptance criteria through collaborative discussion. Awareness of cognitive biases safeguards against flawed assumptions that may reduce product quality. Practices like user story slicing ensure user stories are small, focused, and testable. In parallel, effective requirements engineering supports shift left by combining analysis, documentation, and validation, and by selecting suitable elicitation methods, such as interviews, workshops, or prototyping, to capture stakeholder needs. By shifting left, the team may detect anomalies and defects sooner, reduce rework, and deliver higher-value products. Emerging AI and large language model tools can support shift-left by analyzing requirements for ambiguities, suggesting test scenarios from acceptance criteria, or identifying gaps in user story coverage. Testers should evaluate such tools critically, understanding their limitations, and use them to augment rather than replace collaborative human analysis.

4.1 Using Shift Left to Improve Test Basis Quality

4.1.1 Testware as Requirements

In Agile software development, testware such as user stories with acceptance criteria, checklists, BDD scenarios, acceptance level test cases, examples, test charters, contracts, or automated test scripts can be treated as a form of requirements when used collaboratively, and reviewed and applied for each iteration. These work products help teams clarify expectations, drive shared understanding, and support ongoing validation of software behavior. They are often created during conversations with business stakeholders, making them living requirements rather than static documentation.

Requirements used in behavior-driven development (BDD), acceptance test-driven development (ATDD) or specification by example (Adzic, 2011) are derived from conversations between the team and business stakeholders, and they are often documented in a format that can be executed as tests. These executable specifications embody the acceptance criteria serving both as living requirements and enduring regression tests.

In exploratory testing, session-based test charters often emerge from high-level product ideas or risk assessments. These test charters guide investigation and help identify missing or misunderstood functionality. The test session sheets generated during the exploratory testing session provide informal yet valuable insights into evolving requirements.

In DevOps and continuous delivery models, logs from monitoring, customer analytics, and feedback loops may serve as evolving requirements, when interpreted and translated into testable hypotheses or decision rules, especially when change is driven by production behavior and user interaction data. Teams can capture these insights in test cases or checklists for ongoing validation.

Other forms of testware, such as mind maps, checklists, and visual models (e.g., flow diagrams or state diagrams) may be used during Agile planning and can directly guide development work. They translate implicit understanding into shared reference points.

4.1.2 Storyboarding and Testboarding

Storyboarding and testboarding are collaborative techniques that improve the quality of the test basis by enhancing shared understanding and the early discovery of ambiguities, risks, and omissions. From a software testing perspective, they are particularly effective in Agile teams focused on feedback and whole team collaboration.

Storyboarding visualizes user journeys by presenting the expected behavior as a sequence of steps or scenes (Walker et al., 2013). This narrative approach reveals assumptions, edge cases, and interactions that might not be obvious in textual requirements. It prompts critical questions about workflow logic, enabling stakeholders to discover inconsistencies or incomplete flows before code is written. Because storyboards focus on user interactions, they help uncover gaps in the test basis related to usability, flow transitions, and persona-specific needs.

Testboarding is a simple practice that helps to visualize what needs to be tested for each feature, complementing storyboarding. In addition to validating the acceptance criteria, testboarding shows the main and alternative scenarios, which leads to early alignment among testers, developers, and business representatives on terminology, system behavior, and test priorities. This shared understanding reduces interpretation errors during test design and test execution, thereby increasing test case maintainability.

By combining these techniques, teams create a shared space for exploratory thinking. Testers can pose "what if?" questions and propose negative or alternative scenarios, prompting early improvements in the user story and its acceptance criteria. Practices that help explore the test basis early improve risk identification and reduce late discovery of critical defects by allowing the testers to properly prioritize tests.

These techniques are most powerful when used iteratively - revisiting storyboards and testboards during backlog refinement and iteration planning exposes changes in scope, risks, or understanding. They can serve as living documentation that evolves alongside the product and test basis, helping testers detect when requirements drift or quality goals are compromised.

4.1.3 Example Mapping

Example mapping (Wynne, 2015) is a collaborative technique that refines user stories through structured conversation. It helps teams identify business rules, associated examples (which can become test cases), and capture questions to deepen shared understanding. The team creates colored cards to represent the different elements of the discussion (e.g., yellow for the user story under review, blue for business rules, green for examples that illustrate the rules, and red for questions). The rules are grouped under a user story, and examples are grouped under corresponding rules. The team continues creating cards until they are satisfied that the scope of the user story is clear, or the timebox expires. When time runs out, it may indicate that the user story is too complex or needs refinement, and the team decides on the next steps.

As the conversation flows, the team quickly builds up a visual representation of the user story that reflects its current shared understanding. Too many "question" cards suggest that the team still has a lot to learn about this user story. A lot of "rule" cards, or a single rule with many "example" cards, suggest that the user story is large and complex and should be sliced (see *Section 4.1.5*). In such cases, the team splits the rules and examples between the original story and the new slice, creates another yellow (user story) card for the sliced user story, and puts it on the backlog.

From a testing perspective, example mapping strengthens the test basis by grounding abstract requirements in concrete, agreed examples. These examples reduce ambiguity, uncover hidden assumptions, and reveal conflicting interpretations early.

In Agile software development, the test basis often evolves during refinement and implementation. Example mapping improves quality by ensuring a shared understanding across the whole team and by clarifying acceptance criteria that directly inform acceptance test-driven development (ATDD) and behavior-driven development (BDD). It promotes shift left validation by uncovering gaps in the specification before coding begins.

Example mapping supports the creation of test charters for exploratory testing and helps testers identify relevant test conditions and alternative scenarios as the concrete examples and questions naturally suggest testable boundaries and edge cases to explore. When used iteratively, it makes the test basis more resilient to change, as traceability from examples to rules and user needs is preserved throughout development.

4.1.4 Biases in Agile Testing

A bias is a systematic deviation from neutrality or accuracy that affects how people think, judge, or act, sometimes leading them away from objectivity or fairness. Biases can significantly compromise product quality by distorting the perception of risk, limiting the diversity of testing, and weakening feedback mechanisms critical to Agile software development practices. In testing, biases may originate from individuals, teams, or organizational culture, and influence decisions unconsciously.

Biases that are particularly relevant to Agile testing include:

- Confirmation bias may lead to designing and executing tests that only verify expected behavior, neglecting potential failure paths or unusual conditions. This narrows the scope of exploration and increases the risk of undetected defects, especially in complex systems where implicit assumptions drive development decisions.
- Anchoring effect occurs when early requirements or stakeholder expectations fixate teams on specific interpretations, overlooking alternative behaviors. For example, if initial user stories focus exclusively on happy paths, testing may underrepresent edge cases, accessibility, or internationalization concerns.
- Conformity bias can impair quality when diverse opinions are discouraged, testers copy one another's behaviors or beliefs rather than following their own independent judgment, or testers defer to dominant voices in cross-functional teams. This undermines the value of the whole team approach, which relies on shared responsibility for quality and critical feedback loops throughout the software development lifecycle.

Biases can affect the design of acceptance criteria, reducing clarity and increasing ambiguity in automated acceptance tests or executable specifications. In practices such as specification by example and ATDD, failing to challenge assumptions during collaborative specification can lead to ineffective coverage.

The risk of bias is increased by the whole team approach. To mitigate bias, teams should promote diversity of thought, leverage exploratory testing with session-based testing, and foster a culture of psychological safety that encourages dissent and questioning. Practices such as debriefs after exploratory testing sessions, paired testing, the use of tissue testers, and root cause analysis can help reveal the influence of bias and strengthen collective awareness.

4.1.5 User Story Slicing

User story slicing is the practice of breaking down large user stories into smaller, more manageable pieces that deliver value, are feasible within an iteration, and are testable. Properly sliced user stories facilitate

continuous delivery and enable earlier feedback, thereby supporting Agile principles.

