ISTQB AGILE TESTER

"One for all, all for one"

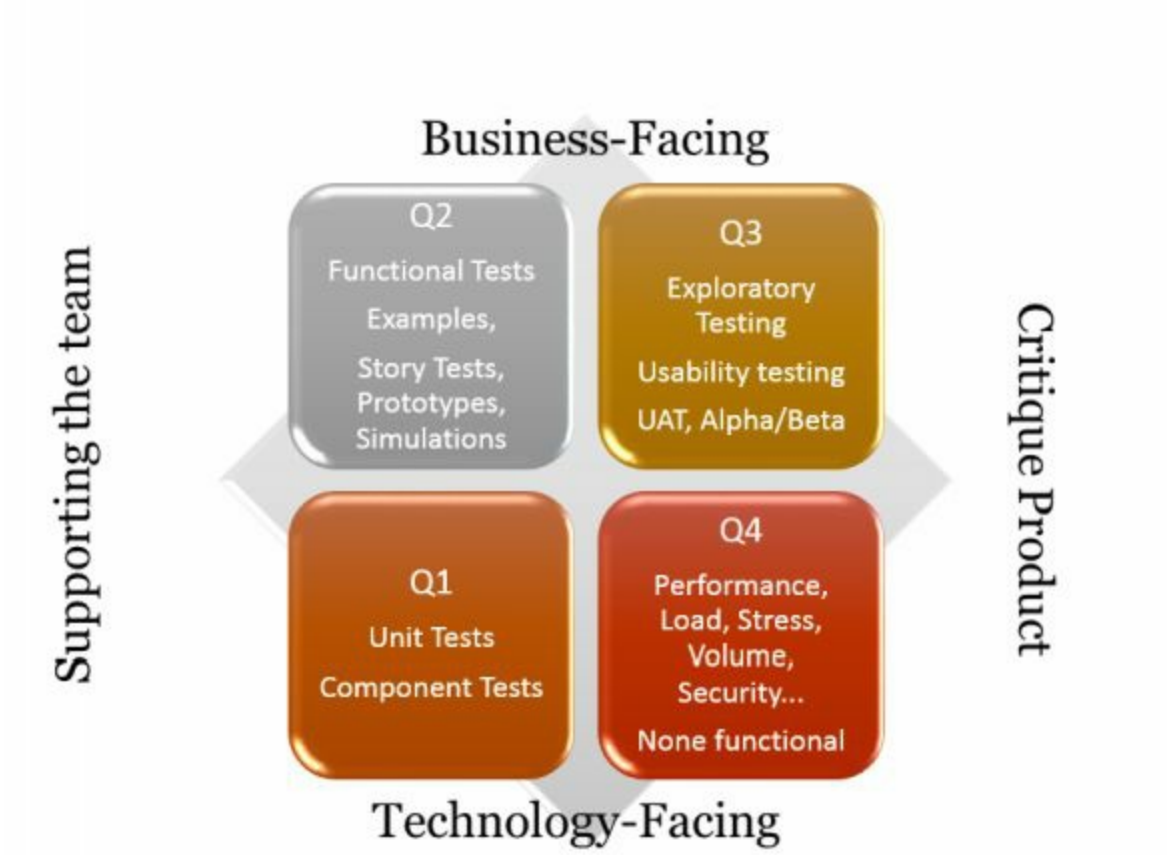
STEEN LERCHE-JENSEN

Introduction to ISTQB Agile Tester

This book will go through the basic of the ISTQB Agile Test Foundation materials. This eBook goes through the full learning materials and make you ready for certification.

- Agile Software Development
- Fundamental Agile Testing Principles, Practices and Processes
- Agile Testing Methods, Techniques, and Tools

The bases for this book is the Syllabus you can find on www.istqb.org. ISTQB is the organization, which administrate all ISTQB education schema and certifications worldwide.



Here you see the Agile Testing Quadrant [Brian Marick] and this is a good foundation to get your mind-set ready for agile testing. We go through the quadrant later.

The ISTQB syllabus have five purpose:

1. For translation in each member countries by their national board.
2. For the exam boards. Helps then create good exams question
3. For us, training providers, so we can produce good courses and books like this.
4. For you, the super developer, tester, business analyst who want to study agile testing.
5. For the whole software development industry to advance the profession of software testing

About the author:

Steen Lerche-Jensen is accredited ISTQB trainer, started within in software development in1989. First, he worked 11 years as developers and the last 15 years as freelance test manager worldwide.

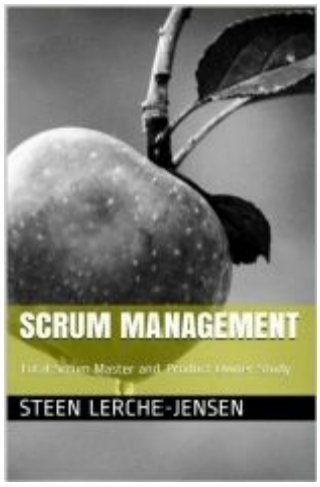
In his spare time, he is president for www.scrum.as, which is an open education program: Scrum Association was started by a LinkedIn group containing more than 3300 members and is growing rapidly. The Open education program is about securing updated and high quality course content. All members can suggest improvement of each part of a course, and course materials are therefor following the fast changing world we are living in.

On www.scrum.as you will also be able to prove your skills and get diploma. Now we offer the following certification:

- International Scrum Master Foundation
- International Product owner Foundation
- International Agile Tester Foundation

Keep an eye on the site, more courses and certifications will be available soon.

Also wrote:



Content

1 Agile Software Development

Les us get started. We will do that by going through the basic of the most common agile methods. Often we will use Scrum to illustrate some points, but other agile methods can of course also be used. Agile is about being good to adapt to the every changing surroundings.

1.1 The Fundamentals of Agile Software Development

You need a mind-set change if you have only worked as tester on a traditional project. Working on agile projects is different and as testers, we need to understand the agile values and principles that follows normal agile methods.

The whole-team approach where developer, tester and domain expert collaborate for a common goal is the heart of agile development, just like "The Three Musketeers". Their motto fits well together with agile mind-set "all for one, one for all" – meaning we do this together. If one fails, we all fail, and we share success together.

In the Agile methods each iteration involves a team working through a full software development cycle, including planning, requirements analysis, design, coding, unit testing, and acceptance testing when a working product is demonstrated to stakeholders. Therefore, working on an agile project/team has differences:

- Testers on agile projects work differently than on traditional projects
- Certain values and principles underpin agile projects
- Testers work as part of a whole-team approach
- Testers collaborate with
    - Developers
    - Business representatives
    - Other stakeholders
- Testers provide early and frequent feedback on quality
- Early feedback provides multiple benefits, as we will see…

1.1.1 Agile Software Development and the Agile Manifesto

Most agile methods are built upon the Agile Manifesto published in 2001 [agilemanifesto.org]. The people behind the manifesto writes - We are uncovering better ways of developing software by doing it and helping others do it. Through this work, we have come to value:

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

Note that these are statements of emphasis, not rejection of traditional concepts

Individuals and Interactions: The first core value of the Agile Manifesto is to value individuals and

interactions over processes and tools. When you allow each person to contribute unique value to your software development project, the result can be powerful.

Working Software: A software development team's focus should be on producing working products. The second Agile core value emphasizes working software over comprehensive documentation.

On projects using agile management tools, the only way to measure whether you are truly done with a product requirement is to produce the working product feature associated with that requirement. For software products, working software means the software meets what has called the definition of done: at the very least, developed, tested, integrated, and documented. After all, the working product is the reason for the project.

Customer Collaboration: Agile management principles extend to your relationship with the customer. Agile's third core value emphasizes customer collaboration. The agile pioneers understood that collaboration, rather than confrontation, produces better, leaner, more useful products. Because of this understanding, agile methodologies make the customer part of the project on an ongoing basis.

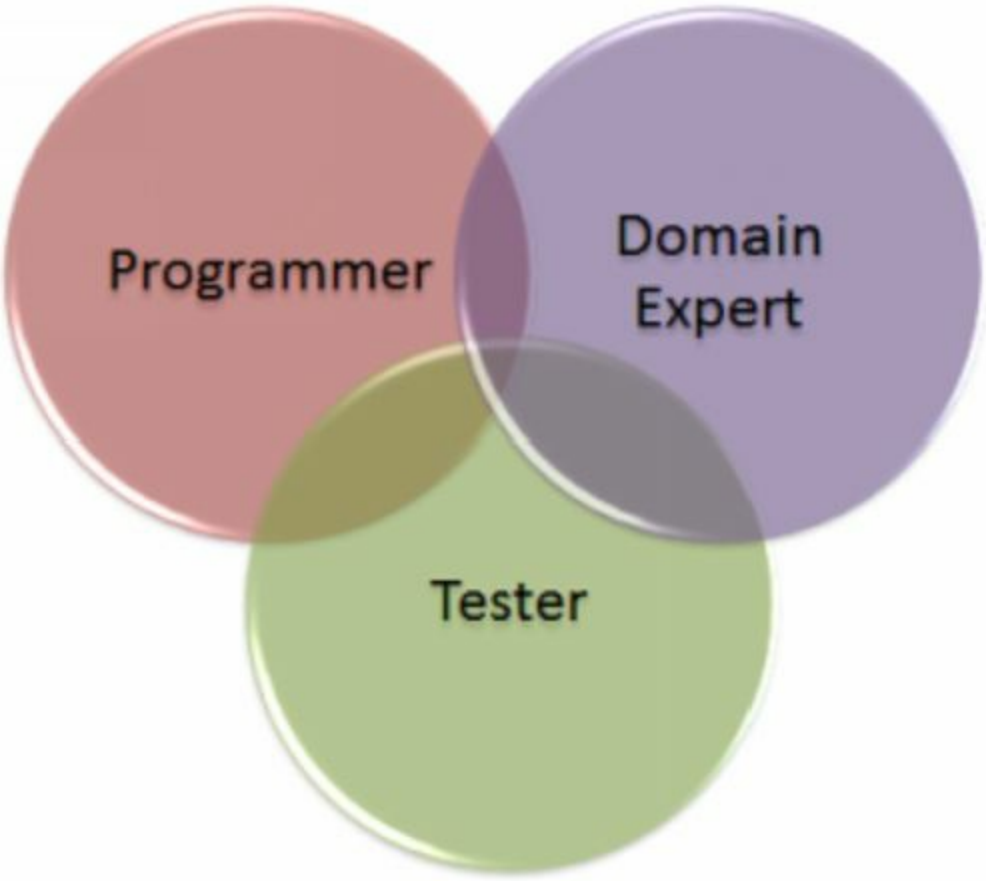
Responding to Change: The fourth core value of agile project management addresses the response to change. Change is a valuable tool for creating great products. Using agile management principles, project teams that can respond quickly to customers, product users, and the market in general are able to develop relevant, helpful products that people want to use.

Agile Principles

The 12 Agile Principles are a set of guiding concepts that support project teams in implementing agile projects. Use these concepts to implement agile methodologies in your projects.

1. Satisfy customers via early, continuous, valuable delivery
2. Welcome changing requirements, even late in project
3. Deliver working software frequently
4. Daily whole-team collaboration
5. Build projects around motivated individuals
6. Use face-to-face conversations where possible
7. Working software is the primary measure of progress
8. Promote sustainable development
9. Emphasize technical excellence and good design
10. Emphasize simplicity (amount of work not done)
11. Self-organizing teams maximize quality
12. Become more effective via regular retrospectives

1.1.2 The Whole-team Approach

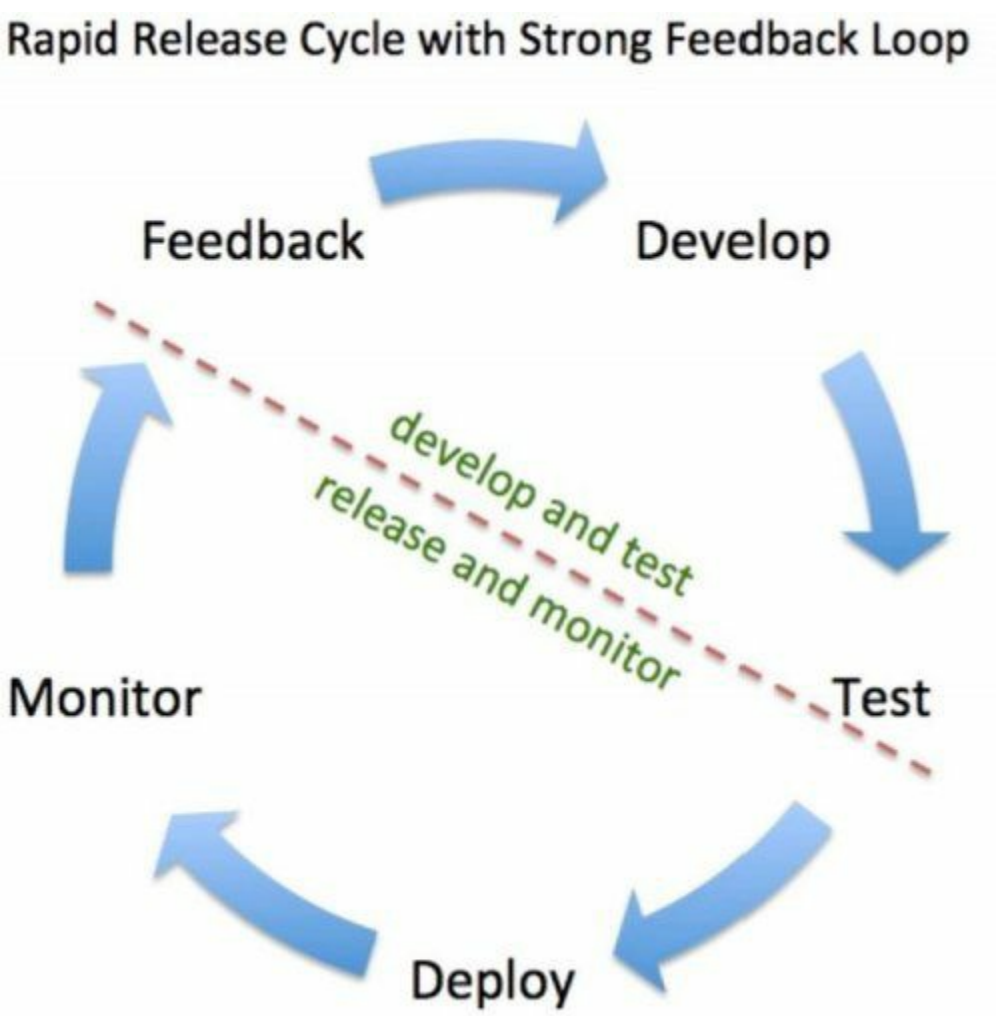


Whole-team approach, also called team-based approach, is a strategy for project management in which everyone on the project team is held equally responsible for the quality and success of the project. Remember "The Three Musketeers" and their motto all for one, one for all".

- Involve everyone with necessary knowledge, skills
- Five to nine people (including business stakeholders)
- Ideally collocated
- Daily stand-up meetings
- Everybody are responsible for quality; testers collaborate:
    - With business representatives on acceptance tests
    - With developers on the testing strategy and test automation
- The whole-team approach (power of three):
    - Transfer knowledge within the team
    - Increase communication and collaboration
    - Avoid of unnecessary documentation
    - Leverage everyone's skills

1.1.3 Early and Frequent Feedback

Frequent feedback is vital for agile development teams to understand whether the team is going in the direction as expected. The product increment developed by agile team will be subjected to stakeholder review and any feedback may be appended to the product backlog, which can prioritized at the discretion of the Product Owner. One of the best ways to provide frequent feedback is through Continuous Integration (discussed later).