To apply user story slicing in a way that supports testability, the team starts by identifying the different dimensions along which a user story can be split:

- **Workflow slicing** decomposes the user story based on user tasks or steps in a business process. Each slice should be focused on a single task or interaction, ideally one that results in observable system behavior. This makes it easier to define precise acceptance criteria and expected results.
- **Slicing by data complexity** begins with a single happy path example using the simplest data and moves alternative data conditions and edge cases to separate user stories. This allows the team to verify core functionality early and incrementally build coverage. Each slice should include just enough variation to enable meaningful tests without overloading the scope.
- **Slicing by interface** separates backend processing logic from user interface (UI) functionality when each can be tested independently. This enables earlier test automation and aligns with architectural layering, especially in continuous integration environments. The team builds backend user stories that expose outputs via APIs before working on user stories that include UI workflows.
- **Slicing by scenario** starts with the most common business scenario and builds on that by creating additional slices for less frequent or more complex cases. The team may use example mapping or structured conversations to elicit these scenarios during refinement. Each scenario slice should be verifiable through specific examples that form the basis for business-facing tests.
- **Vertical slicing** ensures that each user story includes a complete thin slice of functionality across layers, from database to UI, where appropriate. This helps teams deliver demonstrable value, perform end-to-end testing, and verify integration across components. However, including too much in a single slice should be avoided. If the end-to-end scope becomes unmanageable, the team should revert to scenario or workflow slicing.
- **Slicing based on constraints** focuses first on general assumptions and then considers conditions derived from known constraints. For example, the team starts with a user story that assumes the user is authenticated, then slices off a separate user story to handle authentication failures. This helps isolate test conditions and improves clarity when defining boundaries.

Acceptance criteria often determine the success or failure of user story slicing. When slicing, the team cannot just cut the user story; it needs to rethink and rewrite the acceptance criteria so that each slice makes sense. For example, large user stories usually have a set of broad acceptance criteria that should be refined after slicing to match the narrower scope. Acceptance criteria should focus on one narrow user outcome per slice, and combining multiple behavioral variations in one slice should be avoided. When slicing, the original acceptance criteria should not be copy-pasted into each slice. Instead, they should be rewritten per slice so each one is testable, can be demonstrated, and corresponds to its slice. The set of acceptance criteria across slices should eventually cover the full original story.

Regardless of the slicing strategy, the team should always make testability a primary criterion. This is done by asking whether the slice can be independently validated, whether clear acceptance criteria can be derived, and whether it supports the creation of business-facing tests. If any of these conditions are not met, the user story needs further refinement.

4.2 Shift Left and Requirements Engineering

Shift left moves quality activities earlier in the software development lifecycle to detect anomalies and defects early, reduce rework, shorten delivery cycles, and improve alignment with business needs. All the main activities in requirements engineering – requirements elicitation, analysis, specification, and validation – can support shift left by reducing misunderstandings, preventing defects, and minimizing costly rework later in the software development lifecycle:

- Requirements elicitation supports shift left by capturing stakeholder needs early and continuously, enabling development and testing to start with a clear understanding of stakeholder expectations. Continuous elicitation through user stories, backlog grooming, and collaborative workshops ensures that features are defined, testable, and validated, reducing defects and rework later in the iteration.
- Requirements analysis supports shift left by identifying ambiguities, conflicts, risks, and dependencies early, ensuring that development and testing start with clear, feasible, and prioritized requirements. By analyzing requirements, teams can design appropriate solutions, plan tests, and prevent defects before coding begins, reducing costly rework later in the software development lifecycle.
- Requirements specification supports shift left by providing clear, precise, and testable documentation of what the system must do, enabling developers and testers to plan and validate work early. Well-specified requirements reduce misunderstandings, guide early test creation, and help find anomalies before implementation, minimizing downstream defects and rework. Agile testers typically work with free-form textual representations like user stories and epics. When more structure is needed, they may use semi-formal scenario models such as use cases, business process models (e.g., BPMN), or state transition diagrams, depending on the context. Scenario models can illustrate interaction flows, including edge cases that may not be captured in high-level descriptions. Glossaries are often maintained to ensure shared understanding of domain-specific terms, especially when defining acceptance criteria.
- Requirements validation supports a shift left approach by ensuring that requirements are correct, unambiguous, complete, consistent, and testable before development begins. Early validation through reviews, prototyping, and modeling helps detect anomalies and gaps, reducing defects, rework, and costly late-phase changes.

5 Agile Approaches and Test Techniques - 285 minutes

Keywords

Agile software development, edge case, exploratory testing, mob testing, pair testing, test case, test charter, test heuristic, test mnemonic, test smell, tour, vibe testing

Domain Specific Keywords

epic, FEW HICCUPPS, I SLICED UP FUN, RCRCRC, SFDIPOT, TERMS

Learning Objectives:

5.1 Exploratory Testing

CTAL-AT-5.1.1 (K2) Explain test heuristics

CTAL-AT-5.1.2 (K2) Give examples of test mnemonics related to testing in Agile software development

CTAL-AT-5.1.3 (K2) Explain test tours

CTAL-AT-5.1.4 (K4) Analyze user stories and epics to create test charters

CTAL-AT-5.1.5 (K3) Apply exploratory testing to support testing in Agile software development

5.2 Assisted Testing

CTAL-AT-5.2.1 (K2) Explain mob testing

CTAL-AT-5.2.2 (K2) Explain pair testing

CTAL-AT-5.2.3 (K2) Explain vibe testing

5.3 Test Smells

CTAL-AT-5.3.1 (K3) Use test smells to evaluate the quality of test cases

Introduction

Testers must adapt their test strategies and tools to match the fast-paced, iterative nature of Agile software development projects. This chapter explores a set of test techniques and collaboration methods that enhance the effectiveness and efficiency of testing in Agile software development. They emphasize flexibility, real-time feedback, and continuous learning – core values of Agile software development methodologies.

5.1 Exploratory Testing

5.1.1 Test Heuristics

Test heuristics are general guidelines or rules of thumb that help testers make informed decisions, solve problems, and guide their exploration of a system under test. They are not guaranteed to be correct in every context, but they serve as practical strategies that often lead to valuable results when:

- time or resources are limited;
- the specification is incomplete or unclear;
- the tester performs experience-based testing;
- regression testing is performed without full test automation coverage;
- the tester wants to think like a user or simulate real-world use cases.

Examples of test heuristics include:

Guidelines – recommended procedures or steps that direct testers on specific actions to take or approaches to follow in testing. Examples include: “test early and often”, “testing is about information, not just passed/failed”, “always consider boundaries”, “try common workflows and unexpected or invalid behaviors”, “change one variable at a time to isolate effects when testing options, configurations, or inputs”, “test how features work together in sequence, not just in isolation”.

Generic checklists – reusable lists of common things to test, used as reminders or prompts to explore areas which could otherwise be missed. Checklists can be focused on various areas, such as quality characteristics, test types, test levels, risk areas, user roles, and test environments. Examples include usability heuristics by Nielsen (Nielsen, 1994), OWASP Top 10 web application security risks (OWASP, 2025), defect taxonomies such as Beizer’s taxonomy (Beizer, 1990) and the IEEE 1044 taxonomy (*IEEE 1044 – Classification for Software Anomalies*, 2009).

Rules of thumb – unwritten experiential “laws” or commonly accepted truths. Examples include: “if something looks too simple, it is probably hiding complexity”, “if it broke before, it will probably break again”, “defects cluster together”, “if it is hard to test, it is probably hard to use”, “what works with one user might fail with many”.

Mnemonics – memory aids that help testers quickly recall a set of test conditions. Examples include SFDIPOT (see *Section 5.1.2*).

Analogies and metaphors – allow testers to compare the system to familiar concepts and then apply reasoning from those concepts to generate test conditions. Examples include test tours (see *Section 5.1.3*).

5.1.2 Test Mnemonics

Mnemonics are memory aids, usually in the form of acronyms, abbreviations, rhymes, or patterns. Mnemonics do not rigidly prescribe what to test, but help testers recall complex information sets and broaden their perspective during exploratory testing and reviews. However, they require domain, contextual, and risk knowledge, as well as critical thinking and team trust to be effective. In software testing, test mnemonics are commonly used to:

- guide exploratory testing by providing structure to free-form testing without rigid test cases;
- spark creative and critical thinking by prompting testers to think in ways they might not have naturally considered;
- achieve broader coverage by ensuring key areas are not forgotten;
- serve as mental checklists during test planning or test execution, reducing cognitive load by condensing complex concepts into memorable formats.

Software testing employs various test mnemonics, and testers can create their own custom sets. Some popular test mnemonics include:

- SFDIPOT (Structure, Function, Data, Interfaces, Platform, Operations, Time) – supports the general analysis of the system under test (Bolton, 2014). It can be used when exploring a new system or planning test charters.
- RCRCRC (Recent, Core, Risky, Configuration, Repaired, Chronic) – supports regression testing, where each letter represents a word to help discover test conditions (Johnson, 2012). It can be used to decide coverage in Agile software development iterations or regression test cycles.
- I SLICED UP FUN (Inputs, Store, Location, Interactions/Interruptions, Communication, Ergonomics, Data, Usability, Platform, Function, User scenarios, Network) (Kohl, 2010). It can be used to support mobile application testing in generating test conditions.
- FEW HICCUPPS (Familiarity, Explainability, World, History, Image, Comparable products, Claims, Users' desires, Product, Purpose, Statutes) – supports the identification and application of test oracles (Bolton, 2012). It can be used to identify defects by observing different types of inconsistencies.
- TERMS (Tools and Technology, Execution, Requirements and Risks, Maintenance, Security) (Gareev, 2019). It helps to remember different factors that influence the success of test automation.

5.1.3 Test Tours

Test tours use city tour metaphors to guide exploratory testing of a system. The concept is to help testers explore software, much like tourists explore a new place, using different types of "tours" to uncover various kinds of problems. Each tour focuses on a specific perspective, guiding the tester to examine the application from a particular viewpoint. The tours are designed to be informal, lightweight, and creative, making them ideal for exploratory testing, session-based testing, or as test condition generators.

Examples of test tours, adapted from city districts (Whittaker, 2009), include:

Business district. For software, this refers to the core features. This area is bounded by startup and shutdown code and contains the features and functions that users interact with in the software.

Historical district. For software, this refers to legacy code and the history of defective functions or features. Like real history, legacy code is often poorly understood, and many assumptions are made when legacy code is included, modified, or used.

Tourist district. Many cities have districts that tourists primarily visit. Locals and residents tend to avoid these areas. Software mirrors this, with novice users gravitating toward features and functions that experienced users no longer need.

Entertainment district. In cities, these districts offer relaxation and entertainment. Software, like cities, also has supportive features. These tours complement the tours through other districts and fill in the gaps of a comprehensive test plan.

Hotel district. In cities, these districts are a place for rest. In testing, it is a place for the software tester to step away from the primary functionality and popular features and test some of the secondary and supporting functions that are often overlooked or underrepresented in a test plan.

Seedy districts are dangerous, but they still attract a certain class of tourist. Seedy tours are essential for testers as they uncover areas of vulnerability that could be highly problematic to users if they remain in the product. This area is primarily concerned with negative testing and employs test techniques such as error guessing or fault attacks.

5.1.4 Test Charter Creation

Test charters align closely with Agile software development principles and provide a structured yet flexible approach to exploratory testing (see *Section 5.1.5*). A test charter is a short, focused statement that outlines the purpose, scope, and test objectives of an exploratory testing session. It acts as a guiding mission statement or testable hypothesis for the tester during the test session. Test charters can be created based on various sources, such as user stories, acceptance criteria, product risks, or iteration goals. At a minimum, a test charter contains:

- Mission/test objectives – high-level focus or testable goals for the session (e.g., "Validate user login under load")
- Scope – specific areas of interest within the system under test, test level, test conditions, test techniques to be used, exit criteria, priorities, what to cover, and what will not be covered

and may also include other information, such as:

- Actor – intended user of the system
- Purpose – an objective the actor wants to achieve
- Setup – what needs to be in place to start exploratory testing (e.g., test environment)
- Priority – relative importance of this test 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 test charter
- Activities – a list of ideas of what the actor may want to do with the system, what would be interesting to test, and what test techniques could be used
- Test oracle notes – how to evaluate the product to determine correct test 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
- Limitations – e.g., what the product must never do
- Constraints and risks – e.g., regulations, rules, and standards used

To derive test charters from user stories and epics, Agile testers analyze them to identify test objectives and mission focus. They extract the intent of the user story or epic (e.g., who the actor is and what their purpose) and translate it into a mission statement for the test session. Related user stories can be grouped to evaluate integrated behavior.

Agile testers identify what needs to be learned, verified, or explored during the test session. They can use the 5W1H technique, also known as the Kipling method, to extensively answer existing questions and trigger ideas that could contribute to the resolution of a problem:

- What feature is being delivered?
- Who are the users?
- When is it used?
- Where is it used?
- Why is this feature important?
- How does it integrate with other features?

These questions help uncover scope, setup, and priority. Various test heuristics (see *Section 5.1.1*, *Section 5.1.2*, and *Section 5.1.3*) can be helpful in answering these questions and generating test conditions, test data, and test activities to be included in the test charter.

Acceptance criteria define what must be true for a user story to be accepted. These can be directly translated into test conditions. While acceptance criteria guide expected behavior, they also highlight areas to explore beyond the "happy path": What happens if the acceptance criteria are not met? What edge cases or negative test conditions arise? Are the acceptance criteria clear or possibly ambiguous? This helps testers develop test conditions for both positive and negative tests. Acceptance criteria often imply or specify data requirements, such as valid input values, preconditions, or specific formats or ranges.

Acceptance criteria can also serve as lightweight test oracle notes because they define expected behaviors, establish pass/fail thresholds, and align with business representatives' expectations. However, testers should remember that acceptance criteria are not exhaustive test oracles – they do not capture all quality aspects. Exploratory testing is still needed to uncover defects outside the scope of acceptance criteria. Agile testers should treat missing, vague, or ambiguous acceptance criteria as opportunities for exploratory testing to uncover unstated variations and edge cases.

5.1.5 Performing Exploratory Testing

In Agile software development environments, where requirements change frequently, exploratory testing offers the flexibility and responsiveness that scripted approaches often lack. Exploratory testing is also important in Agile software development projects due to the limited time available for test analysis and the limited details of user stories.

Exploratory testing can be applied at various points in the Agile software development lifecycle, including:

- during iteration execution (to quickly test new features or changes before or alongside scripted testing)
- in iteration reviews/demos (to explore the product informally while gathering feedback)
- after major changes (to uncover integration or regression defects that scripted tests might miss)
- when acceptance criteria are vague or minimal (to clarify expectations and find edge cases)

Exploratory testing is typically based on test charters and performed in test sessions (Ghazi et al., 2017). A test session is a time-boxed, uninterrupted period (usually 60–120 minutes). These test sessions help organize test efforts and provide structure in a flexible test approach. A test session focuses on a specific test charter and is often documented with test session sheets, observations, and findings. The typical structure of an exploratory testing session is as follows:

- Pre-session setup. The test charter is defined, the test environment is prepared, and the test session timebox is set.
- Test execution phase. The tester learns how the product works, explores, and evaluates it. The tester follows the intent of the test charter but adapts as new information is discovered. They use test heuristics and test mnemonics to stimulate test conditions. Because exploratory testing is unscripted, adequate documentation is essential for reproducibility, test reporting, and improvement. Standard note-taking methods include session sheets, free-form notes, screenshots, screen recordings, and mind maps. Session recording tools can also be used. Documented information typically includes coverage (including risk coverage), evaluation notes, recording of the system's actual behavior, and anomalies detected.
- Post-session review. The exploratory tester discusses test session results with the stakeholders, including:
  - test charter versus reality (did the tester follow the test charter, find unexpected areas, and meet session goals?);
  - defects found;
  - questions raised (e.g., unclear requirements or technical uncertainties);
  - next steps (are new test charters or regression tests needed?).