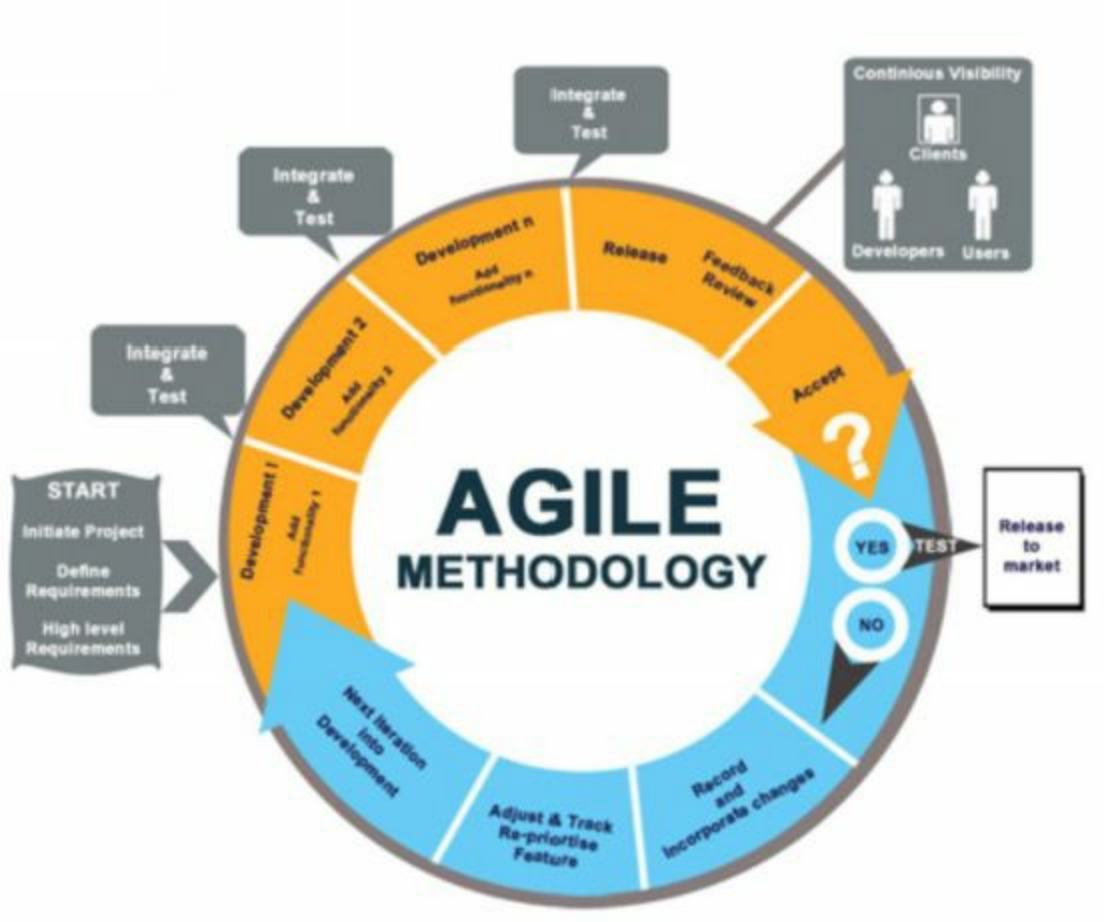
- Short iterations lead to frequent feedback on quality from testers and customers
- Customer feedback focuses attention on business value and risk
- Benefits:
    - Reduced requirements misunderstandings
    - Early and ongoing clarification of customer needs
    - Early discovery and resolution of bugs
    - Understanding of velocity and quality capability
- Agile projects can enjoy steady progress

1.2 Aspects of Agile Approaches

Let us have a look at some of the agile approaches like:

- Collaborative user story creation
- Retrospective
- Continuous integration
- Iteration and release planning

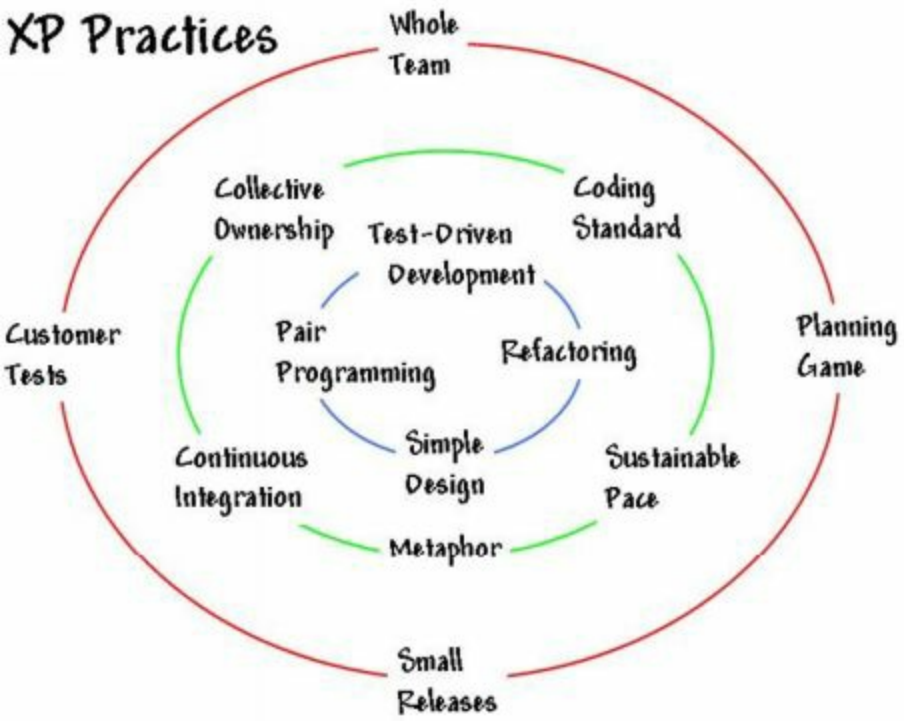
1.2.1 Aspects of agile approaches



- Agile methods are not unified; there is diversity
- Each method implements the Agile Manifesto differently
- In this course, we consider Extreme Programming, Scrum, and Kanban as popular, representative methods
- There are common practices across these methods, which we'll examine

Extreme programming (XP)

XP is a software development methodology, which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.



Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed. It also include a flat management structure, simplicity and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers.

The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels. As an example, Code reviews are considered a beneficial practice; taken to the extreme, code can be reviewed *continuously*, i.e. the practice of Pair programming.

Some key points;

- Values: communication, simplicity, feedback, courage, respect
- Principles: humanity, economics, mutual benefit, self-similarity, improvement, diversity, reflection, flow, opportunity, redundancy, failure, quality, baby steps, accepted responsibility
- Primary practices: sit together, whole team, informative workspace, energized work, pair programming, stories, weekly cycle, quarterly cycle, slack, ten-minute build, continuous integration, test first programming, incremental design.

- Many other agile practices use some aspects of XP

Scrum

Scrum is a lightweight agile project management framework mainly used for software development. It describes an iterative and incremental approach for project work.

Overview of Scrum Framework

Scrum can be used in all kinds of software development: for developing complete software packages, for developing only some parts of bigger systems, for customer or internal projects.

The Scrum Framework itself is very simple. It defines only some general guidelines with only a few rules, roles, artefacts and events. Nevertheless, each of these components is important, serves a specific purpose and is essential for a successful usage of the framework.

The Main Components of Scrum Framework are:

- The three roles: Scrum Master, Scrum Product Owner and the Scrum Team
- A prioritized Backlog containing the end user requirements
- Sprints
- Scrum Events: Sprint Planning Meeting (WHAT-Meeting, HOW-Meeting), Daily Scrum Meeting, Sprint Review Meeting, Sprint Retrospective Meeting

Important in all Scrum projects are self-organization and communication within the team. There is no longer a project manager in a classical sense. In the Scrum Framework, the Scrum Master and the Scrum Product Owner share his responsibilities. However, in the end the team decides what and how much they can do in a given project iteration (Sprint).

Another central aspect within the Scrum Framework is continuous improvement: inspect & adapt. The Scrum Teams have to frequently inspect and assess their created artefacts and processes in order to

adapt and optimize them. In the midterm this will optimize the results, increases predictably and therefore minimize overall project risk.

The Scrum Framework tries to deal with the fact that the requirements are likely to change quickly or are not completely known at the start of the project. The low-level requirements are only defined at the time when they are going to be really implemented. In Scrum, changes and optimizations of product, requirements and processes are an integral part of the completely engineering cycle.

Another cornerstone of the Scrum Framework is communication. The Scrum Product Owner works closely with the Scrum Team to identify and prioritize functionality. This functionality is written down in user stories and stored in a Scrum Product Backlog. The Product Backlog consists everything that needs to be done in order to successfully deliver a working software system.

The Scrum Team is empowered to only select the user stories they are sure they can finish within the 2-4 weeks of Sprints. As the Scrum Team is allowed to commit their own goals, they will be more motivated and work with best possible performance. The Scrum Master is another important role in the Scrum Framework as it works as a servant-master with the Scrum Team. His/her main tasks are to make the Scrum team understand how Scrum operates, to protect the Scrum Team from external interruptions and to remove impediments that hinder the Scrum Team to reach its maximum productivity.

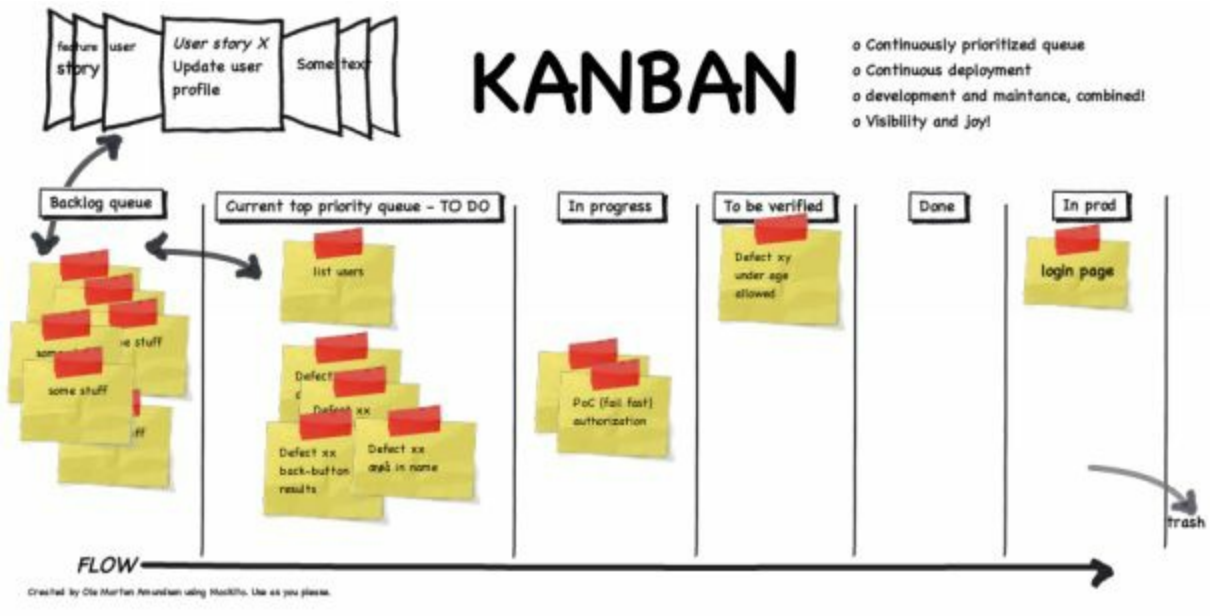
The Scrum Framework in its simple form is best used for smaller, one-team projects. However, with the introduction of additional roles like the "Chief Scrum Product Owner" it is also usable in bigger multi-teams and/or distributed-team projects.

Kanban

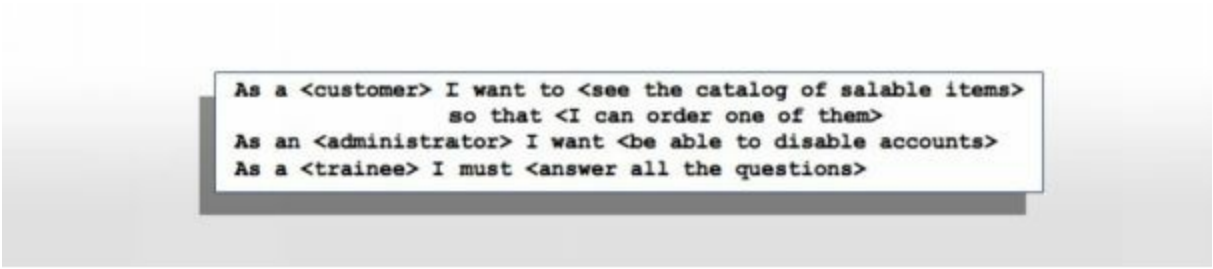
Kanban is a method for managing knowledge work with an emphasis on just-in-time delivery while not overloading the team members. In this approach, the process, from definition of a task to its delivery to the customer, is displayed for participants to see and team members pull work from a queue.

- Optimize flow of work in value-added chain
- Instruments:
    - Kanban board
    - Work-in-progress limit
    - Lead time
- Both Kanban and Scrum provide status transparency and backlogs, but:
    - Iteration is optional in Kanban
    - Items can be delivered one at a time *or* in a release

- Time boxing is optional



1.2.2 Collaborative User Story Creation



Some facts:

- Developers, testers, and business stakeholders collaborate to capture requirements in user stories
- User stories include:
  - Functional and non-functional elements
  - Acceptance criteria for each element
- Testers bring a unique perspective to this process
  - Identify missing elements
  - Ask open-ended questions
  - Suggest tests for the user story
  - Confirm the acceptance criteria
- Acceptance criteria clarify the feature and establish clear completion measures

Creating User Stories:

User Stories adhere to a specific, predefined structure, and are thus a simplistic way of documenting the requirements, and desired end-user functionality. A User Story tells you three things about the requirement; **Who**, **What**, and **Why**. The requirements expressed in User Stories are short, simple, and easy-to-understand statements. The predefined, standard format results in enhanced communication among the Stakeholders and better estimations by the Team. Some User Stories may be too large to handle within a single Sprint. These large User Stories are often called Epics. Once Epics come up in the Prioritized Product Backlog to be completed in an upcoming Sprint, they should be decomposed into manageable User Stories.

The Prioritized Product Backlog is a dynamic list that is continuously updated because of reprioritization and new, updated, refined, and sometimes, deleted User Stories. These updates to the Backlog are typically the result of changing business requirements.

As the Customer representative conceives a User Story, it is written down on a note card with a name and a brief description. If the developer and the Customer representative find a User Story deficient in some way (too large, complicated, or imprecise), it is rewritten until satisfactory—often using the INVEST guidelines (see Chapter 8.5 below). Commonly, User Stories should not be considered definitive once they have been written down, since requirements tend to change throughout the development lifecycle, which agile processes handles by not carving them in stone upfront.

Format for creation of User Story

A useful format for creating a User Story is the following:
"As a *<role>,* I want *<goal/desire>* so that *<benefit>".*

Here is an example using the format: As a Database Administrator, I should be able to revert a selected number of database updates so that the desired version of the database is restored.

User Story Acceptance Criteria

Every User Story has associated Acceptance Criteria. User Stories are subjective, so the Acceptance Criteria provide the objectivity required for the User Story to be considered as Done or Not Done during the Sprint Review. Acceptance Criteria provide clarity to the Team on what is expected of a User Story, remove ambiguity from requirements, and help in aligning expectations. The Product Owner defines and communicates the Acceptance Criteria to the Scrum Team.

In the Sprint Review Meetings, the Acceptance Criteria provide the context for the Product Owner to decide if a User Story has been completed satisfactorily. It is important and the responsibility of the Scrum Master to ensure that the Product Owner does not change the Acceptance Criteria of a committed User Story in the middle of a Sprint.

INVEST criteria for User Stories

The INVEST system features the following characteristics.

Letter	Meaning	Description

I	Independent	The User Story should be self-contained, in a way that there is no inherent dependency on another User Story.
N	Negotiable	User Stories, up until they are part of an iteration, can always be changed and rewritten.
V	Valuable	A User Story must deliver value to the end user.
E	Estimable	You must always be able to estimate the size of a User Story.
S	Scalable (small sized)	User Stories should not be so big as to become impossible to plan or Task or prioritize with some level of certainty.
T	Testable	The User Story or its related description must provide the necessary information to make test development possible.

1.2.3 Retrospectives

Some facts and some techniques:

- Meet at the end of each iteration
    - What worked and what didn't work so well
    - How to improve and how to retain success
- Topics: process, people, organizations, relationships, tools
- Follow through is required
- Essential to self-organization and continual improvement
- Address: test effectiveness/efficiency, test case quality, team satisfaction, and testability issues

The Retrospect Sprint Meeting is an important element of the 'inspect-adapt' Scrum framework and it is the final step in a Sprint. All Scrum Team members attend the meeting, which is facilitated or

moderated by the Scrum Master. It is recommended, but not required for the Product Owner to attend. One Team member acts as the scribe and documents discussions and items for future action. It is essential to hold this meeting in an open and relaxed environment to encourage full participation by all Team members. Discussions in the Retrospect Sprint Meeting encompass both what went wrong and what went right. Primary objectives of the meeting are to identify three specific items:

- Things the Team needs to keep doing—best practices
- Things the Team needs to begin doing—process improvements
- Things the Team needs to stop doing—process problems and bottlenecks

These areas are discussed and a list of Agreed Actionable Improvements is created

Explorer—Shopper—Vacationer—Prisoner (ESVP)

This exercise can be conducted at the start of the Retrospect Sprint Meeting to understand the mind-set of the participants and set the tone for the meeting. Attendees are asked to anonymously indicate which best represents how they feel regarding their participation in the meeting.

- Explorer—Wants to participate in and learn everything discussed in the retrospective
- Shopper—Wants to listen to everything and choose what he takes away from the retrospective
- Vacationer—Wants to relax and be a tourist in the retrospective
- Prisoner—Wants to be elsewhere and is attending the retrospective because it is required

The Scrum Master then collates the responses, prepares, and shares the information with the group.

Speed Boat

Speedboat is a technique that can be used to conduct the Retrospect Sprint Meeting. Team members play the role of the crew on a speedboat. The boat must reach an island, which is symbolic of the project vision. The attendees to record engines and anchors use sticky notes. Engines help them reach the island, while anchors hinder them from reaching the island. This exercise is Time-boxed to a few minutes. Once all items are documented, the information is collated, discussed, and prioritized by way of a voting process. Engines are recognized and mitigation actions are planned for the anchors, based on priority.

Metrics and Measuring Techniques

Various metrics can be used to measure and contrast the Team's performance in the current Sprint to their performance in previous Sprints. Some examples of these metrics include:

- Team velocity: Number of Story Points done in a given Sprint
- Done success rate: Percentage of Story Points that have been Done versus those Committed
- Estimation effectiveness: Number or percentage of deviations between estimated and actual

time spent on Tasks and User Stories

- Review feedback ratings: Feedback can be solicited from Stakeholder(s) using quantitative or qualitative ratings, providing a measurement of Team performance
- Team morale ratings: Results from self-assessments of Team member morale
- Peer feedback: 360 degree feedback mechanisms can be used to solicit constructive criticism and insight into Team performance
- Progress to release: Business value provided in each release, as well as value represented by the current progress towards a release. This contributes to the motivation of the Team and to the level of work satisfaction.

The outcome of the Retrospect Sprint Meeting

- Agreed Actionable: Improvements: Agreed Actionable Improvements are the primary output of the Retrospect Sprint process. They are the list of actionable items that the Team has come up with to address problems and improve processes in order to enhance their performance in future Sprints.
- Assigned Action Items and Due Dates: Once the Agreed Actionable Improvements have been elaborated and refined, action items to implement the improvements may be considered by the Scrum Team. Each action item will have a defined due date for completion.
- Proposed Non-Functional Items for Prioritized Product Backlog: When the initial Prioritized Product Backlog is developed, it is based on User Stories and required functionalities. Often, non-functional requirements may not be fully defined in the early stages of the project and can surface during the Sprint Review or Retrospect Sprint Meetings. These items should be added to the Prioritized Product Backlog as they are discovered. Some examples of non-functional requirements are response times, capacity limitations, and security related issues.
- Retrospect Sprint Log: The Retrospect Sprint Logs are records of the opinions, discussions, and actionable items raised in a Retrospect Sprint Meeting. The Scrum Master could facilitate creation of this log with inputs from Scrum Core Team members. The collection of all Retrospect Sprint Logs becomes the project diary and details project successes, issues, problems, and resolutions. The logs are public documents available to anyone in the organization.
- Scrum Team Lessons Learned: The self-organizing and empowered Scrum Team is expected to learn from any mistakes made during a Sprint. These lessons learned help the Teams improve their performance in future Sprints. These lessons learned may also be documented in Scrum Guidance Body Recommendations to be shared with other Scrum Teams.

There may be several positive lessons learned as part of a Sprint. These positive lessons learned are a key part of the retrospective, and should be appropriately shared within the Team and with the Scrum Guidance Body, as the Teams work towards continuous self-improvement.

- Updated Scrum Guidance Body Recommendations: Because of a Retrospect Sprint Meeting, suggestions may be made to revise or enhance the Scrum Guidance Body Recommendations. If the Guidance Body accepts these suggestions, these will be incorporated as updates to the Scrum Guidance Body documentation.

Retrospect Project

The Retrospect Project Meeting is a meeting to determine ways in which Team collaboration and effectiveness can be improved in future projects. Positives, negatives, and potential opportunities for improvement are also discussed. This meeting is not Time-boxed and may be conducted in person or in a virtual format. Attendees include the Project Team, Chief Scrum Master, Chief Product Owner, and Stakeholder(s). During the meeting, lessons learned are documented and participants look for opportunities to improve processes and address inefficiencies.

Some of the tools used in the Retrospect Sprint process can also be used in this process. Examples include:

- Explorer—Shopper—Vacationer—Prisoner (ESVP) exercise
- Speed Boat
- Metrics and Measuring Techniques

The outcome of the Retrospect Project Meeting

- Agreed Actionable Improvements: Agreed Actionable Improvements are the primary output of the Retrospect Project Review. They are the list of actionable items that the Team has come up with to address problems and improve processes in order to enhance their performance in future Sprints.
- Assigned Action Items and Due Dates: Once the Agreed Actionable Improvements have been elaborated and refined, action items to implement the improvements may be considered by the Scrum Team or Stakeholders. Each action item will have a defined due date for completion
- Proposed Non-functional Items for Program Product Backlog and Prioritized Product Backlog: When the initial Program Product Backlog or Prioritized Product Backlog are developed, they are based on User Stories and required functionalities. Often, non-functional requirements may not be fully defined in the early stages of the project and can surface during the Sprint Review, Retrospect Sprint or Retrospect Project Meetings. These items should be added to the Program Product Backlog (for the program) and Prioritized Product Backlog (for the project) as they are discovered. Some examples of non-functional requirements are response times, capacity limitations, and security related issues.

1.2.4 Continuous Integration

Continuous integration (**CI**) is the practice, in software engineering, of merging all developer working copies with a shared mainline several times a day. It was first named and proposed by Grady Booch in his method, did not advocate integrating several times a day. It was adopted as part of extreme programming (XP), which did advocate multiple integrations a day, perhaps as many as tens a day. The main aim of CI is to prevent integration problems, referred to as "integration hell" in early descriptions of XP. CI is not universally accepted as an improvement over frequent integration, so it is important to distinguish between the two, as there is disagreement about the virtues of each.

Some key points:

- Merge changes and integrate all code daily (or more often) for quick discovery of defects
- Developers code, debug, and check-in
- Continuous integration automatically:
    - Static code analysis
    - Compile
    - Unit test (functional/non-functional)
    - Deploy
    - Integration test (functional/non-functional)
    - Report
- Automated continuous integration, regression testing, and feedback
- Manual testing of new features, changes, and defect fixes
- Some defects are put in the product backlog

Benefits of and Risks Continuous Integration

- Benefits:

    - Early defect detection
    - Feedback on quality
    - Daily test releases
    - Lower regression risk
    - Confidence and visibility into progress
    - No big-bang integration
    - Executable software always on tap
    - Less repetitive testing
    - Quick feedback on process improvements

- Risks:
    - The introduction and ongoing use of CI tools may fail
    - The CI process may not be adopted by team
    - Test automation costs too much or takes too long
    - Insufficient skills to create good tests
    - Over-reliance on unit tests
- CI process requires tools

1.2.5 Release Planning

Release Planning Sessions are conducted to develop a Release Plan. The plan defines when various sets of usable functionality or products will be delivered to the Customer. In Scrum, the major objective of a Release Planning Meeting is to enable the Scrum Team to have an overview of the releases and delivery schedule for the product they are developing, so that they can align with the expectations of the Product Owner and relevant Stakeholders (primarily the project sponsor).

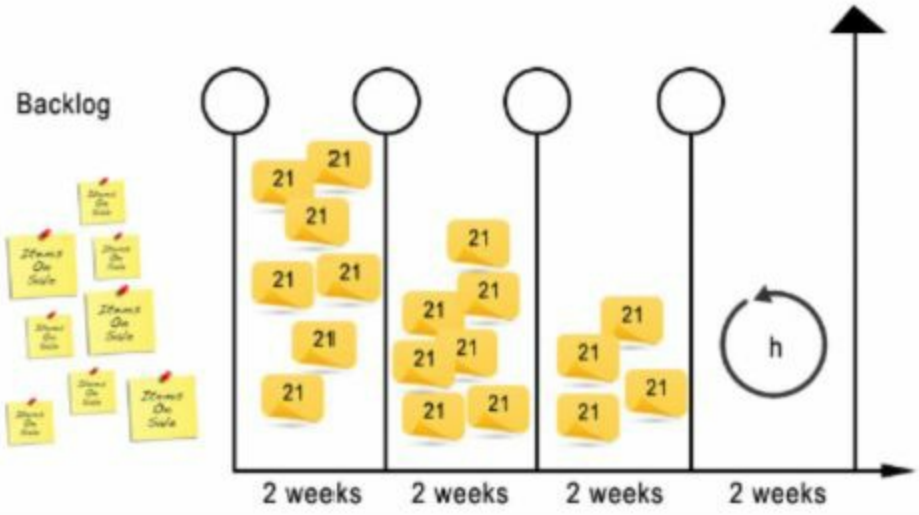
Release Planning

- Planning is an on-going activity
- For agile lifecycles: release and iteration planning
- Release planning
    - Define product backlog (user stories for release)
    - Refine user epics
    - Basis of test approach and test plan for release
    - Identify project/quality risks, estimate effort

- Testers:
    - Define testable user stories
    - Participate in project and quality risk analyses
    - Estimate test effort
    - Plan testing for release
- Release plans may change during project due to internal and external factors
- Testers must handle changes, see larger context of release, and obtain an adequate test basis and test oracles

Many organizations have a strategy regarding release of products. Some organizations prefer continuous deployment, where there is a release after creation of specified usable functionality. Other organizations prefer phased deployment, where releases are made at predefined intervals. Depending on the organization's strategy, Release Planning Sessions in projects may be driven by functionality, in which the objective is to deliver a release once a predetermined set of functionality has been developed; or the planning may be driven by date, in which the release happens on a predefined date.

Since Scrum framework promotes information-based, iterative decision making over the detailed upfront planning practiced in traditional waterfall style project management, Release Planning Sessions need not produce a detailed Release Plan for the entire project. The Release Plan can be updated continually, as relevant information is available.



Release Prioritization Methods

Release Prioritization Methods are used to develop a Release Plan. These methods are industry and organization specific and are usually determined by the organization's senior management.

Release Planning Schedule

A Release Planning Schedule is one of the key outputs of the Conduct Release Planning process. A

Release Planning Schedule states which Deliverables are to be released to the Customers, along with planned intervals, and dates for releases. There may not be a release scheduled at the end of every Sprint iteration. At times, a release may be planned after a group of Sprint iterations is completed. Depending on the organization's strategy, Release Planning sessions in projects may be driven by functionality, in which the objective is to deliver once a predetermined set of functionality has been developed, or the planning may be driven by date, in which case the release happens on a predefined date. The Deliverable should be released when it offers sufficient business value to the Customer.

Iteration Planning

Based on the various inputs including business requirements and Release Planning Schedule, the Product Owner and the Scrum Team decide on the Length of Sprint for the project. Once determined, the Length of Sprint often remains the same throughout the project.

However, the Length of Sprint may be changed if and when the Product Owner and the Scrum Team deem appropriate. Early in the project, they may still be experimenting to find the best Sprint length. Later in the project, a change in the Length of Sprint normally means it can be reduced due to improvements in the project environment.

A Sprint could be time-boxed for from 1 to 6 weeks. However, to get maximum benefits from a Scrum project, it is recommended to keep the Sprint time-boxed to 4 weeks, unless there are projects with very stable requirements, when Sprints can extend up to 6 weeks.

- Iteration planning
    - Define sprint backlog
    - Update test plan
    - Select and elaborate user stories
    - Analyze risks for user stories
    - Estimate each user story, reconcile with team velocity
    - Clarify use story with product owner
    - Assign tasks to the team
- Iteration plans sometimes change
- Testers
    - Participate in risk analysis
    - Determine testability
    - Create acceptance tests
    - Create tasks for user stories
    - Estimate test effort
    - Define test levels
    - Identify functional and non-functional test areas

- Work on test automation
- Communicate with team
- Accommodate changes

Target Customers for Release

Not every release will target all Stakeholders or users. The Stakeholder(s) may choose to limit certain releases to a subset of users. The Release Plan should specify the target Customers for the release.

Refined Prioritized Product Backlog

The Prioritized Product Backlog, developed in the Create Prioritized Product Backlog process, may be refined in this process. There may be additional clarity about the User Stories in the Prioritized Product Backlog after the Scrum Core Team conducts Release Planning Sessions with Stakeholder(s).

2 Fundamental Agile Testing Principles, Practices, and Processes

Agile testing follows the agile manifesto and principles; in addition to this, we have nine agile testing principles [Hendrickson]:

1. Testing Moves the Project Forward
2. Testing is NOT a Phase…
3. Everyone Tests
4. Shortening Feedback Loops
5. Tests represent expectations
6. Keep the Code Clean
7. Lightweight Documentation
8. "Done Done," Not Just Done
9. From test last to Test-Driven

There are also 10 other agile testing principles, which was introduced in the book Agile testing [By Lisa Crispin and Janet Gregory]. So easy to get confused, but both of them contain good principles

1. Provide continuous feedback.
2. Deliver value to the customer.
3. Enable face-to-face communication.
4. Have courage.
5. Keep it simple.
6. Practice continuous improvement.
7. Respond to change.
8. Self-organize.
9. Focus on people.
10. Enjoy.

Agile is different from traditional lifecycles and the principles also shows that. If you do not remember or have seen the traditional testing principles, they come here.

ISTQBs 7 testing principle:

1. Testing shows presence of defects
2. Exhaustive testing is impossible
3. Early testing
4. Defect clustering
5. Pesticide paradox
6. Testing is context dependent
7. Absence-of-errors fallacy

The seven traditional testing principles also works fine for agile projects, they are the core of testing and should not be forgotten just because the projects is agile. Another thing to mention is that traditional projects also can adapts many of the agile testing principles.

2.1 The Differences between Testing in Traditional and Agile Approaches

There are differences between testing in traditional lifecycle models (e.g., sequential such as the V-model or iterative such as RUP) and Agile lifecycles in order to work effectively and efficiently. The Agile models differ in terms of the way testing and development activities are integrated, the project work products, the names, entry and exit criteria used for various levels of testing, the use of tools, and how independent testing can be effectively utilized.

Organizations vary considerably in their implementation of lifecycles. Deviation from the ideals of Agile lifecycles may represent intelligent customization and adaptation of the practices. The ability to adapt to the context of a given project, including the software development practices actually followed, is a key success factor for testers.

Let us look at some of the differences:

2.1.1 Testing and Development Activities

Comparing Agile and Sequential 1	
Agile	Sequential
• Short iterations deliver valuable, working features • Release and iteration planning and quality risk analysis • User stories selected in each iteration • Continuous test execution, overlapping test levels • Testers, developers, and business stakeholders test • Hardening iterations may occur periodically • Avoid accumulating technical debt • Pairing (at least in XP) • Testing and quality coaching is a	• Longer timeframes, deliver large groups of features • Overall planning and risk analysis up front, with control throughout • Scope of requirements established up front • Test execution in sequential levels in last half of project • Testers, developers, and business stakeholders test • Hardening happens at end of lifecycle, in system test/SIT • Unrecognized technical debt is a major risk • Pairing is unusual • Testing and quality coaching is a

best practice • Heavy use of test automation for regression risk • Change may occur during project—deal with it • Lightweight work products	best practice • Test automation is a best practice • Unmanaged change can result in a death march • Risk of over-documentation

Big bang vs iterations deliveries:

One of the main differences between traditional lifecycles and agile lifecycles is the idea of very short iterations, each iteration resulting in working software that delivers features of value to business stakeholders. In an agile team, testing activities occur throughout the iteration, not as a final activity.

Role in testing

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

Test coaches

On agile teams, we often see testers serve as testing and quality coaches within the team, sharing testing knowledge and supporting quality assurance work within the team. This promotes a sense of collective ownership of quality of the product.

Test Automation

You will find that Test automation at all levels of testing occurs in many agile teams, and this can mean that testers spend time creating, executing, monitoring, and maintaining automated tests and results. Because of the heavy use of test automation, a higher percentage of the manual testing on agile projects tends to be done using experience-based and defect-based techniques such as software attacks, exploratory testing, and error guessing.

Changes

One core agile principle is that change may occur throughout the project. Therefore, lightweight work product documentation is favored in agile projects. Changes to existing features have testing implications, especially regression testing implications. The use of automated testing is one way of managing the amount of test effort associated with change.

2.1.2 Project Work Products

Categories of work products [Agile test Syllabus]:

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

On agile projects, we try to minimize documentation, emphasize working software and automated tests.

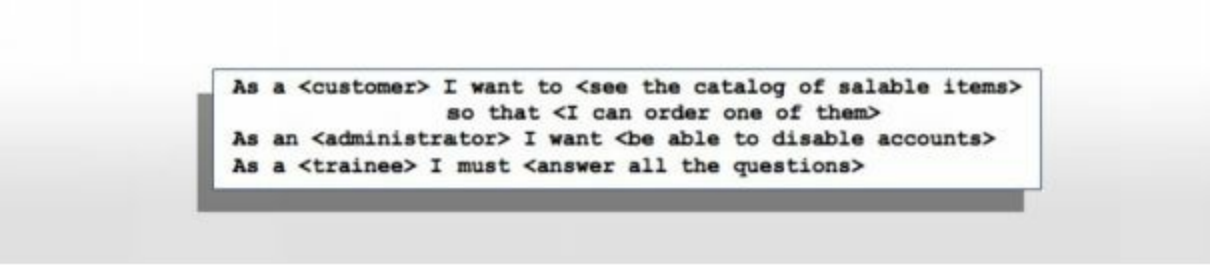
Avoid producing vast amounts of documentation. Instead, focus is more on having working software, together with automated tests that demonstrate conformance to requirements.

Sufficient documentation must be provided for business, testing, development, and maintenance activities:

In a successful agile project, a balance is struck between increasing efficiency by reducing documentation and providing sufficient documentation to support business, testing, development, and maintenance activities.