When performing exploratory testing, the tester is not limited in any way regarding test approaches, test techniques, or test types. They can use black-box test techniques and white-box test techniques. However, due to the nature of exploratory testing, experience-based testing and test heuristics are most commonly used (see *Section 5.1.1*, *Section 5.1.2*, and *Section 5.1.3*).

Exploratory testers need to apply test oracles flexibly and contextually, often relying on their experience, product knowledge, and team collaboration. In exploratory testing, test oracles are often:

- consistent with user stories or acceptance criteria (see *Section 5.1.4*)
- expressing common sense or user expectations
- consistent with similar features or past versions
- following standards or guidelines (e.g., accessibility or UI standards)

Test oracle categories supporting exploratory testing may include FEW HICCUPPS test mnemonic (see *Section 5.1.2*).

5.2 Assisted Testing

5.2.1 Mob Testing

Mob testing (also known as ensemble testing) is a collaborative test approach in which a small group (typically 5-8 people) tests the same system simultaneously. It builds on the concept of swarming, where multiple team members converge on a given testing problem. This real-time collaboration combines diverse skills, perspectives, and immediate feedback to enhance testing quality, speed, and creativity.

The mob testing setup includes a shared screen, a computer, and a whiteboard (in case of remote work the environment may be virtual, rather than physical). It uses the following roles:

- Moderator – session coordinator who oversees session flow, ensures role adherence, manages time, and fosters balanced participation.
- Navigator – the primary tester who makes final decisions on actions, incorporating insights and guidance from the mob.
- Driver – a typist implementing the team's collective instructions, and does not typically take the lead on strategy, ensuring ideas pass through another person's mind first.
- Mob – a group of observers who monitor progress and contribute insights as needed.

Mob testing consists of two parts: mobbing and retrospective. The mobbing session starts with arranging people in a circle with the driver at the keyboard. People rotate every so often (usually every 4 minutes); the navigator becomes the driver, the driver becomes the first person in the mob, and so on.

The moderator sits at the back and does not rotate with the rest of the mob. If it is necessary for them to step in, they can pause the mob and assume whatever role is needed, except that of the driver. The mob rotates frequently, forcing everyone to pay attention. The mobbing session typically lasts 2 hours.

After the mobbing session ends, the retrospective follows to improve the process for future sessions. Typically, the moderator silently collects observations from each participant on sticky notes. The team then groups similar items to reveal patterns and opportunities for continuous improvement.

5.2.2 Pair Testing

Pair testing is a test approach in which two people work together at a single workstation (or a virtual environment in case of remote work or a distributed team) to test the same test object in real time (e.g., tester + tester, tester + developer, tester + product analyst). Typically, in pair testing, one person operates the computer (the driver), and the other observes, reviews, takes notes, and strategizes (the navigator). Pair testing draws inspiration from pair programming, a practice popularized by extreme programming (XP).

Pair testing supports knowledge sharing by encouraging learning from each other's domain expertise and test strategies, and is particularly useful when onboarding new team members. Working in pairs helps find defects faster because of the collaborative synergy it creates. This also improves coverage by combining diverse perspectives. Real-time feedback helps refine test conditions, which leads to more creative and effective test cases. It can also improve communication by reducing misunderstandings about requirements or functionality. Pair testing keeps testers engaged and reduces the likelihood of missing test steps. The immediate feedback loop can even trigger instant defect fixes (which still need to be reported appropriately), especially in developer-tester pairs.

Pair testing is particularly valuable when collaboration, learning, and rapid feedback are critical. In particular, it is helpful in:

- exploratory testing
- testing complex or new features
- cross-functional testing (e.g., UI + API)
- mentoring junior testers or cross-training team members

Pair testing is not without its challenges. It is time-consuming and effort-intensive, as two people work on the same task, which can seem less efficient than working separately. Also, testers may have different working styles, levels of assertiveness, or ways of thinking. This can be a benefit, but it can also lead to friction, discomfort, and reduced productivity. One person may dominate the session (e.g., always driving or making decisions), while constant talking, thinking aloud, and justifying decisions can be mentally exhausting.

To overcome these challenges:

- pair testing should be used strategically – e.g., for complex or high-risk features only
- open communication and active listening should be practiced
- roles should be clearly defined (driver and navigator) and switched regularly to avoid fixed patterns of behavior
- clear goals should be set before starting each pair testing session

5.2.3 Vibe Testing

Vibe testing is an emerging, informal AI-assisted test approach that has yet to be fully standardized. It has developed alongside the growth of AI-generated code, commonly referred to as "vibe coding." Typically, developers manually write code, create test cases, and follow structured quality control workflows. With vibe coding, however, developers describe the application's required behavior in natural language, typically as prompts to LLMs (large language models), which then generate the code, often reducing the need for detailed human review. This development shift demands a test approach – called vibe testing – to verify that the end product aligns with user intent.

Vibe testing emphasizes intent-first validation, focusing on testing what the application should do rather than merely what it currently does, instead of relying on static test scripts. Rather than meticulously specifying each test case in advance, testers interact with the application as real users would, relying on exploratory testing and AI-generated test cases. Vibe testing accelerates the test process, makes testing more adaptable to rapidly evolving AI-built applications, reduces reliance on manual test writing, and aligns closely with the intuitive, high-level design style of vibe coding. However, it also presents challenges. Since developers may not fully understand or review the underlying code, there is a risk of hidden defects, security vulnerabilities, AI hallucinations, or misunderstood behaviors. Vibe testing mitigates this risk by acting as a safeguard, ensuring that the application delivers the intended functionality, rather than just what the AI interpreted.

For instance, suppose the developer used an LLM to generate code with the prompt “Create a login page with email and password fields. Users should be redirected to the dashboard upon successful login.” The tester uses prompts or manual exploration to test the system's behavior from the user's perspective. The examples of prompts include: "Test login with invalid email format and confirm it doesn't proceed

to dashboard" or "After logging in, confirm the user sees the correct dashboard based on role (e.g., administrator versus regular user)." The original developer prompt did not specify input validation rules, handling incorrect inputs, or role-based dashboard logic. These are common expectations, but unless explicitly stated, the LLM might skip them. Vibe testing aims to eliminate these potential gaps by testing what the application should do in the real world, rather than assuming the LLM got everything right from a vague prompt.

5.3 Test Smells

A test smell is a symptom of a poorly designed test. While it does not always indicate a problem, it suggests potential risks and increased maintenance costs. Just as code smells in software development should be addressed to maintain code quality, test smells should be addressed to maintain an effective and trustworthy test strategy. This syllabus describes only test smells in manual test cases. For automated test cases, see (Aljedaani et al., 2021).

The following test smells are grouped into categories, each with a description of the test smell and a recommended solution:

Dependency and complexity smells

Interdependent Tests

- Smell: One test's success depends on another test executing first or leaving residual data.
- Solution: Make tests independent, self-contained, and executable in any order.

Hidden Dependencies

- Smell: The test case relies on data or configuration changes that are not specified in the preconditions.
- Solution: Make dependencies explicit and controlled.

Call on Me

- Smell: Reliance on extensive calling of other test cases or on passing parameters between them.
- Solution: Avoid nesting calls and chaining test cases; instead, keep tests simple and use clear naming and documentation.