Typical business-oriented work products on agile projects:

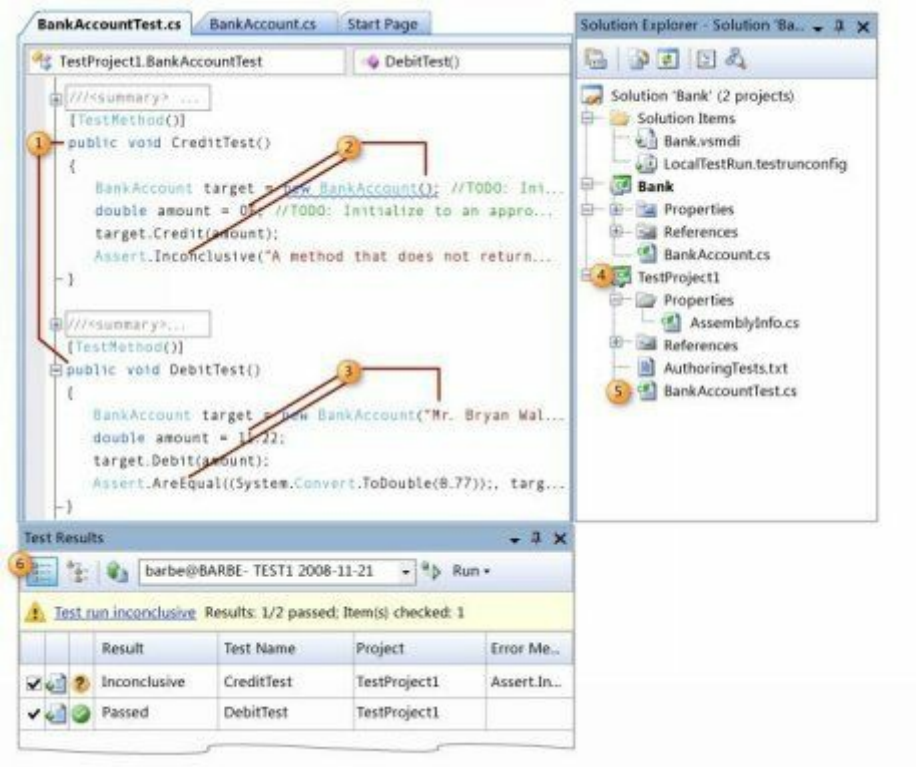
- User stories (the agile form of requirements specifications)
- Acceptance criteria



User story example

Typical developer work products on agile projects:

- Code
- Unit test (normal this will be automated)
  If unit test are created incrementally, before each portion of the code is written, in order to provide a way of verifying, once that portion of code is written, whether it works as expected. While this approach is referred to as test first or test-driven development, in reality the tests are more a form of executable low-level design specifications rather than tests.



Unit test example

Typical tester work products on agile projects:

- Manuel tests
- Automated tests
- Test plans (lightweight)
- Quality risk catalogues (lightweight)
- Defects reports and results logs (lightweight)
- Test metrics (lightweight)

Agile testing dashboard example

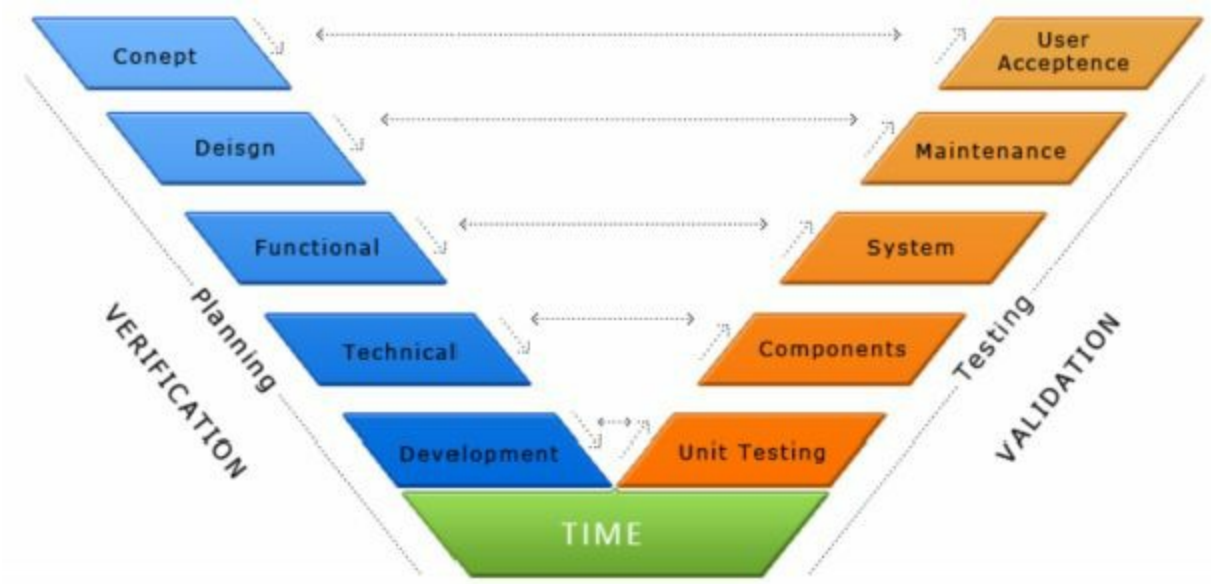
In regulated, safety critical, distributed, or highly complex projects, more documentation is often required:

On some agile projects, teams transform user stories and acceptance criteria into more formal requirements specifications. Vertical and horizontal traceability reports may be prepared to satisfy auditors, regulations, and other requirements.

2.1.3 Test Levels

In waterfalls lifecycle models (sequential lifecycle) and the expanded V-Model, the test levels are often defined such that the exit criteria of one level are part of the entry criteria for the next level. In the below model, you see five test levels:

- Unit Testing
- Component Testing
- System Testing
- Maintenance Testing
- User Acceptance Testing



V-Model:

In some iterative models, this rule does not apply. Test levels overlap. Requirement specification, design specification, and development activities may overlap with test levels.

Agile Test Levels:

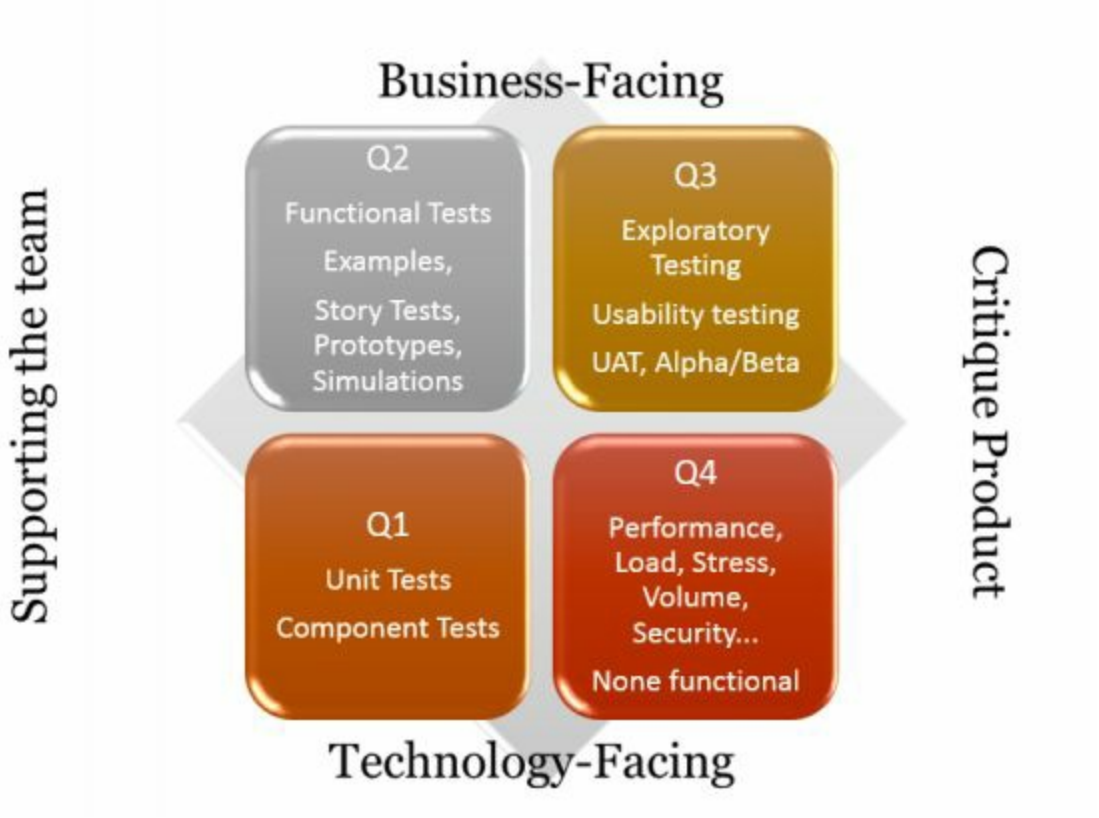
Changes to requirements, design, and code can happens often during agile, but changes are normally kept in the product backlog, but can happen at any point in an iteration. During an iteration, any given user story will typically progress sequentially through the following test activities:

- Unit testing (developer, often automated)
- Feature acceptance testing
- Feature verification (developer or tester, often automated, based on acceptance criteria)
    - Feature validation (developers, testers, and business stakeholders, usually manual, shows usefulness, progress)
    - Regression testing throughout the iteration (via automated unit tests and feature verification tests)
- System testing (tester, functional and non-functional)
- Unit integration testing (dev+test, sometimes not done) System integration testing (tester, sometimes one iteration behind)
- Acceptance testing (alpha, beta, UAT, OAT, regulatory, and contractual, at end of each iteration, after each iteration, or after a series of iterations)

Some highlights of this:

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

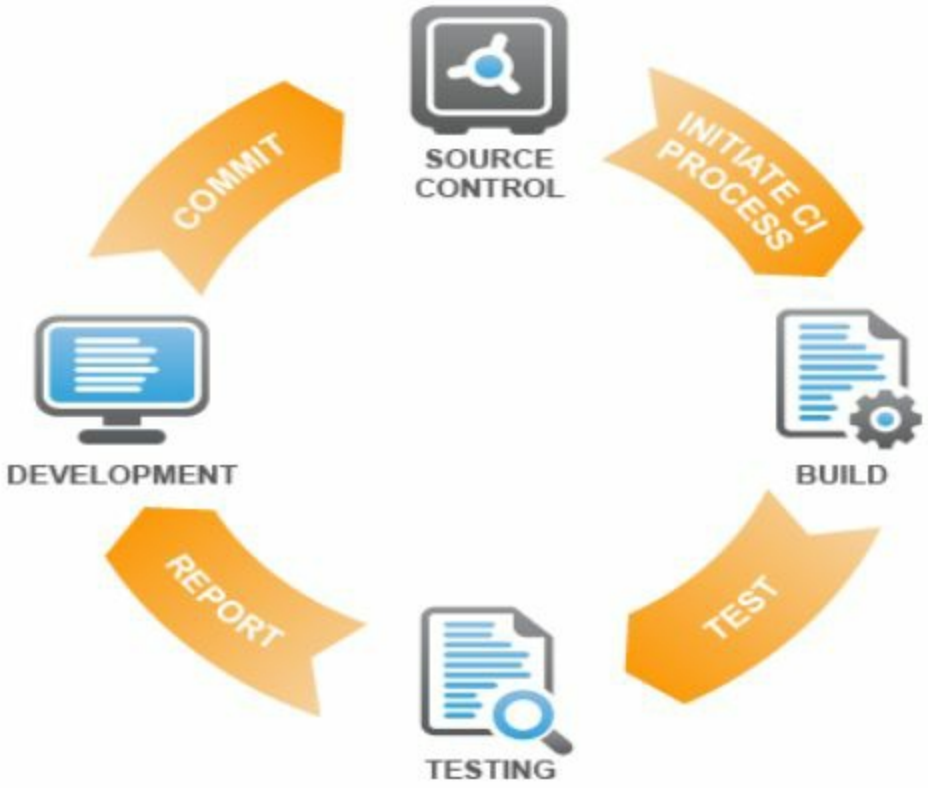
There may be a system test level, which starts once the first user story is ready for such testing. This can involve executing functional tests, as well as non-functional tests for performance, reliability, usability, and other relevant test types. This will be determined under the grooming of the user stories before the iteration starts. A good help to do this analyses is to uses the agile test quadrant to make sure you get the right quality focus on each user story.



Internal alpha tests and external beta tests may occur, at the close of each iteration, either after the completion of each iteration, or after a series of iterations. User acceptance tests, operational acceptance tests, regulatory acceptance tests, and contract acceptance tests also may occur, at the close of each iteration, either after the completion of each iteration, or after a series of iterations.

2.1.4 Testing and Configuration Management

Automated tools in agile projects are often used to develop, test, and manage software development. Static analysis tools are used by Developers for unit testing, and code coverage.



Developers continuously check the code and unit tests into a configuration management system, using automated build and test frameworks. These frameworks allow the continuous integration of new software with the system, with the static analysis and unit tests run repeatedly as new software is checked in.

Automated tests, at all levels, in build/test frameworks, achieving continuous integration:

Software tests have to be repeated often during development cycles to ensure quality. Every time source code is modified software, tests should be repeated. For each release of the software, it may be tested on all supported operating systems and hardware configurations. Manually repeating these tests is costly and time consuming. Once created, automated tests can be run repeatedly at no additional cost and they are much faster than manual tests.

Some of the goal of the automated tests are:

- **Confirm that the build is functioning and installable**
  If any automated test fails, the team should fix the underlying defect in time for the next code check-in.

- Help to manage the regression risk associated with the frequent change that often occurs in agile projects.

- Saves Time and Money
  Automated software testing can reduce the time to run repetitive tests from days to hours. A timesaving that translates directly into cost savings.

- Improves Accuracy
  Automated tests perform the same steps precisely every time they are executed and never forget to record detailed results.

- Increases Test Coverage
  Automated software testing can increase the depth and scope of tests to help improve software quality. Lengthy tests that are often avoided during manual testing can be run

unattended. They can even be run on multiple computers with different configurations. Automated software testing can look inside an application and see memory contents, data tables, file contents, and internal program states to determine if the product is behaving as expected. Automated software tests can easily execute thousands of different complex test cases during every test run providing coverage that is impossible with manual tests.

2.1.5 Organizational Options for Independent Testing

When we think about how independent the test team is? It is very important to understand that independence is not an either/or condition, but a range and yes it will occur in agile team as well:

- At one end of the range lies the absence of independence, where the programmer performs testing within the programming team.
- Moving toward independence, we find an integrated tester or group of testers working alongside the programmers, but still within and reporting to the development manager.
- Then moving little bit more towards independence we might find a team of testers who are independent and outside the development team, but reporting to project management.
- Near the other end of the continuum lies complete independence. We might see a separate test team reporting into the organization at a point equal to the development or project team. We might find specialists in the business domain (such as users of the system), specialists in technology (such as database experts), and specialists in testing (such as security testers, certification testers, or test automation experts) in a separate test team, as part of a larger independent test team, or as part of a contract, outsourced test team.

In some Agile teams, developers create many of the tests in the form of automated tests. One or more testers may be embedded within the team, performing many of the testing tasks. However, given those testers' position within the team, there is a risk of loss of independence and objective evaluation.

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

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

2.2 Status of Testing in Agile Projects

On Agile projects changes takes place often. When many changes happen, we can expect that test status, test progress, and product quality constantly evolve, and testers need to find ways to get that information to the team so that they can make decisions to stay on track for successful completion of each iteration.

Change can affect existing features from previous iterations. Therefore, manual and automated tests must be updated to deal effectively with regression risk.

Short summary:

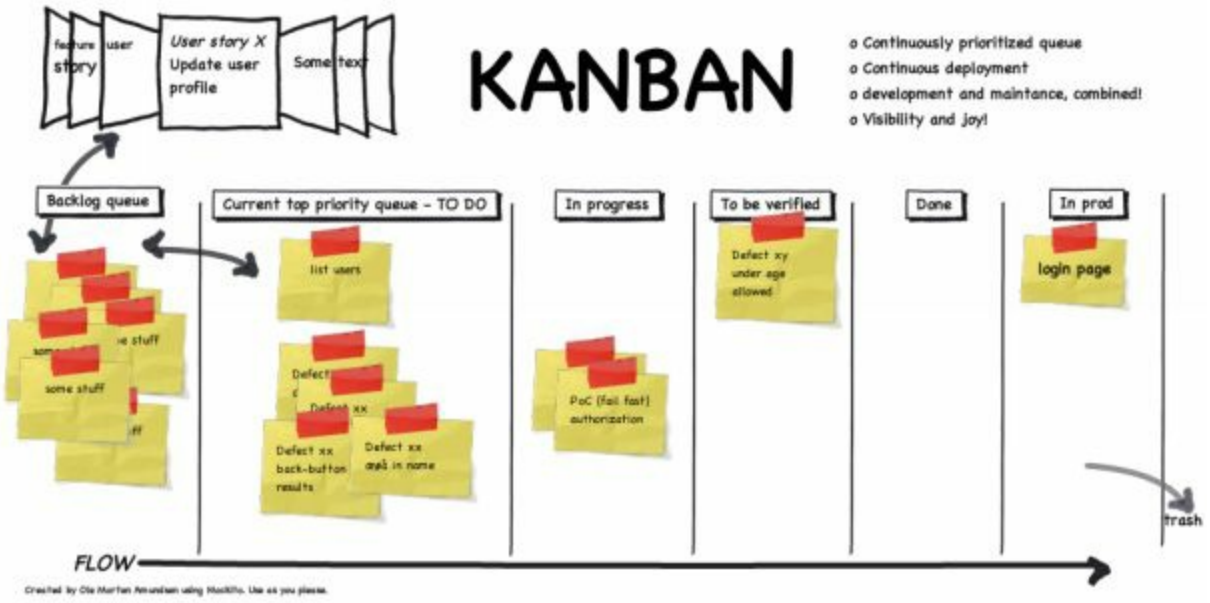
- On agile project, change happens, rapidly and often
- When the features change, so does product quality
- Change can make a mess of your status reporting processes if you're not careful
- Change also means that accurate test status is critical for smart team decisions
- Change can have an impact on features from previous iterations, meaning regression testing is needed.
- Change often means existing tests must change

Let us look at some of the differences:

2.2.1 Communicating Test Status, Progress, and Product Quality

Agile teams progress by having working software at the end of each iteration. To determine when the team will have working software, the teams can monitor progress in different ways:

- Test progress can be recorded using automated test results, agile task boards, and burndown charts
- Test status can be communicated via wikis, standard test management tools, and during stand-ups
- Project, product, and process metrics can be gathered (e.g., customer satisfaction, test pass/fail, defects found/fixed, test basis coverage, etc.)
    - Metrics should be relevant and helpful
    - Metrics should never be misused
- Automating the gathering and reporting of status and metrics allows testers to focus on testing



Example of Kanban task board.