ORacle

- Smell: Using decision logic in the test steps, such as offering alternative actions and multiple expected results.
- Solution: Make test cases deterministic and branch-free by splitting them into separate test cases, moving decision logic into preconditions, and avoiding the use of "or" in expected results.

Expected results smells

No Clear Expected Results

- Smell: Test procedures list actions but do not clearly define pass/fail criteria.

- Solution: Add expected results, using a structured format that keeps expectations visible and traceable. Avoid subjective language and establish traceability to requirements or acceptance criteria.

One More Step

- Smell: Expected results include additional test steps, often using action verbs such as "check", "verify", and "see".
- Solution: Keep actions and expected results clearly separated from test execution by moving verification into its own step, and avoid using action verbs in the expected results.

Test steps smells

Ambiguous Steps

- Smell: Instructions are vague or open to interpretation.
- Solution: Use specific, objective language with concrete terms and data. Clearly define inputs, avoid subjective words, and reference specific UI elements or labels.

Hotstepper

- Smell: The test steps list every action separately, even when the action has no effect.
- Solution: Merge steps that have no specific effect.

Bulk Steps

- Smell: Many actions are included in a single test step.
- Solution: Use preconditions when appropriate and follow the "one action = one step" rule, ensuring corresponding expected results are added for each action.

Level of detail smells

Overly Long or Detailed Test Case

- Smell: Test cases with many test steps touching multiple features or test cases that include the full navigation path before each small action.
- Solution: Break down into smaller test cases, modularize reusable parts, remove repeated navigation details, and use preconditions effectively.

God Test Case

- Smell: A single test case is used to test multiple alternative scenarios or even the whole system.
- Solution: Design small, focused, independent test cases, organized into test suites.

Hardcoded Test Data

- Smell: Each test case explicitly includes all its test data.
- Solution: Use high-level data descriptions where appropriate and use parameters to separate the test data from the test case.

Lack smells

Environment Assumptions

- Smell: The test case assumes specific test environment conditions without documenting them.
- Solution: Explicitly document environment preconditions and setup requirements.

No Cleanup or Teardown

- Smell: The test case does not specify how test data is handled on completion of the test.
- Solution: Define explicit cleanup or teardown procedures.

Excess smells

Invalidation Heaven

- Smell: Each different type of invalid input is handled by a separate test case.
- Solution: Combine the handling of invalid inputs into a single test case by using parameters.

Copy-Paste Duplication

- Smell: Multiple test cases only differ by a few test data values.
- Solution: Refactor into a single, reusable, parameterized test case to prevent duplication, while ensuring traceability to requirements to measure coverage.

Format and language smells

Unstructured Formatting

- Smell: Test steps are not numbered, with no headings.
- Solution: Use templates.

Typos

- Smell: Test cases include spelling mistakes and incomplete words or sentences.
- Solution: Run a spellchecker.

Click-Push-Press

- Smell: The terminology and phrasing used in test cases are inconsistent.
- Solution: Use a glossary of standard expressions.

6 Test Automation and Test Tools - 30 minutes

Keywords

Agile software development, test automation

Learning Objectives:

6.1 Test Automation in Agile Software Development

CTAL-AT-6.1.1 (K2) Distinguish between different test automation approaches applicable to Agile software development

6.2 Test Tools in Agile Software Development

CTAL-AT-6.2.1 (K2) Give examples of test tools helpful in Agile testing

Introduction

Test automation can provide fast, reliable feedback in short test cycles. Test automation complements, but does not replace exploratory testing and manual testing. Test tools aid collaboration, requirement tracking, performance monitoring, and test reporting. Tools that integrate smoothly into Agile software development improve visibility, accelerate feedback, and help the whole team share responsibility for quality.

6.1 Test Automation in Agile Software Development

The decision to automate is driven by both risk and value. Test automation is well-suited to tasks that are repetitive, deterministic, and high-value in terms of risk reduction, especially when failures could impact business-critical functionality. Tests that will never fail provide little value, and offer minimal benefit when automated. Similarly, tests whose behavior changes frequently are poor candidates for test automation because the maintenance costs often outweigh the benefits.

It is considered a best practice to decide during iteration planning or refinement which user stories to automate, and to what extent. This facilitates a more accurate estimate of the time required to achieve full acceptance of the user story, resulting in greater predictability of iteration delivery.

Agile software development adopts different test automation approaches depending on context, team skills, system criticality, and delivery frequency. One common test approach is to automate tests that provide fast feedback and repeatable validation of the most valuable and stable parts of the system. This typically includes component tests, API tests, and a carefully scoped set of end-to-end tests, in accordance with the test pyramid (see: (ISTQB-CTFL, v4.0.1)). Exploratory testing, and one-off testing (i.e., tests performed only once to test a specific test condition at a particular moment) are normally executed manually because they rely on human insight, are driven by emergent behavior, or the maintenance costs would be too high. The same considerations often apply to usability testing.

The timing of test automation is also a key consideration. In a test-first approach, tests are automated early, before or while the code is written. This allows tests to guide development and provide continuous verification. In contrast, automating regression tests is often an activity that evolves during or after the iteration, depending on when functionality stabilizes.

Agile testing integrates test automation throughout the iteration, beginning with build verification in continuous integration, continuing with exploratory testing supported by test automation tools, and concluding with automated regression test suites that confirm system stability before release.

Successful test automation in Agile software development requires a multi-layered strategy that focuses on automating what delivers consistent, valuable feedback, selecting the right tests to automate, and integrating test automation progressively across the software development lifecycle.

The Agile test automation pyramid, introduced by Mike Cohn, promotes a balanced testing strategy by prioritizing fast, low-level unit tests, supporting them with a smaller layer of service and integration tests, and maintaining only a minimal set of end-to-end UI tests, enabling rapid feedback, reduced costs, and greater CI/CD stability.

Other ISTQB syllabi provide more in-depth knowledge on test automation (see: (ISTQB-TAE, v2.0), (ISTQB-TAS, v1.0)).

6.2 Test Tools in Agile Software Development

The test tools used in Agile testing span multiple categories and support both the iterative and collaborative nature of Agile development. Agile teams benefit from tools that facilitate communication, continuous feedback, and test automation across the entire software development lifecycle. Test tool selection should align with the test strategy, test automation layers, and feedback objectives rather than convenience or trend.

A variety of tools support Agile testing activities and collaboration across the team, each serving a distinct purpose within the software development and delivery process:

Task management and tracking tools enable the team to plan, prioritize, and track user stories, test tasks, and defects, supporting transparency and a shared understanding of team progress and risks. These tools are essential for maintaining the backlog, managing defects, and managing test charters.

Communication and information-sharing tools support real-time collaboration, especially in distributed teams. They enhance interaction during refinement sessions, iteration planning, test reviews, and retrospectives. These tools help testers to share insights, discuss defects, and communicate customer feedback quickly across the whole team.

Test design, test implementation, and test execution tools support the creation and execution of both automated tests and manual tests. These include test automation frameworks for unit testing, API testing, and GUI test automation. They also enable specification by example practices like ATDD and BDD, facilitating collaboration among team members on acceptance criteria.

Continuous integration tools enable frequent code integration and rapid failure detection while always keeping software ready for deployment. These tools are essential in Agile software development, where short feedback loops support iterative delivery and help mitigate regression risks early.

Configuration management tools ensure consistency across test environments by managing code, test scripts, and infrastructure changes. These tools are often integrated into CI/CD pipelines to support DevOps and continuous delivery goals.

Exploratory testing tools support session-based test management, test charter tracking, recording test session sheets, and taking screenshots. These tools help capture test conditions and test results with minimal documentation overhead.

Monitoring and analytics tools provide production-level insights by tracking usage patterns, defect rates, and performance metrics. They are especially useful in Agile teams that practice monitoring as testing and those that leverage real user feedback to refine product behavior post-deployment.