It is common practice that the Sprint Backlog is represented on a Scrum board or task board, which provides a constantly visible depiction of the status of the User Stories in the backlog. Also included in the Sprint Backlog are any risks associated with the various tasks. Any mitigating activities to address the identified risks would also be included as tasks in the Sprint Backlog.

The story cards, development tasks, test tasks, and other tasks created during iteration planning are captured on the task board, often using colour-coordinated cards to determine the task type. During the iteration, progress is managed via the movement of these tasks across the task board into columns such as to do, work in progress, verify, and done. Agile teams may use tools to maintain their story cards and agile task boards, which can automate dashboards and status updates.

A bit more on some of the topics:

Burndown charts:

Burndown charts can be used to track progress across the entire release and within each iteration. A burndown chart represents the amount of work left to be done against time allocated to the release or iteration.

The Sprint Burndown Chart is a graph that depicts the amount of work remaining in the ongoing Sprint. A planned Burndown accompanies the initial Sprint Burndown Chart. The Sprint Burndown Chart should be updated at the end of each day as work is completed. This chart shows the progress that has been made by the Scrum Team and allows for the detection of estimates that may have been incorrect. If the Sprint Burndown Chart shows that the Scrum Team is not on track to finish the tasks in the Sprint on time, the Scrum Master should identify any obstacles or impediments to successful completion and try to remove them

A related chart is a Sprint Burnup Chart. Unlike the Sprint Burndown Chart that shows the amount of work remaining, the Sprint Burnup Chart depicts the work completed as part of the Sprint.

Example of Sprint Burndown chart

The initial Sprint Backlog defines the start-point for the remaining efforts. The remaining effort of all activities are collected on a daily base and added to the graph. In the beginning, the performance is often not as good as predicted by the ideal burndown rate due to wrong estimations or impediments that have to be removed in order to get on full speed.

To provide an instant, detailed visual representation of the whole team's current status, including the status of testing, teams may use Agile task boards.

Acceptance criteria:

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

team reviews and addresses any issues that may be blocking the progress of those tasks.

Daily stand-up meeting:

- In agile task boards, tasks move into columns (*to do*, *work in progress*, *verify*, and *done*)
- *Done* means all tests for the task pass
- Task board status reviewed during stand-ups, which include testers and developers (whole team)
- Each attendee should address:
    - What have you completed since the last meeting?
    - What do you plan to complete by the next meeting?
    - What is getting in your way?
- Team discusses any blockages or delays for any task, and works collaboratively to resolve them

Other ways to capture test and quality status:

To secure overall product quality, Agile teams my perform customer satisfaction surveys to receive feedback on whether the product meets customer expectations.

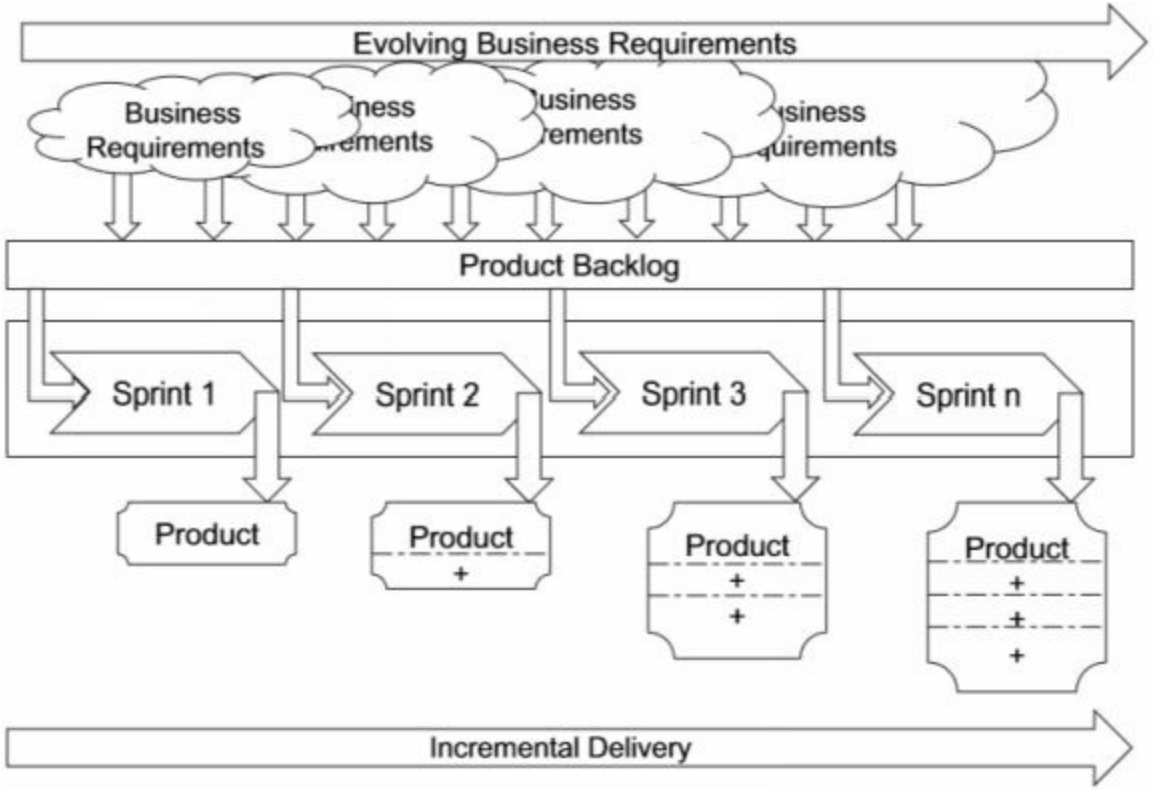
Agile teams may also use other metrics similar to those captured in traditional development methodologies, such as;

- test pass/fail rates,
- defect discovery rates,
- confirmation and regression test results,
- defect density,
- defects found and fixed,
- requirements coverage,
- risk coverage,
- code coverage,
- code churn to improve the product quality.

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

2.2.2 Managing Regression Risk with Evolving Manual and Automated Test Cases

In an Agile project, as each iteration completes, the product. Therefore, the scope of testing also increases.



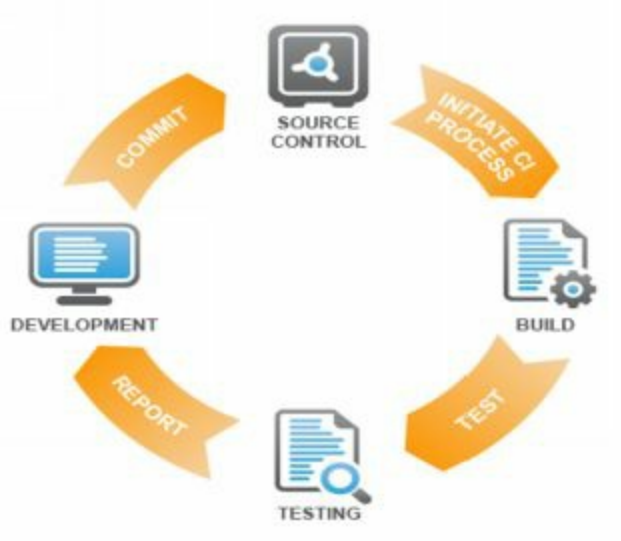
Regression Testing:

Along with testing the code changes made in the current iteration, testers also need to verify no regression has been introduced on features that were developed and tested in previous iterations. The risk of introducing regression in Agile development is high due to extensive code churn (lines of code added, modified, or deleted from one version to another).

Automation of test:

Since responding to change is a key Agile principle, changes can also be made to previously delivered features to meet business needs. In order to maintain velocity without incurring a large amount of technical debt, it is critical that teams invest in test automation at all test levels as early as possible.

- Automated tests at all levels reduce technical debt and provide rapid feedback
- Run automated regression tests in continuous integration and before putting a new build in test
    - Automated unit tests provide feedback on code and build quality (but not on product quality)
    - Automated acceptance tests provide feedback on product quality with respect to regression



Continuous integration

- Fix regression and build bugs immediately
- For efficiency, also automate test data generation and loading, build deployment, environment management, output comparison

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

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

Automated acceptance tests are run regularly as part of the continuous integration full system build. These tests are run against a complete system build at least daily, but are generally not run with each code check-in as they take longer to run than automated unit tests and could slow down code check-ins. The test results from automated acceptance tests provide feedback on product quality with respect to regression since the last build, but they do not provide status of overall product quality.

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

Automated tests contained in the regression test set are generally run as part of the daily main build in the continuous integration environment, and again when a new build is deployed into the test environment. As soon as an automated regression test fails, the team stops and investigates the

reasons for the failing test. The test may have failed due to legitimate functional changes in the current iteration, in which case the test and/or user story may need to be updated to reflect the new acceptance criteria. Alternatively, the test may need to be retired if another test has been built to cover the changes. However, if the test failed due to a defect, it is a good practice for the team to fix the defect prior to progressing with new features.

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

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

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

Doing test automation in a wrong way can be costly, so also important for the tester in agile team to look at the whole test tool/test automation process, here some hint;

Keep test updated:

It is also critical that all test assets such as automated tests, manual test cases, test data, and other testing artifacts are kept up- to-date with each iteration. It is highly recommended that all test assets be maintained in a configuration management tool in order to;

- enable version control,
- ensure ease of access by all team members,
- support making changes as required due to changing functionality while still preserving the historic information of the test assets.

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

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

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

2.3 Role and Skills of a Tester in an Agile Team

In an Agile team, testers have to collaborate with all other team members and with business stakeholders. Therefor a tester need skills so they can fulfill the activities they perform within an agile team.

2.3.1 Agile Tester Skills

Agile testers should have skills from traditional testing methods. Just as a Java team member should have Java skills, a tester should have testing skills. In addition to these skills, a tester in an agile team should be competent in test automation, test-driven development, acceptance test-driven development, white-box, black box, and experience-based testing.

Agile methodologies depend heavily on collaboration, communication, and interaction between the team members as well as stakeholders outside the team, testers in an agile team need good interpersonal skills.

Testers in Agile teams should:

- Act positive and solution-oriented with team members and stakeholders
- Have a critical eye: Display critical, quality-oriented, sceptical thinking about the product
- Be pro-Active: Actively acquire information from stakeholders (rather than relying entirely on written specifications)
- Accurately evaluate and report test results, test progress, and product quality
- Use the tester skills: Work effectively to define testable user stories, especially acceptance criteria, with customer representatives and stakeholders
- Be a good Team-member: Collaborate within the team, working in pairs with programmers and other team members
- Have a open mind-set: Respond to change quickly, including changing, adding, or improving test cases
- Be professional: Plan and organize their own work

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

2.3.2 The Role of a Tester in an Agile Team

The role of a tester in an agile team includes activities that generate and provide feedback not only on test status, test progress, and product quality, but also on process quality. These activities include:

- Understanding, implementing, and updating the test strategy
- Measuring and reporting test coverage across all applicable coverage dimensions
- Ensuring proper use of testing tools

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

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

Agile organizations may encounter some test-related organizational risks:

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

To mitigate these risks, organizations may consider variations for preserving independence.

3 Agile Testing Methods, Techniques, and Tools

3.1 Agile Testing Methods

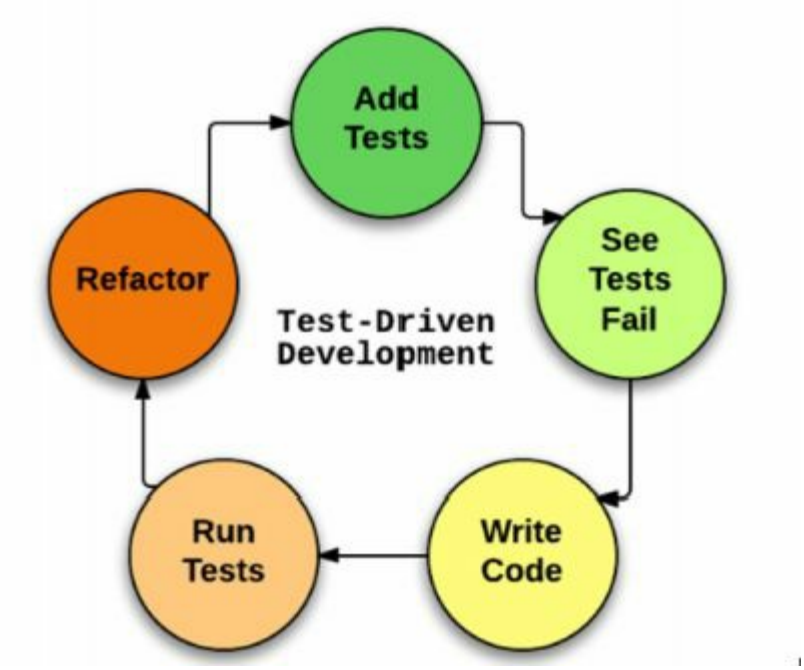
Agile or not, there are testing practices on every project or team which can be used to secure quality products. In addition, principles from the traditional testing are used a lot within the agile testing approach. “Early testing” is one of them, where a well know agile approach include writing tests in advance to express proper behavior, focusing on early defect prevention, detection, and removal, and ensuring that the right test types are run at the right time and as part of the right test level. Let us have a look at some of the most used agile testing approaches.

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

First, we will look at Test-driven development often call TDD but we will also look at, acceptance test-driven development, and behavior-driven development. All three complementary techniques in use among agile teams to carry out testing across the various test levels. All 3 techniques represent the traditional testing principles “Test Early”, since the tests are defined before the code is written.

Test-Driven Development (TDD):

Test-driven development (TDD) is used to develop code guided by automated test cases. The process for test-driven development is as shown in the figure:



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

Often this is on unit level and are code-focused, but can also be used at the integration or system levels. TDD got popularity through Extreme Programming, but is also used in other Agile methodologies and also used by traditional testing projects. Using TDD makes it easy for the developers to focus on clearly-defined expected results. Normal these tests are automated and are used in continuous integration.

Acceptance Test-Driven Development (ATDD):

Acceptance Test-Driven Development is based on communication between the business customers, the developers, and the testers, and it defines acceptance criteria and tests during the creation of user stories. Here are some key element for ATDD:

- Define acceptance criteria and tests early in development
- Collaborate so every stakeholder understands behavior
- Process (more about the process later):
    - Define tests for intended behavior
    - Create automated acceptance tests

    - Program intended behavior
    - Run automated acceptance tests
- Create reusable regression tests for continuous integration
- Test data and service layers
- Test system/acceptance level in appropriate environments
    - Identify/quickly resolve defects
    - Validate feature behavior
    - Measure acceptance criteria
    - Deliver to external testing teams

Jennitta Andrea has described ATDD as (Andrea, 2010) . . . "*the practice of expressing functional story requirements as concrete examples or expectations prior to story development. During story development, a collaborative workflow occurs in which: examples are written and then automated; granular automated unit tests are developed; and the system code is written and integrated with the rest of the running, tested software. The story is "done"— deemed ready for exploratory and other testing— when these scope-defining automated checks pass*".

Here some details regarding some of the points:

Acceptance criteria and tests early in development Tests early:

The team agrees on a list of criteria, which must be met before a product increment "often a user story" is considered "done", and we can say that this defines acceptance criteria and tests during the creation of user stories as early as possible.

One of the software testing principles says "Start Testing Early" in the software development life cycle. This also count for agile projects. A lot of misunderstanding can me avoid through the life cycle if we manage to get good and understandable "done criteria's" as early as possible.

Also, remember early testing is not only about making use acceptance criteria', it is also about review of existing documentation and other kind of test basis, to secure it has an acceptable content and structure for the team to work with. Good reviews can clean out a lot of misunderstanding up front.

Reusable tests for regression testing:

Acceptance test-driven development creates reusable tests for regression testing. Regression testing is a type of software testing that seeks to uncover new software bugs, or regressions, in existing functional and non-functional areas of a system after changes such as enhancements, patches or configuration changes, have been made to them.

The intent of regression testing is to ensure that changes such as those mentioned above have not introduced new faults. One of the main reasons for regression testing is to determine whether a change

in one part of the software affects other parts of the software. In agile development, this is extremely important because we change the code in every sprint. Therefore, we need to make sure that code that worked in previous sprint still works after new implementations. Regression testing is an integral part of agile development methods and it is a common goal to automation as many of these test as possible.