AI tools can automatically generate test cases from code changes, requirements, or user behavior patterns; predict build failures using code changes and historical data; analyze user stories or requirements to detect ambiguity, missing acceptance criteria, or inconsistent terminology; and create realistic test data that covers edge cases.

7 References

Standards

IEEE 1044 – Classification for Software Anomalies, 2009. Standard. Institute of Electrical and Electronics Engineers.

ISO 29119-6 Software and systems engineering – Software testing: Part 6: Guidelines for the use of ISO/IEC/IEEE 29119 (all parts) in agile projects, 2021. Technical report. International Organization for Standardization.

ISTQB® Documents

ISTQB-ATLAS, ISTQB®, 2023. *Certified Tester Agile Test Leadership at Scale: Syllabus*. Version 1.0.

ISTQB-CTFL, ISTQB®, 2024. *Certified Tester Foundation Level: Syllabus*. Version 4.0.1.

ISTQB-TAE, ISTQB®, 2024. *Certified Tester Test Automation Engineering: Syllabus*. Version 2.0.

ISTQB-TAS, ISTQB®, 2024. *Certified Tester Test Automation Strategy: Syllabus*. Version 1.0.

Glossary References

References for terminology used in this document:

- Agile Alliance: https://agilealliance.org/agile101/agile-glossary
- IEEE/Pascal for software engineering: https://pascal.computer.org/sev_display/index.action
- IREB-CPRE for Requirements Engineering: https://cpre.ireb.org/en/downloads-and-resources/glossary
- ISTQB® Glossary: https://glossary.istqb.org/

Books

ADZIC, Gojko, 2011. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning.

ANDERSON, D.J.; REINERSTEN, D.G., 2010. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press.

ANDERSON, L.W.; KRATHWOHL, D.R., 2001. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman. ISBN 9780801319037.

BECK, K.; ANDRES, C., 2004. *Extreme Programming Explained: Embrace Change, 2nd Ed.* Addison-Wesley.

BEIZER, Boris, 1990. *Software Testing Techniques*. Van Nostrand Reinhold: Boston MA.

GREAVES, K.; LAING, S., 2019. *Growing Agile: A Coach's Guide to Agile Testing*. Leanpub.

GREGORY, J.; CRISPIN, L., 2019. *Agile Testing Condensed: A Brief Introduction*. Leanpub.

WHITTAKER, J.A., 2009. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Addison-Wesley Professional.

Articles

ALJEDAANI, Wajdi; PERUMA, Anthony; ALJOHANI, Ahmed; ALOTAIBI, Mazen; MKAOUER, Mohamed Wiem; OUNI, Ali; NEWMAN, Christian D.; GHALLAB, Abdullatif; LUDI, Stephanie, 2021. Test Smell Detection Tools: A Systematic Mapping Study, pp. 170–180. Available from DOI: 10.1145/3463274.3463335.

GHAZI, Ahmad Nauman; GARIGAPATI, Ratna Pranathi; PETERSEN, Kai, 2017. Checklists to Support Test Charter Design in Exploratory Testing, pp. 251–258. ISBN 978-3-319-57633-6.

NIELSEN, Jakob, 1994. Enhancing the explanatory power of usability heuristics. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 152–158.

WALKER, R.; CENYDD, L. ap; POP, S.; MILES, H. C.; HUGHES, C. J.; TEAHAN, W. J.; ROBERTS, J. C., 2013. Storyboarding for visual analytics. *Information Visualization*. Vol. 14, pp. 27–50.

Web Pages

Agile Alliance, [n.d.]. Available also from: https://agilealliance.org/agile101/agile-glossary.

BECK, K.; BEEDLE, M.; BENNEKUM, A. van; COCKBURN, A.; CUNNINGHAM, W.; FOWLER, M.; GRENNING, J.; HIGHSMITH, J.; HUNT, A.; JEFFRIES, R.; KERN, J.; MARICK, B.; MARTIN, R.; MELLOR, S.; SCHWABER, K.; SUTHERLAND, J.; THOMAS, D., 2001. *Agile Manifesto* [online]. [visited on 2025-08-30]. Available from: https://agilemanifesto.org/.

BOLTON, M., 2012. *FEW HICCUPS* [online]. [visited on 2025-07-27]. Available from: https://developsense.com/blog/2012/07/few-hiccupps.

BOLTON, M., 2014. *How Models Change* [online]. [visited on 2025-07-27]. Available from: https://developsense.com/blog/2014/07/how-models-change.

GAREEV, A., 2019. *The TERMS for Test Automation* [online]. [visited on 2025-07-27]. Available from: https://www.automation-beyond.com/2019/04/10/the-terms-for-test-automation.

GREGORY, J., 2021. *Testing from a holistic point of view* [online]. [visited on 2025-09-05]. Available from: https://janetgregory.ca/testing-from-a-holistic-point-of-view/.

IEEE/Pascal for software engineering, [n.d.]. Available also from: https://pascal.computer.org/sev_display/index.action.

IREB-CPRE for Requirements Engineering, [n.d.]. Available also from: https://cpre.ireb.org/en/downloads-and-resources/glossary.

JOHNSON, K., 2012. *Software Testing Heuristics and Mnemonics* [online]. [visited on 2025-07-27]. Available from: https://www.slideshare.net/slideshow/kn-johnson-2012-heuristics-mnemonics/15019733.

KOHL, J., 2010. *Test Mobile Applications with I SLICED UP FUN!* [online]. [visited on 2025-07-27]. Available from: https://www.kohl.ca/articles/ISLICEDUPFUN.pdf.

OWASP, 2025. *Top Ten Web Application Security Risks* [online]. 2025-07-27. [visited on 2025-07-27]. Available from: https://owasp.org/www-project-top-ten.

PROSCI, 2026. *The Prosci ADKAR Model* [online]. 2026-01-04. [visited on 2026-01-04]. Available from: https://www.prosci.com/methodology/adkar.

SUTHERLAND, J.; SCHWABER, K., 2020. *Scrum Guide* [online]. [visited on 2025-08-30]. Available from: https://scrumguides.org/index.html.

WYNNE, M., 2015. *Introducing Example Mapping* [online]. [visited on 2025-10-07]. Available from: https://cucumber.io/blog/bdd/example-mapping-introduction.

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

8 Further Reading

GREGORY, J.; CRISPIN, L., 2024. *Holistic Testing: Weave Quality into Your Product*. Leanpub.

HENDRICKSON, Elisabeth, 2013. *Explore It!* Pragmatic Bookshelf.

ISO 29119-1 Software and systems engineering – Software testing: Part 1: General concepts, 2022. Standard. International Organization for Standardization.

ISO/IEC 20246: Software and systems engineering – Work product reviews, 2017. Standard. International Organization for Standardization.

ISO/IEC 25010: Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model, 2023. Standard. International Organization for Standardization.

ISO/IEC 25019: Systems and software Quality Requirements and Evaluation (SQuaRE) – Quality-in-use model, 2023. Standard. International Organization for Standardization.

ISO/IEC/IEEE 29119-3: Software testing, Part 3: Test documentation, 2021. Standard. International Organization for Standardization.

ISO/IEC/IEEE 29119-4: Software testing, Part 4: Test techniques, 2021. Standard. International Organization for Standardization.

ISO/IEC/IEEE 29119-5: Software testing, Part 5: Keyword-driven testing, 2016. Standard. International Organization for Standardization.

ISTQB-AI, ISTQB®, 2021. *Certified Tester AI Testing: Syllabus*. Version 1.0.

ISTQB-ITP, ISTQB®, 2011. *Certified Tester Expert Level Improving the Test Process: Syllabus*. Version 1.0.

ISTQB-MBT, ISTQB®, 2024. *Certified Tester Model-Based Tester: Syllabus*. Version 1.1.

ISTQB-PT, ISTQB®, 2018. *Certified Tester Performance Testing: Syllabus*. Version 1.0.