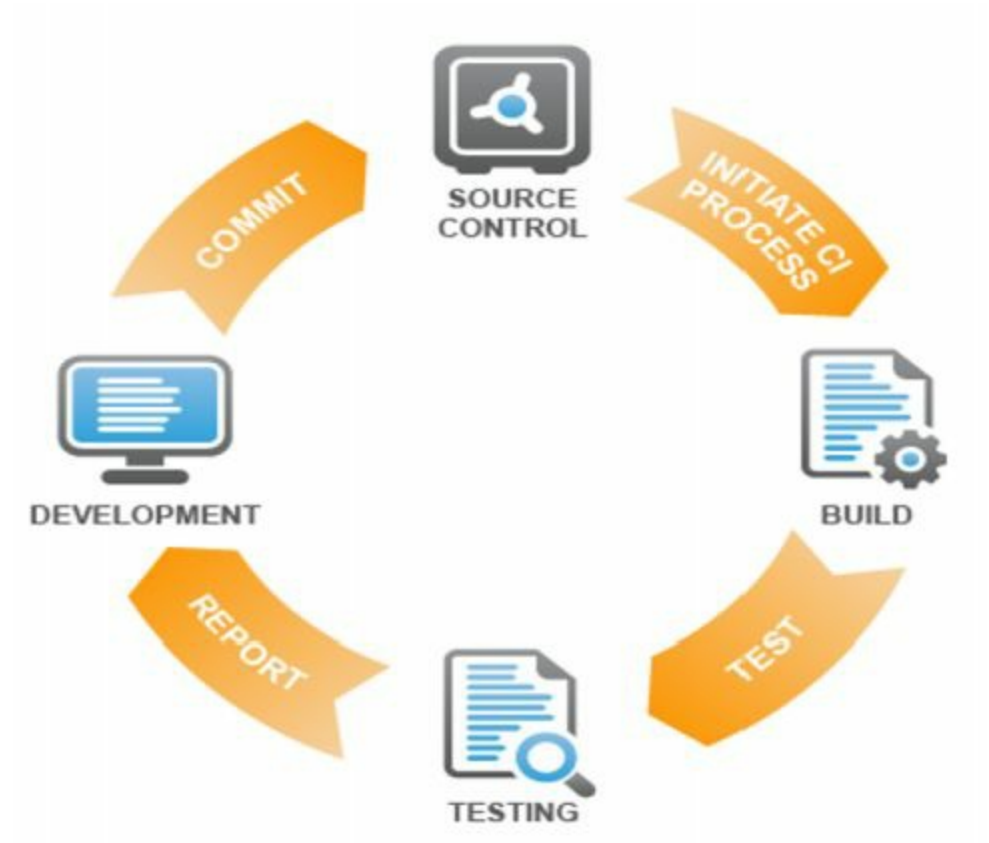
Continuous integration process:

Specific tools support creation and execution of such tests, often within the continuous integration process.

Continuous Integration is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily - leading to multiple integrations per day. An automated build (including test) to detect integration errors as quickly as possible verifies each integration. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

Continuous integration

- Fix regression and build bugs immediately
- For efficiency, also automate test data generation and loading, build deployment, environment management, output comparison



Test data and service layers tools:

These tools can connect to data and service layers of the application, which allows tests to be executed at the system or acceptance level.

ATDD allows quick resolution of defects and validation of feature behavior. It helps determine if the DONE criteria (acceptance criteria) are met for the feature.

Behavior-Driven Development (BDD):

BDD is similar in many ways to TDD except that the word "test" is replaced with the word "Behavior". Its purpose is to help the team to focus on testing the code based on the expected behavior of the software. BDD is usually done in very English-like language, which helps the Domain experts to understand the implementation. Some key points:

- Behavior-driven development is black-box approach
- Focus on expected behavior:
    - Developer creates a test for the class under development
    - Tests should make sense to stakeholders (including testers)
    - Clarify where defect lies (code, user story, or test)
- Helps to define test cases for developer based on tester/stakeholder collaboration

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

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

Here an example on a story that is described using the BDD method:

Story: Books returns go to stock
In order to keep track of book stock
As a book store owner
I want to add books back to stock when they are returned

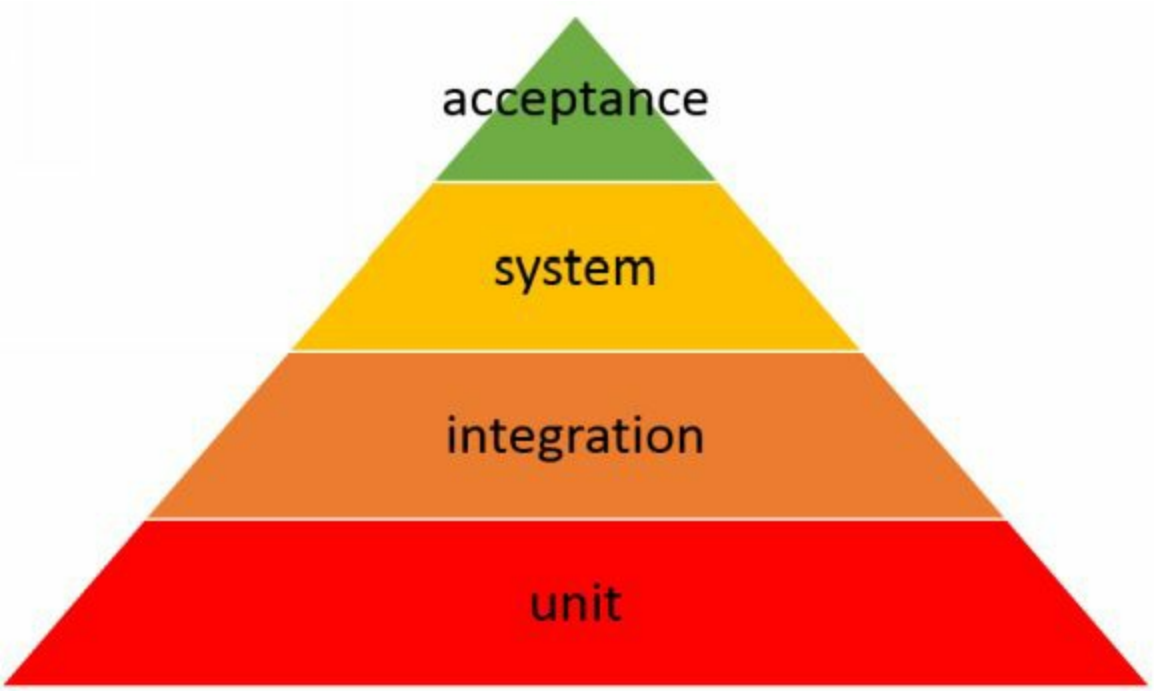
Scenario 1: Refunded books should be returned to stock
Given a customer previously bought a book from me
And I currently have three books left in stock
When he returns the book for a refund
Then I should have four books in stock

Scenario 2: Replaced books should be returned to stock
Given that, a customer buys a softcase book
And I have two softcase books in stock
And three hardcase in stock.
When he returns the softcase for a replacement in harcase,
Then I should have three softcase in stock
And two hardcase in stock

From these requirements, the behavior-driven development framework generates code that can be used by developers to create test cases. Behavior-driven development helps the developer collaborate with other stakeholders, including testers, to define accurate unit tests focused on business needs. There are several different examples of BDD software tools in use in projects today, for different platforms and programming languages.

3.1.2 The Test Pyramid

A software system may be tested at different levels. Often we show this as the Test Pyramid:

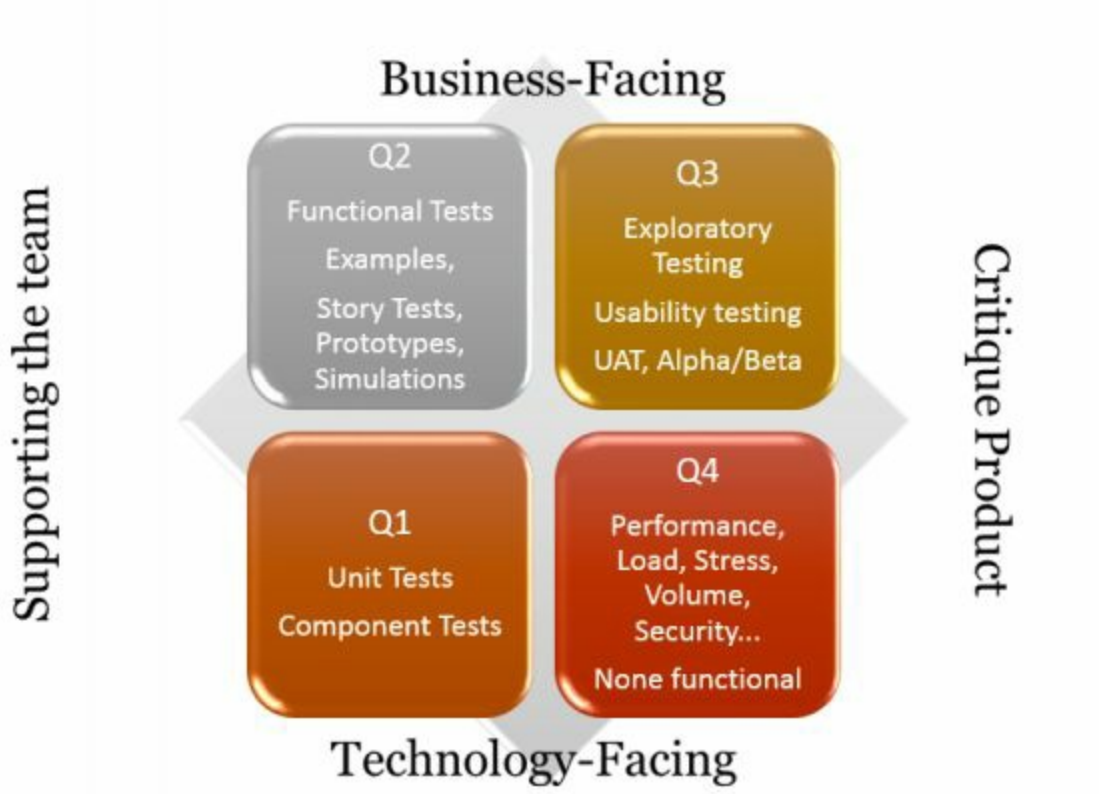


- Test pyramid emphasizes lower-level tests over upper-level tests
- Automate unit and integration tests through API
- Automate system and acceptance tests through GUI
- Test pyramid concept is early QA and testing principle in action

3.1.3 Testing Quadrants, Test Levels, and Testing Types

Testing quadrants align the test levels with the appropriate test types in the agile methodology.

- The test quadrants associate test levels, types, goals, and focus
    - Covers important dynamic types/levels
    - Differentiates, describes types
- Any test shown in any quadrant can occur in each iteration



Agile testing quadrant: defined by Brian Marick

How to find out about which testing types to be used testing your Feature and stories in agile projects? It is useful to ask if those tests are

- Business facing or
- Technology facing.

A business-facing test is one you could describe to a business/domain expert. Here you are talking to the users of the system, using user terms and language: "If a book is sold out in your online book store, does the system automatically set up a message if a potential buyer search for the book “Book is sold out”, or will the search result not show the book in the search list?»

In the testing quadrants, tests can be business (user) or technology (developer) facing.

Some tests:

- Support the work done by the agile team and confirm software behavior.
- Other tests can verify the product.

Tests can be fully manual, fully automated, a combination of manual and automated, or manual but supported by tools. The four quadrants are as follows [ISTQB syllabus]:

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

Test from each of the quadrants can be used in any given iteration. For more information about the agile testing quadrant, visit the blog from Brian Marick.

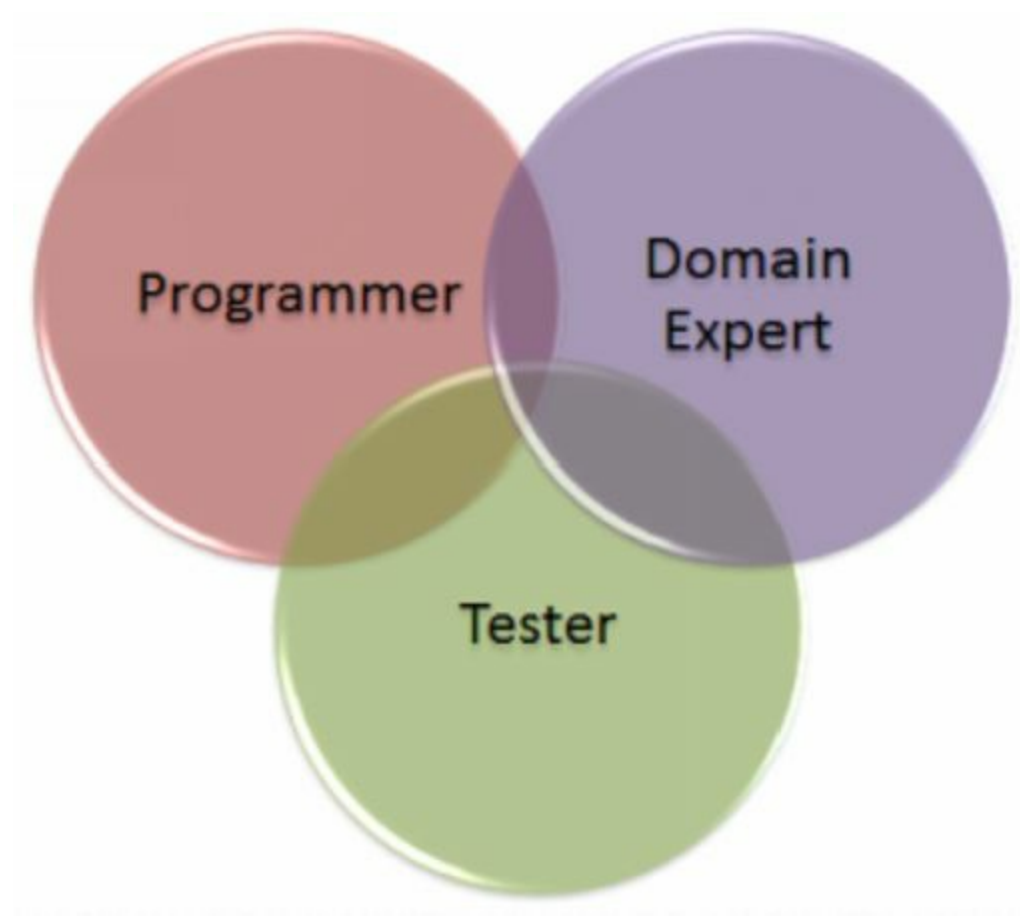
3.1.4 The Role of a Tester

If we take the agile method Scrum as an example we will here have a look at the role "Tester" in the agile lifecycle. We will look at the following:

- Teamwork
- Sprint Zero
- Integration
- Test planning
- Agile testing practices

Teamwork

Agile is all about teamwork, often known in the agile methods as the whole-team approach. It consist of three role:



The 3 roles represent the team, but let us have a look at the role “Tester”.

- Teamwork is fundamental to agile:
    - **Cross functional**: a group of people with different functional expertise working toward a common goal. Working together on test strategy, test planning, test specification, test execution, test evaluation, and test results reporting
    - **Self-organizing**: a process where some form of global order or coordination arises out of the local interactions between the components of an initially disordered system. The team may consist only of developers, but ideally, there would be one or more testers and domain experts.
    - **Collocated**: Testers sit together with the developers and the product owner.
    - **Collaborative**: is working with others to do a task and to achieve shared goals. Testers collaborate with their team members, other teams, the stakeholders, the product owner, and the Scrum Master.
    - **Empowered**: Technical decisions regarding design and testing are made by the team as a whole (developers, testers, and Scrum Master), in collaboration with the product owner and other teams if needed.
    - **Committed**: A funny little fable is often told to shows the strong commitment needed in agile teams:

      A Pig and a Chicken are walking down the road.
      The Chicken says: "Hey Pig, I was thinking we should open a restaurant!"
      Pig replies: "Hm, maybe, what would we call it?"

The Chicken responds: "How about 'ham-n-eggs'?"
The Pig thinks for a moment and says: "No thanks. I'd be committed, but you'd only be involved!"

The tester is committed to question and evaluate the product's behavior and characteristics with respect to the expectations and needs of the customers and users.

- **Transparent**: used in science, engineering, business, the humanities and in a social context more generally, implies openness, communication, and accountability. Development and testing progress is visible on the Agile task boar
- **Credible**: The tester must ensure the credibility of the strategy for testing, its implementation, and execution, otherwise the stakeholders will not trust the test results. This is often done by providing information to the stakeholders about the testing process.
- **Open to feedback**: (from Latin retrospectare, "look back") generally means to take a look back at events that already have taken place. Feedback is an important aspect of being successful in any project, especially in Agile projects. Retrospectives allow teams to learn from successes and from failures.
- Resilient: Testing must be able to respond to change, like all other activities in agile projects.

- Teamwork maximizes likelihood of successful testing

Sprint Zero

Sprint zero is usually necessary because there might be things that need to be done before a Scrum project can start and it is the first iteration of the project where many preparation activities take place.