ISTQB-TM, ISTQB®, 2024. *Certified Tester Advanced Level Test Management: Syllabus*. Version 3.0.

ISTQB-TTA, ISTQB®, 2021. *Certified Tester Advanced Level Technical Test Analyst: Syllabus*. Version 4.0.

ISTQB-UT, ISTQB®, 2018. *Certified Tester Usability Testing: Syllabus*. Version 1.0.

9 Appendix A – List of Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
API	Application Programming Interface
ATDD	Acceptance Test-Driven Development
BDD	Behavior-Driven Development
BPMN	Business Process Model and Notation
CI/CD	Continuous Integration/Continuous Delivery
GQM	Goal-Question-Metric
LLM	Large Language Model
MTTR	Mean Time to Recovery
OWASP	Open Web Application Security Project
TDD	Test-Driven Development
UI	User Interface
XP	eXtreme Programming

10 Appendix B – Domain Specific Terms

Term Name	Definition
bias	A systematic deviation from objectivity that may affect test objectives, test activities, or test results.
canary release	A deployment strategy in which a new version of a system is released to a limited subset of users to evaluate it before a wider rollout to all users.
community of practice	A group of professionals who share a common interest, passion, or challenge, meet regularly to share knowledge, learn from each other's experiences, and improve skills to solve problems and innovate in their specific domain.
dark launch	A technique in which new or changed functionality is deployed to the production environment without being visible or accessible to end users, in order to validate behavior, performance, or risk under real conditions before full release.
Definition of Done	A formal description of the state of the increment when it meets the quality measures required for the product.
epic	A description of a stakeholder need which is typically larger than what can be implemented in a single iteration.
example mapping	A collaborative technique in Agile software development that uses rules, examples, and questions to clarify requirements and support test design.
feature toggle	A mechanism to modify system behavior without changing code.
FEW HICCUPPS	A test heuristic for Familiarity, Explainability, World, History, Image, Comparable products, Claims, Users' desires, Product, Purpose, Statutes, to support the identification of test oracles.
Goal-Question-Metric	A measurement approach that defines goals, derives questions to assess the achievement of those goals, and identifies metrics to answer the questions in a systematic and traceable manner.

Term Name	Definition
hardening iteration	An iteration, typically conducted at the end of a series of development iterations, in which the primary focus is on stabilizing the product by addressing outstanding defects, improving performance, completing integration and system testing activities, and preparing the product for release.
I SLICED UP FUN	A test heuristic for Inputs, Store, Location, Interactions/Interruptions, Communication, Ergonomics, Data, Usability, Platform, Function, User scenarios, Network.
iteration	A fixed-length, time-boxed cycle in which a set of planned activities is performed to produce a potentially releasable version of a product.
metric	A measurement scale and method used for measurement.
RCRCRC	A test heuristic for Recent, Core, Risky, Configuration, Repaired, Chronic.
risk-storming	A structured brainstorming technique in which stakeholders collaboratively identify potential risks in order to support risk-based testing.
SFDIPOT	A test heuristic for Structure, Function, Data, Interfaces, Platform, Operations, Time.
specification by example	A development technique in which the specification is defined by examples.
storyboarding	A visual representation of a sequence of user interactions or system behaviors, used to clarify requirements, support test design, and communicate scenarios effectively.
structured conversation	A facilitated, purposeful discussion that follows a defined structure or set of prompts to ensure shared understanding, focus on objectives, and effective exchange of information among stakeholders.
TERMS	A test heuristic for Tools and Technology, Execution, Requirements and Risks, Maintenance, Security.
test session sheet	A document used in exploratory testing to record the execution and outcomes of a time-boxed test session, including notes on test activities performed, observations, test coverage achieved, defects found, and follow-up items.

Term Name	Definition
user journey	A representation of the sequence of steps a user performs to achieve a specific goal when interacting with a system, including the user's experience, perceptions, and emotional responses at each step.
user story	A short narrative describing a need from a user's perspective together with the expected benefit when this need is satisfied.
user story slicing	The practice of breaking down a large or complex user story into smaller, manageable, and testable user stories.

11 Appendix C – Learning Objectives/Cognitive Level of Knowledge

The specific learning objectives applying to this syllabus are shown at the beginning of each chapter. Each topic in the syllabus will be examined according to the learning objective for it.

The learning objectives begin with an action verb corresponding to its cognitive level of knowledge as listed below.

Level 1: Remember (K1)

The candidate will remember, recognize and recall a term or concept.

Action verbs: Recall, recognize.

Examples
Recall the concepts of the test pyramid.
Recognize the typical objectives of testing.

Level 2: Understand (K2)

The candidate can select the reasons or explanations for statements related to the topic, and can summarize, compare, classify and give examples for the testing concept.

Action verbs: Classify, compare, differentiate, distinguish, explain, give examples, interpret, summarize

Examples	Notes
Classify test tools according to their purpose and the test activities they support.	
Compare the different test levels.	Can be used to look for similarities, differences or both.
Differentiate testing from debugging.	Looks for differences between concepts.
Distinguish between project and product risks.	Allows two (or more) concepts to be separately classified.
Explain the impact of context on the test process.	
Give examples of why testing is necessary.	
Infer the root cause of defects from a given profile of failures.	
Summarize the activities of the work product review process.	

Level 3: Apply (K3)

The candidate can carry out a procedure when confronted with a familiar task, or select the correct procedure and apply it to a given context.

Action verbs: Apply, implement, prepare, use

Examples	Notes
Apply boundary value analysis to derive test cases from given requirements.	Should refer to a procedure / technique / process etc.
Implement metrics collection methods to support technical and management requirements.	
Prepare installability tests for mobile apps.	
Use traceability to monitor test progress for completeness and consistency with the test objectives, test strategy, and test plan.	Could be used in an LO that wants the candidate to be able to use a technique or procedure. Similar to 'apply'.

Level 4: Analyze (K4)

The candidate can separate information related to a procedure or technique into its constituent parts for better understanding and can distinguish between facts and inferences. Typical application is to analyze a document, software or project situation and propose appropriate actions to solve a problem or task.

Action verbs: Analyze, deconstruct, outline, prioritize, select.

Examples	Notes
Analyze a given project situation to determine which black-box or experience-based test techniques should be applied to achieve specific goals.	Examinable only in combination with a measurable goal of the analysis. Should be of the form 'Analyze xxxx to xxxx' (or similar).
Prioritize test cases in a given test suite for execution based on the related product risks.	
Select the appropriate test levels and test types to verify a given set of requirements.	Needed where the selection requires analysis.

Level 5: Evaluate (K5)

The candidate may make judgments based on criteria and standards. He detects inconsistencies or fallacies within a process or product, determines whether a process or product has internal consistency and detects the effectiveness of a procedure as it is being implemented (e.g., determine if a scientist's conclusions follow from observed data.)

Action verbs: Assess, critique, evaluate, recommend

Examples	Notes
Assess a test organization using either TPI Next or TMMi.	
Critique the appropriateness of the test activities in an organization for the given context.	
Evaluate an organization to determine the options for proper placement of the test team.	Should be restricted to the evaluation, not including the resulting solution (which would be K6)
Recommend measures to create acceptance of the changes by the people involved.	

Level 6: Create (K6)

The candidate puts elements together to form a coherent or functional whole. Typical application is to reorganize elements into a new pattern or structure, devise a procedure for accomplishing some task, or invent a product (e.g., build habitats for a specific purpose).

Action verbs: Create, design, develop, plan

Examples	Notes
Create a test improvement plan considering change management issues with appropriate steps and actions.	
Design an organizational structure for a given scope of a test process improvement program	
Develop a defect management process for a testing organization, including the defect report workflow and communication	
Plan and perform assessment interviews using a particular process or content-based model.	

Cognitive levels of learning objectives are based on (L. Anderson et al., 2001).