The tester collaborates with the team on the following activities during this iteration:

- First iteration:
  - Identify scope
  - Divide user stories into sprints

    - Create system architecture
    - Plan, acquire, and install tools
    - Create initial test strategy for all test levels
    - Perform initial quality risk analysis
    - Define test metrics
    - Specify the definition of "done"
    - Define exit criteria
    - Create the task board
- Sets the direction for testing throughout sprints

Tester role in Integration

In Agile projects, the objective is to:

- Continuously deliver customer value:
    - In the each sprint
    - At the end of the project
- Continuous testing requires identifying all dependencies between functions and features
    - Integration strategy is helpful for this process
    - Important to identify all dependencies between underlying functions and features

Test Planning

Since testing is fully integrated into the agile team, test planning should start during the release planning session and be updated during each sprint. Planning will include the following tasks:

- What: test scope, extent of testing, test and sprint goals
- Why: explain what's in and out of scope
- Who: assign testing of features and characteristics
- Where: which test environment and what changes, data, tools, and configurations are needed for it (think ahead)
- When: start date and frequency for test tasks
- How: which test methods, techniques, tools, and data
- Prerequisites: predecessor tasks, expertise, training
- Dependencies: functions, code, system components, vendor, technology, tools, activities, tasks, teams, test types, test levels, and constraints
- Project and quality risks: as described in Foundation and later

- Priority: consider customer/user importance, risks, dependencies
- Time: duration and effort required to test, as described later
- Sprint planning populates task board with tasks (1-2 days)

Agile Testing Practices

Just a small list of some agile practices, which may be useful for testers in a scrum team:

- Pairing: Pair testing is a software development technique in which two team members work together at one keyboard to test the software application. One does the testing and the other analyzes or reviews the testing. This can be done between one tester and developer or business analyst or between two testers with both participants taking turns at driving the keyboard.
- Incremental test design: Test cases and charters are gradually built from user stories and other test bases, starting with simple tests and moving toward more complex ones.
- Mind mapping: A mind map is a diagram used to visually organise information. A mind map is often created around a single concept, drawn as an image in the center of a blank landscape page, to which associated representations of ideas such as images, words and parts of words are added. Major ideas are connected directly to the central concept, and other ideas branch out from those. Mind mapping is a useful tool when testing. For example, testers can use mind mapping to identify which test sessions to perform, to show test strategies, and to describe test data.

3.2 Assessing Quality Risks and Estimating Test Effort

Risk based testing is known from both traditional and agile projects. Risk-based testing (RBT) is a type of software testing that functions as an organizational principle used to prioritize the tests of features and functions in software, based on the risk of failure, the function of their importance and likelihood or impact of failure. Moreover, it is to estimate the effort required to minimize those risk to an acceptable level or remove them totally.

To mitigate the risk we use test design techniques. Often we use the same kind of techniques known from traditional testing, but due to the short iterations and the often many changes in agile development lifecycles, some adaptations of those techniques are required.

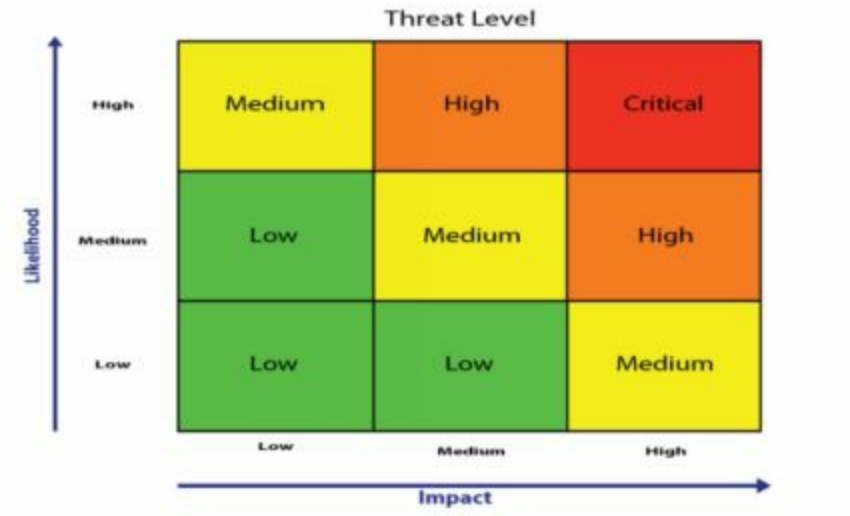
In short Risk based testing is:

- Reduce product (quality) risk to an acceptable level
- Lightweight techniques can be used:
    - Identify quality risks
    - Assess level of risk
    - Estimate test effort
    - Mitigate risks through test design, implementation, and execution
- Short iterations and fast change has implications

3.2.1 Assessing Quality Risks in Agile Projects

Prioritizing, allocation and selection of test condition is one of the biggest challenges within both traditional and agile testing. However, let us have a look at some of the content in assessing Product (Quality) risks:

- Select, allocate, and prioritize test conditions to maximize effectiveness and efficiency
- Product (Quality) risk analysis supports this process
    - Risk: a possible negative outcome
    - Level of risk: based on likelihood and impact
    - Product (Quality) risks: potential problems with product quality
    - Project risks: potential problems for project success
- Agile quality risk analysis occurs:
    - At a high level during release planning by business stakeholders
    - At a detailed level during iteration planning by the whole team
- In each iteration, the tester designs, implements, and executes tests for the risks



Example on a Risk graph based on likelihood and impact

Examples of Product (quality) risks for a system include:

- Incorrect calculations in invoice (a functional risk related to accuracy)
- Slow response to user online payment transaction (a non-functional risk related to efficiency and response time)
- Difficulty in understanding an error message (a non-functional risk related to usability and understand ability)

Example: Difficulty in understanding an error message

Product (Quality) Risk:

- Quality risks include all features and attributes that can affect customer, user, stakeholder satisfaction
    - Incorrect calculations (functional)
    - Slow response time (non-functional performance risk)
    - Confusing interface (non-functional usability risk)
- Risk analysis prioritizes tasks and guides the sizing of the tasks
    - High risks require extensive testing, come earlier, and involve more story points
    - Low risks receive cursory testing, come later, and involve fewer story points
- Risk-based prioritization also includes sprint backlog items

Product (Quality) risk analysis process in an agile project:

1. Gather the Agile team members together, including the tester(s)
2. List all the backlog items for the current iteration (e.g., on a task board)
3. Identify the quality risks associated with each item, considering all relevant quality characteristics
4. Assess each identified risk, which includes two activities: categorizing the risk and determining its level of risk based on the impact and the likelihood of defects
5. Determine the extent of testing proportional to the level of risk.
6. Select the appropriate test technique(s) to mitigate each risk, based on the risk, the level of risk, and the relevant quality characteristic.

The tester and the team:

The tester in the team (can be anyone) then designs, implements, and executes tests to mitigate the risks. Meaning looking at the whole picture to satisfy the customer, User and other stakeholders;

- features,
- behaviour's,
- quality characteristics,
- and attributes

Change can happen during the release or sprint, therefor, the team should remain aware of additional information that may change the set of risks and/or the level of risk associated with known quality risks. From time to time adjustment of the product risk analysis will take place. Adjustments includes:

- identifying new risks,
- re-assessing the level of existing risks,
- and evaluating the effectiveness of risk mitigation activities.

Review up front is an example on how to mitigate product risk before test execution starts.

3.2.2 Estimating Testing Effort Based on Content and Risk

Just as the agile team estimates the functionality development in a user story during release planning, the team also need to address the testing effort:

- Test strategy is defined during release planning
- During iteration planning, user stories are estimated (e.g., via planning poker in story points)
- Story points give implementation effort
- Risk level should influence story points
- Planning poker can be used to reach consensus, involve whole team, and avoid missing anything
- Reliable estimation, including testing, is necessary for smooth work pace and meaningful velocity

Let us have a look at some techniques.

Agile estimation:

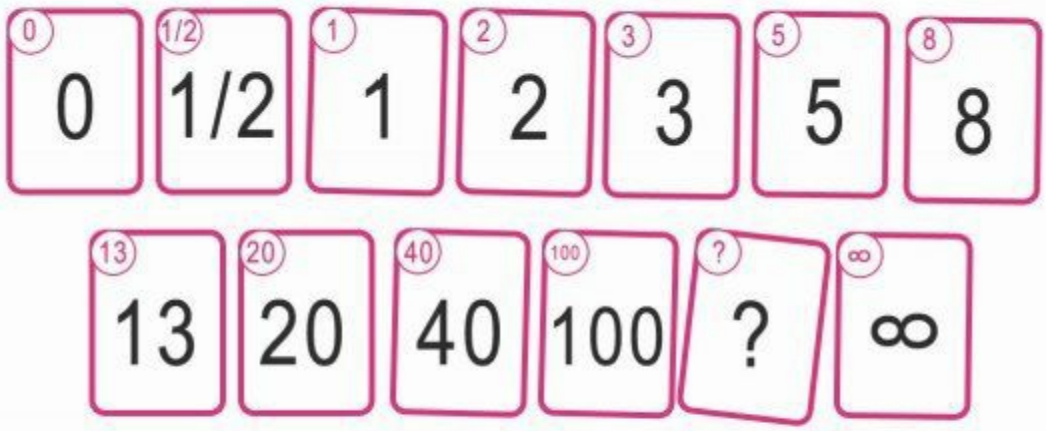
All the entries within the Scrum Product Backlog have to be estimated to allow the Scrum Product Owner to prioritize the entries and to plan releases – This include estimating the test effort.

Planning Poker / Scrum Poker

One commonly used method during the estimation process is to play Planning Poker® (also called Scrum Poker). When using Planning Poker®, influence between the participants are minimized and therefore a more accurate estimation result is produced.

In order to play Planning Poker the following is needed:

- The list of features to be estimated
- Decks of numbered cards.



The Scrum Framework itself does not prescribe a single way for the agile teams to estimate their work. However within the Scrum Framework the estimation is not normally done in terms of time - a more abstracted metric to quantify effort is used. Common estimating methods include numeric sizing (1 through 10), t-shirt sizes (XS, S, M, L, XL, XXL, XXXL) or the Fibonacci sequence (1, 2, 3, 5, 8, 13, 21, 34, etc.).

Important is that the team shares a common understanding of the scale it uses, so that every member of the team is comfortable with it.

A typical deck has cards showing the Fibonacci sequence including a zero: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89; other similar progressions are also possible. The reason for using the Fibonacci sequence is to reflect the uncertainty in estimating larger items. A high estimate usually means that the story is not well understood in detail or should be broken down into multiple smaller stories. Smaller stories can be estimated in greater detail. It would be a waste of time to discuss if it is 19, 20 or 25; the story is simply (too) big.

The game is then played in the following steps:

- The Scrum Product Owner presents the story to be estimated. The Scrum Team asks questions and the Scrum Product Owner explains in more detail. If many stories have to be estimated a time-constraint (e.g. only one minute for explanation) might be set as well. If the time-constraint is reached and the Scrum Team does not understand the story it is a sign that the story has to be re-written.
- Each member of the Scrum Team privately chooses the card representing the estimation.
- After everyone has chosen a card, all selections are revealed.
- People with high and low estimates are allowed to explain their estimate.

- Estimation starts again until a consent is found.
- This game is repeated until all stories are estimated.

In addition, to be take into account when estimate test effort:

- **Extensive**: run large number of tests, both broad and deep, combine and vary interesting conditions, use all relevant techniques with strong coverage criteria
- **Broad**: run medium number of tests, exercise many different interesting conditions, use most relevant techniques with medium coverage criteria
- **Cursory**: run small number of tests, sample most interesting conditions, use efficient techniques with weak coverage criteria
- **Opportunity**: leverage other tests or activities to test 1-2 interesting conditions, investing very little time and effort, using reactive techniques especially

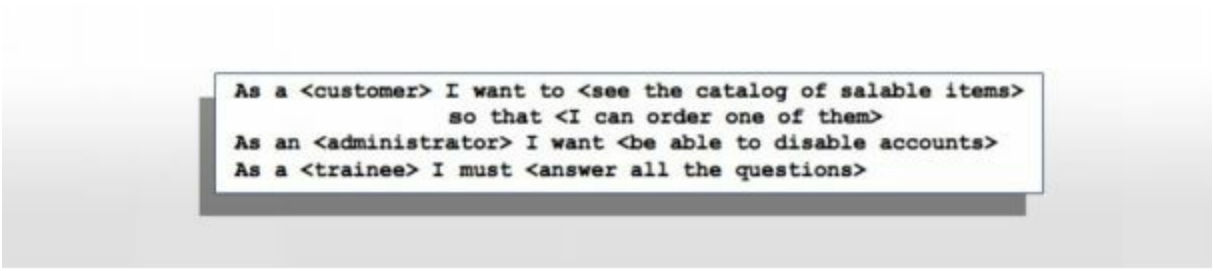
Report bugs only: allocate only a small amount of extra time to report and manage these accidental bugs

3.3 Techniques in Agile Projects

Testing is testing, and most of the techniques and test levels we know from traditional testing may be applied to agile projects as well. In addition, agile projects often use variances in test techniques, terminologies, and documentation.

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

In the simplest definition, the Scrum Product Backlog is simply a list of all things that needs to be done within the project. It replaces the traditional requirements specification artifacts. These items can have a technical nature or can be user-centric e.g. in the form of user stories. Non-functional requirements are sometimes specified in the user stories. Examples of non-functional requirement can be performance - , Volume -, security requirements.



The user story often have an important role as Test Basis, but also look for other test basis to higher you test quality: Test basis in agile projects:

- User stories
- Experience from previous projects or retrospective
- Existing functions, features, and quality characteristics of the system
- Code, architecture, and design
- User profiles as personas
- Information on defects from existing and previous projects
- A categorization of defects in a defect taxonomy
- Any relevant standards or contracts
- User documentation
- Quality risks documentation

During the sprint "iteration", the developers code and implement the functions and features outlined in the user story. In order to be able to decide when an activity from the Sprint Backlog is completed, the Definition of Done (DoD) is used. It is a comprehensive checklist of necessary activities that ensure that only truly done features are delivered, not only in terms of functionality but in terms of quality as well. The DoD may vary from one Scrum Team to another, but must be consistent within one team.

Consider the following criteria:

- Each user story consistent with the others in the iteration
- Aligned with product theme
- Understood by the entire agile team
- Have sufficiently-detailed, testable acceptance criteria
- Card, conversation, and confirmation completed
- User story acceptance tests completed
- Development and test tasks for selected user stories identified, estimated, and within achievable velocity

As a tester, it is important that the user stories are testable and the acceptance "Done" criteria should address the following topics where relevant:

- Externally observable functional behavior
- Relevant quality characteristics, especially non-functional ones
- Steps to achieve goals or tasks (use cases)
- Business rules or procedures relevant to the user story
- Interfaces between system and users, other systems, external data repositories, etc.
- Design and implementation constraints
- Format, types, valid/invalid/default data

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

- How the system is supposed to work and be used
- The system interfaces that can be used/accessed to test the system

- Whether current tool support is sufficient
- Whether the tester has enough knowledge and skill to perform the necessary tests

As a tester, you will often find out that you need other information:

- Testers also need
    - Information about testing interfaces
    - What tools are available to support testing
    - Clarification on system operation and utilization (if test bases unclear)
    - Clear definition of done (shared across team), including test coverage
- Given the lightweight documentation, consider if you have necessary knowledge and skill
- Throughout iteration, information gaps that affect testing will be found
- Testers must work collaboratively with others on the team to resolve those gaps
- Unlike sequential projects, getting relevant information for testing is an ongoing process on agile projects
- Measuring whether a specific test level or activity is done is part of the tester role

Here an example of User story Acceptance Criteria (DoD):

User story:	Acceptance Criteria (DoD):
As a customer, I want to be able to open a popup window that shows the last 30 transactions on my account, with backward/forward arrows allowing me to scroll through transaction history, so that I can see my transaction history without closing the "enter payment amount" window	• Initially populated with 30 most-recent transactions; if no transactions, display "No transaction history yet" • Backward scrolls back 10 transactions; forward scrolls forward 10 transactions • Transaction data retrieved only for current account • Displays within 2 seconds of pressing "show transaction history" control • Backward/forward arrows at bottom • Conforms to corporate UI standard • Can minimize or close pop-up through standard controls at upper right • Properly opens in all supported browsers

Test Levels:

On agile projects, we also need different Test levels and definition of done (DoD) for each of them.

Often we only think about DoD when we talk about User stories, but we have to think about it for Test Levels, User stories, Features, Iterations and Release.

Let us have a look at a list of examples that may be relevant for each of them:

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

	repository • All defects found are reported and possibly fixed • No unresolved major defects (prioritized according to risk and importance)
User Story	• The user stories selected for the iteration are complete, understood by the team, and have detailed, testable acceptance criteria • All the elements of the user story are specified and reviewed, including the user story acceptance tests, have been completed • Tasks necessary to implement and test the selected user stories have been identified and estimated by the team
Feature	• All constituent user stories, with acceptance criteria, are defined and approved by the customer • The design is complete, with no known technical debt • The code is complete, with no known technical debt or unfinished refactoring • Unit tests have been performed and have achieved the defined level of coverage • Integration tests and system tests for the feature have been performed according to the defined coverage criteria • No major defects remain to be corrected • Feature documentation is complete, which may include release notes, user manuals, and on- line help functions
Iteration (Sprint)	• All features for the iteration are ready and individually tested according to the feature level criteria • Any non-critical defects that cannot be fixed within the constraints of the iteration added to the product backlog and prioritized • Integration of all features for the iteration completed and tested • Documentation written, reviewed, and approved
Release	• Coverage: All relevant test basis elements for all contents of the release have been covered by testing. The adequacy of the coverage is determined by what is new or changed its complexity and size, and the associated risks of failure. • Quality: The defect intensity (e.g., how many defects are found per day or per transaction), the

defect density (e.g., the number of defects found compared to the number of user stories, effort, and/or quality attributes), estimated number of remaining defects are within acceptable limits, the consequences of unresolved and remaining defects (e.g., the severity and priority) are understood and acceptable, the residual level of risk associated with each identified quality risk is understood and acceptable.