12 Appendix D – Business Outcomes traceability matrix with Learning Objectives

This section lists the traceability between the Business Outcomes and the Learning Objectives of Advanced Level Agile Tester.

Business Outcomes: Advanced Level Agile Tester		CTAL-AT-BO1	CTAL-AT-BO2	CTAL-AT-BO3	CTAL-AT-BO4	CTAL-AT-BO5	CTAL-AT-BO6	CTAL-AT-BO7	CTAL-AT-BO8
CTAL-AT-BO1	Collaborate in cross-functional teams, being familiar with principles and basic practices of Agile software development	6							
CTAL-AT-BO2	Adapt existing testing experience and knowledge to Agile values and principles		8						
CTAL-AT-BO3	Support the Agile team in test planning			6					
CTAL-AT-BO4	Apply relevant Agile software development approaches and test techniques to ensure tests that provide adequate coverage				10				
CTAL-AT-BO5	Assist business stakeholders in defining understandable and testable user stories, scenarios, requirements, and acceptance criteria, as appropriate					6			
CTAL-AT-BO6	Create and implement various Agile software development test approaches						10		
CTAL-AT-BO7	Support and contribute to test automation in an Agile software development project							2	
CTAL-AT-BO8	Work with - and share information with - other team members using effective communication styles and channels								9

Business Outcomes: Advanced Level Agile Tester		CTAL-AT-BO1	CTAL-AT-BO2	CTAL-AT-BO3	CTAL-AT-BO4	CTAL-AT-BO5	CTAL-AT-BO6	CTAL-AT-BO7	CTAL-AT-BO8
LO Number	Learning Objective (K-Level)								
1	Test Strategy and Test Approach Challenges - 60 minutes								
1.1	Test Types								
CTAL-AT-1.1.1	Compare test types to be performed during and after an iteration (K2)		X	X					
1.2	End-to-End Testing								
CTAL-AT-1.2.1	Explain when end-to-end testing should be performed (K2)		X	X					
1.3	Formal Testing and Holistic Testing								
CTAL-AT-1.3.1	Compare the benefits and drawbacks of formal testing and holistic testing (K2)		X						
1.4	Regression Test Approaches								
CTAL-AT-1.4.1	Differentiate among regression test approaches (K2)		X	X			X		
2	People and Teams - 60 minutes								
2.1	Whole Team Approach								
CTAL-AT-2.1.1	Compare generalization and specialization within a team (K2)	X							X
CTAL-AT-2.1.2	Give examples of how to motivate business representatives to perform test activities (K2)	X							X
CTAL-AT-2.1.3	Summarize how the whole team approach can assist the development team (K2)	X							X
2.2	Tissue Testers								
CTAL-AT-2.2.1	Explain how and when to use tissue testers (K2)				X				

Business Outcomes: Advanced Level Agile Tester		CTAL-AT-BO1	CTAL-AT-BO2	CTAL-AT-BO3	CTAL-AT-BO4	CTAL-AT-BO5	CTAL-AT-BO6	CTAL-AT-BO7	CTAL-AT-BO8
3	Test Management and Test Process Improvement - 210 minutes								
3.1	Test Planning								
CTAL-AT-3.1.1	Summarize how to perform test planning in Agile software development (K2)	X	X	X					
CTAL-AT-3.1.2	Outline a project test strategy using testing quadrants (K4)		X				X		
3.2	Test Monitoring and Test Control								
CTAL-AT-3.2.1	Explain how to perform test monitoring and test control in Agile software development (K2)			X					
3.3	Test Reporting								
CTAL-AT-3.3.1	Compare the different types of coverage that can be used for test reporting in Agile software development (K2)		X		X				
3.4	Test Process Improvement								
CTAL-AT-3.4.1	Select appropriate test process improvement measures based on metrics in Agile software development (K4)	X	X						X
CTAL-AT-3.4.2	Explain how to perform test process improvement in Agile software development (K2)	X		X					X
4	Shift Left - 135 minutes								
4.1	Using Shift Left to Improve Test Basis Quality								
CTAL-AT-4.1.1	Give examples of how testware can be used as a form of requirements (K2)					X			X
CTAL-AT-4.1.2	Explain how storyboarding and testboarding can be used to increase the quality of the test basis (K2)					X			X

Business Outcomes: Advanced Level Agile Tester		CTAL-AT-BO1	CTAL-AT-BO2	CTAL-AT-BO3	CTAL-AT-BO4	CTAL-AT-BO5	CTAL-AT-BO6	CTAL-AT-BO7	CTAL-AT-BO8
CTAL-AT-4.1.3	Explain how example mapping can be used to increase the quality of the test basis (K2)					X			X
CTAL-AT-4.1.4	Give examples of how biases can negatively affect the product quality (K2)					X			
CTAL-AT-4.1.5	Apply user story slicing to achieve testable user stories (K3)					X			
4.2	Shift Left and Requirements Engineering								
CTAL-AT-4.2.1	Explain how requirements engineering supports shift left (K2)					X			
5	Agile Approaches and Test Techniques - 285 minutes								
5.1	Exploratory Testing								
CTAL-AT-5.1.1	Explain test heuristics (K2)				X		X		
CTAL-AT-5.1.2	Give examples of test mnemonics related to testing in Agile software development (K2)				X		X		
CTAL-AT-5.1.3	Explain test tours (K2)				X		X		
CTAL-AT-5.1.4	Analyze user stories and epics to create test charters (K4)								X
CTAL-AT-5.1.5	Apply exploratory testing to support testing in Agile software development (K3)				X		X		
5.2	Assisted Testing								
CTAL-AT-5.2.1	Explain mob testing (K2)				X		X		
CTAL-AT-5.2.2	Explain pair testing (K2)				X		X		
CTAL-AT-5.2.3	Explain vibe testing (K2)				X		X		
5.3	Test Smells								

Business Outcomes: Advanced Level Agile Tester		CTAL-AT-BO1	CTAL-AT-BO2	CTAL-AT-BO3	CTAL-AT-BO4	CTAL-AT-BO5	CTAL-AT-BO6	CTAL-AT-BO7	CTAL-AT-BO8
CTAL-AT-5.3.1	Use test smells to evaluate the quality of test cases (K3)				X		X		
6	Test Automation and Test Tools - 30 minutes								
6.1	Test Automation in Agile Software Development								
CTAL-AT-6.1.1	Distinguish between different test automation approaches applicable to Agile software development (K2)							X	
6.2	Test Tools in Agile Software Development								
CTAL-AT-6.2.1	Give examples of test tools helpful in Agile testing (K2)							X	

13 Appendix E – Release Notes

This is a major version of the syllabus, so detailed changes from the previous version are not provided. Instead, as the difference between v1.0 and v2.0 is significant, the general nature and reason for the changes are described below.

Since Agile is now a standard method of software development, the basics of Agile are no longer described in the learning objectives. Instead, these basics are covered in Chapter 1: Introduction to Agile.

Part of the content from the Agile Tester v1.0 syllabus has been moved to the Foundation Level v4.0 syllabus. Therefore, that content is not covered in the current Agile Tester v2.0 syllabus. This applies to the test-first approach, test pyramid, Agile testing quadrants, independent testing, collaborative user story creation, retrospectives, and test effort estimation. Some of this content is extended in this syllabus (e.g. Agile testing quadrants on a K4 level).

New topics have been introduced (such as mnemonics, heuristics, test tours, biases, coverage metrics for reporting, holistic testing, tissue testers, mob testing, vibe testing, story boarding, and example mapping), and some of the content discussed in the Foundation Level 4.0 syllabus has been expanded upon (e.g., exploratory testing and the whole team approach).

14 Trademarks

ISTQB® is a registered trademark of International Software Testing Qualifications Board TMMi® is a registered trademark of TMMi Foundation. TPI-Next® is a registered trademarks of Sogeti, The Netherlands.

15 Index