- Time: If the pre-determined delivery date has been reached, the business considerations associated with releasing and not releasing need to be considered.
- Cost: The estimated lifecycle cost should be used to calculate the return on investment for the delivered system (i.e., the calculated development and maintenance cost should be considerably lower than the expected total sales of the product). The main part of the lifecycle cost often comes from maintenance after the product has been released, due to the number of defects escaping to production.

3.3.2 Applying Acceptance Test-Driven Development

Acceptance test-driven development (ATDD) involves the whole agile team, Developers, Testers and also the Business, and it is a test-first approach done before programming. The test can be manual or automated.

Normally it is following this process:

- Workshop: User stories are analyzed, discussed and written
  - Fixed during the workshop: Any incompleteness, ambiguities or errors in the user story
- Create tests:
  - Can be done together with the whole team, or by individual tester
  - Validation by business representatives

The test should with examples show how to use the system describing the specific characteristics of the user story:

- One or more **positive** path (examples, also called tests)
- One or more negative path
- And examples covering all non-functional attributes (volume, performance, stress, security ...)

The tests are written in clear normal language understandable for all stakeholders, and should contain, is any:

- Preconditions
- The input
- Related output

The test (examples) should cover all functionality and non-functional characteristics that are described in the user story, but should not expand and give examples that are not documented in the user story. Nor should two examples cover the same characteristics of the user story.

3.3.3 Functional and Non-Functional Black Box Test Design

You can use all normal testing techniques on agile projects as well to help the developers and testers to design the tests. Often the techniques are used very early on agile project, before programming start; this is Test Driven Development (TDD) and follows the good testing principles "Early Testing". So when the team and the developers design their test they can apply traditional black box test design techniques like:

- Equivalence partitioning
- Boundary value analysis
- Decision tables
- State transition
- Class tree
- And others

For example, if a user story for banking loan system includes lower and upper limits for the minimum and maximum loan value, test the boundaries (valid and invalid) as well as other equivalence partitions.

Normally non-functional requirements are also documented in the user stories, and black box design techniques can be used to create the tests.

For example, boundary values can be useful to test non-functional requirements: For example, if the system should support 900 concurrent users, test that (and probably beyond)

In the last chapter, we will go through the most basic black box test design techniques. They are good to know for everybody working on agile teams, because they are the foundation of all good testing practices securing good quality in the system deliveries.

3.3.4 Exploratory Testing and Agile Testing

Time and minimal documentation are big factor in agile projects given us tester limited time for test analysis. Therefore, exploratory testing techniques are important. We should combine exploratory test design techniques with other techniques as part of our reactive test strategy, since requirements are never perfect. Some examples on other techniques:

- analytical risk-based testing

- analytical requirements-based testing
- model-based testing
- regression-averse testing

Exploratory Testing and Reactive Strategies

- Exploratory testing and other techniques associated with reactive test strategies are useful in all situations, since requirements are never perfect
- In agile projects, limited documentation and on-going change make these reactive strategies even more useful
- Blend reactive testing with other strategies (e.g., analytical risk-based, analytical requirements-based, regression-averse)
- For exploratory testing:
    - Analysis during iteration planning produces the test condition(s) for the *test charter (more below)* which will guide a test session (60-120 minute) or a test thread (not time-boxed)
    - Test design and test execution occur at the same time, covering the charter, once software is delivered to testers
- Test design can use all dynamic test techniques discussed in Foundation, Advanced Test Analyst, and Advanced Technical Test Analyst, influenced by the results of the previous tests

Test Charter:

Test conditions are shown in a test charter, and may include the following information:

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

Session-based test management

There are different methods to manage exploratory testing, and one of them is session-based test management. The session goes like this:

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

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

- What is most important to find out about the system?
- In what way may the system fail?
- What happens if.....?
- What should happen when.....?
- Are customer needs, requirements, and expectations fulfilled?
- Is the system possible to install (and remove if necessary) in all supported upgrade paths?
- Also, consider heuristics such as boundaries, CRUD (Create, Read, Update, Delete), configuration variations, and possible interruptions
- Utilize all creativity, intuition, ideas, and skills with…
    - The system
    - The business domain
    - The ways people use the software
    - The ways the software fails

Documentation of the process (logging)

It is a common know mistake that exploratory testing should not be documented and logged. Logging the process is important; otherwise, it will not be possible to see how a problem was found in the system. Some documentation is needed, here some example on good useful information:

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

If insufficient logging is done, testing may need to be repeated.

Test logs should be captured and summarized in the relevant test management or task management system. In agile projects, some of the information is often shown on the task board, making it easy for the stakeholders to understand the status for all testing activities.

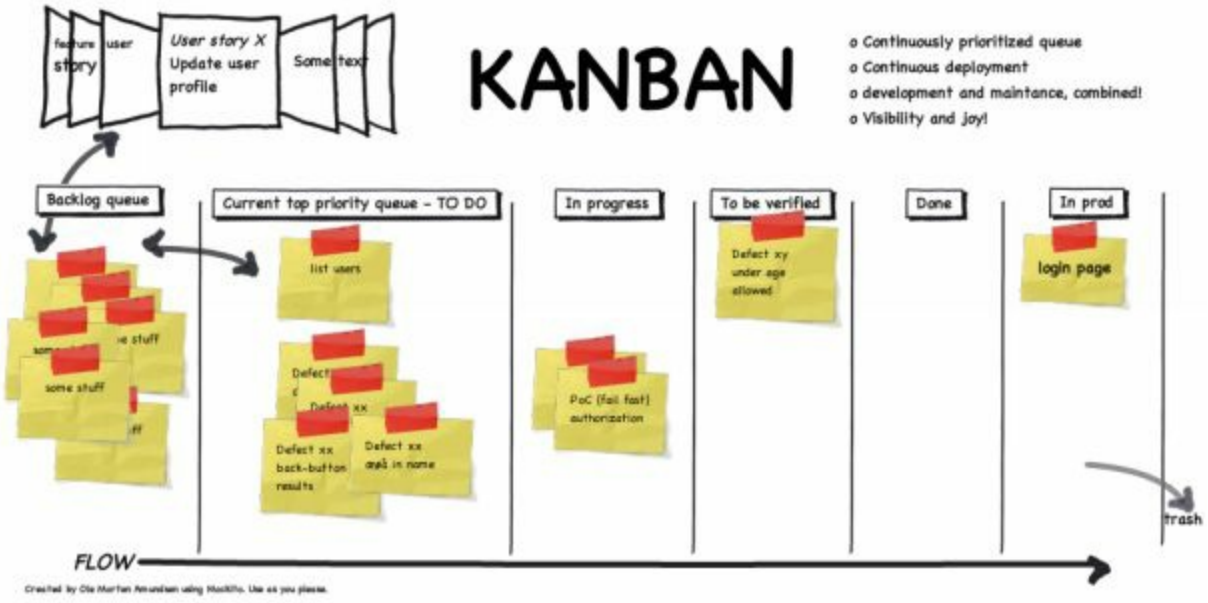
3.3 Tools in Agile Projects

Many different tools is used in Agile Projects, also the tools we know from traditional testing, but some time they are used in a different way. On some agile project, they chose an all-inclusive toolbox (Application lifecycle management or task management solution) which in general help the agile project with tool features that are relevant for them. It can be tools, which include task boards, burndown charts and user stories backlogs. However, traditional tools like test management tools. Requirement tools, and defect management tools are still used on many agile team with good success.

Nevertheless, there are tools that are important in agile teams who do lots of automated test, and that is configuration management tool, which help support of automated test at the belonging test artifacts. Tools to help with team collaboration are also important to make it easy to share information in the agile team, which are key to agile practices.

Let us have a look at some key factors on some of the tools.

3.4.1 Task Management and Tracking Tools



Some teams use physical task boards to manage and track the user stories during the sprint, other agile projects use tools for the support of this, and some team even have electronic task boards connected to an application. However, no matter how fancy you solution the agile team is using it has the following purposes [ISTQB syllabus]:

- Record stories and their relevant development and test tasks, to ensure that nothing gets lost during a sprint
- Capture team members' estimates on their tasks and automatically calculate the effort required to implement a story, to support efficient iteration planning sessions
- Associate development tasks and test tasks with the same story, to provide a complete picture of the team's effort required to implement the story
- Aggregate developer and tester updates to the task status as they complete their work, automatically providing a current calculated snapshot of the status of each story, the iteration, and the overall release
- Provide a visual representation (via metrics, charts, and dashboards) of the current state of each user story, the iteration, and the release, allowing all stakeholders, including people on geographically distributed teams, to quickly check status
- Integrate with configuration management tools, which can allow automated recording of code check-ins and builds against tasks, and, in some cases, automated status updates for tasks

3.4.2 Communication and Information Sharing Tools

Information sharing is very important in agile teams so they can collaborate in an effective way. Agile team use verbal communication, email, and chat solution like (Skype, Messenger, Lync and instant messaging solutions) but also wikis and desktop sharing.

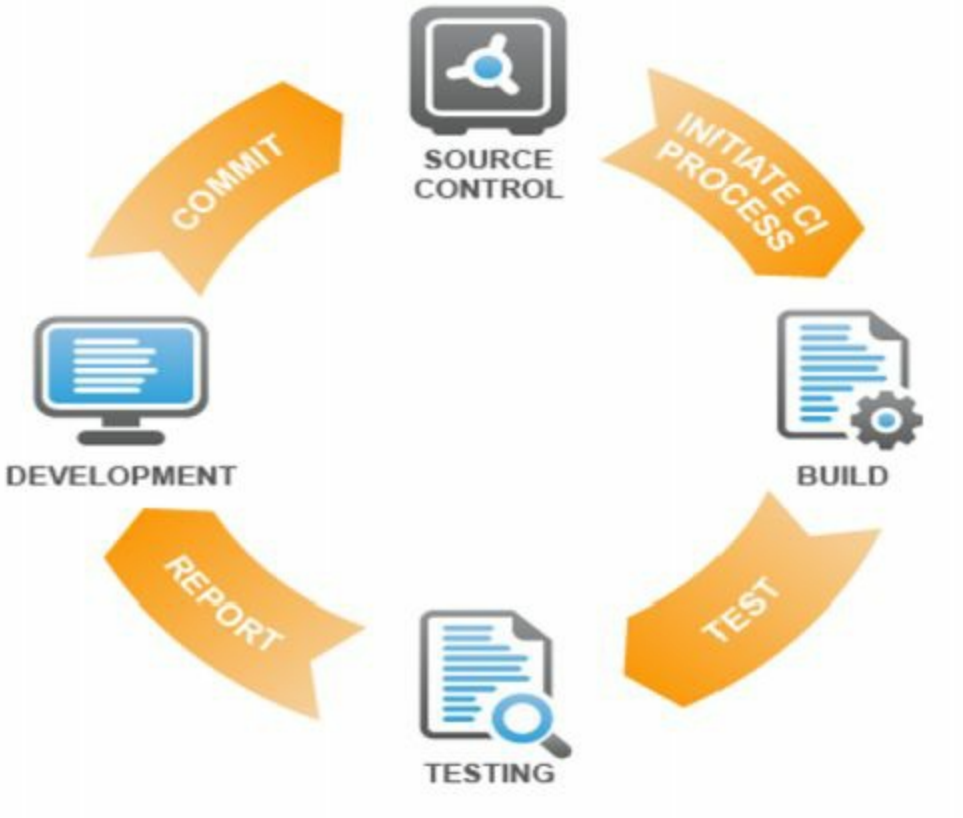
Wikis: used for diagrams, discussions, photos of whiteboard drawings; a scrapbook of useful tools and techniques; metrics, charts, and dashboards on product status; recording conversations.

Instant messaging: used for direct communication between teams, groups, and pairs; support for distributed teams; saving money via VOIP.

Desktop sharing: used for product demonstrations, code reviews, and pairing; recording product demonstrations

3.4.3 Software Build and Distribution Tools

Daily build and deployment are a key factor for success in agile projects, and as we have described earlier as well continuous integrations tools and build distributions tools are used.



- Continuous integration tools provide:
  - Quick feedback code quality
  - Stepwise integration of systems
  - Visibility on build status and history
  - Automatic reporting
- Automatic deployment tools provide:
  - Locate and install appropriate build in test environment
  - Reduce errors and delays of hand installation

3.4.4 Configuration Management Tool

Configuration management (CM) is a systems engineering process for establishing and maintaining consistency of a product's performance, functional and physical attributes with its requirements, design and operational information throughout its life. For this process, we use:

- Configuration management tools:
  - Store source code, automated tests, manual tests, and other test work products
  - Provide traceability between which versions of software and the tests used
  - Allow rapid change while saving historical information

3.4.5 Test Design, Implementation, and Execution Tools

Tool used by agile testers for these purpose includes:

Mind maps for test design tools: This tools help organizing a mind map. A mind map is a diagram used to visually organize information. A mind map is often created around a single concept, drawn as an image in the center of a blank landscape page, to which associated representations of ideas such as

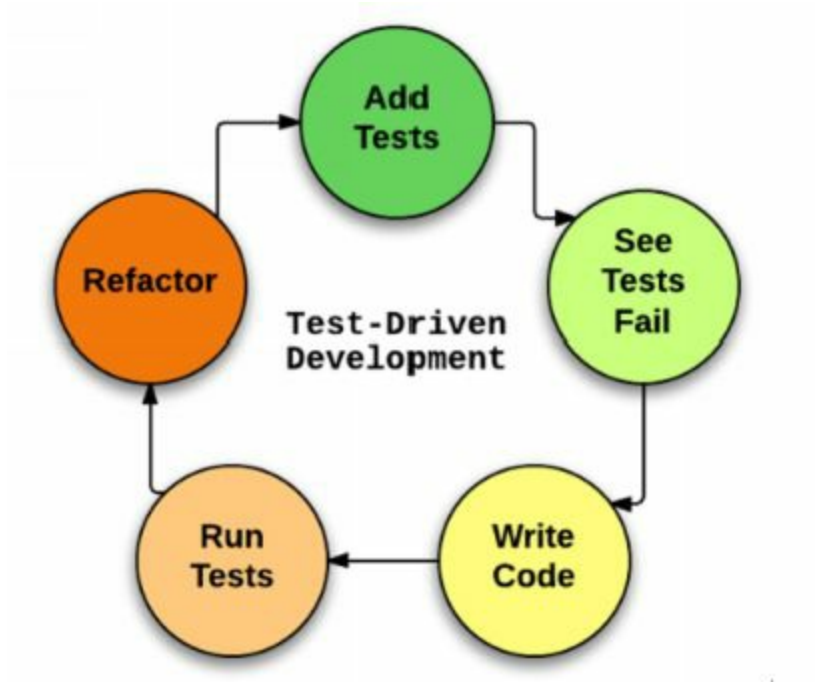
images, words and parts of words are added. Major ideas are connected directly to the central concept, and other ideas branch out from those. This can be a very useful in connection to exploratory testing.

Application lifecycle management or task management for test case management tools: These tools the tester use test case management, and can be part of the whole team's management toolbox.

Application lifecycle management (ALM), called also ADLM (Application Development Life-cycle Management), is the product lifecycle management (governance, development, and maintenance) of application software – In agile we will see lightweight solutions for this in the tools. It encompasses requirements management, software architecture, computer programming, software testing, and software maintenance, change management, continuous integration, project management, and release management.

Test data preparation, generation, load, validation, and tools for making data anonym:
These tools generate data for testing purpose and in agile projects that is important because you test and build daily, and the life cycle for the iteration is normal short. Therefore, the team need good tools, helping them with a never-ending data generation tasks.

Automated test execution tool: These tools support behavior-driven development, test-driven development, and acceptance test-driven development.



Exploratory test tool: These tools capture and log activities performed on an application during an exploratory test session are beneficial to the tester and developer, as they record the actions taken.

Agile teams can use many other tools for test design, implementation and execution, and remember many open-source options are available.

3.4.6 Cloud Computing and Virtualization Tools

These tools are used for:

- Virtualization can increase the test resources available
- Setting up a new test environment can be quick and sometimes cheap
- Known-good configurations can be saved prior to installing a new release

In addition to server virtualization, service virtualization, client virtualization, and data virtualization can be useful for testers

References

[Aalst13] Leo van der Aalst and Cecile Davis, “TMap NEXT® in Scrum,” ICT-Books.com, 2013.

[Adzic09] Gojko Adzic, “Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing,” Neuri Limited, 2009.

[Anderson13] David Anderson, “Kanban: Successful Evolutionary Change for Your Technology Business,” Blue Hole Press, 2010.

[Beck02] Kent Beck, “Test-driven Development: By Example,” Addison-Wesley Professional, 2002.

Beck04] Kent Beck and Cynthia Andres, “Extreme Programming Explained: Embrace Change, 2e” Addison-Wesley Professional, 2004.

[Black07] Rex Black, “Pragmatic Software Testing,” John Wiley and Sons, 2007. [Black09] Rex Black, “Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, 3e,” Wiley, 2009.

[Chelimsky10] David Chelimsky et al, “The RSpec Book: Behavior Driven Development with Rspec, Cucumber, and Friends,” Pragmatic Bookshelf, 2010.

[Cohn04] Mike Cohn, “User Stories Applied: For Agile Software Development,” Addison-Wesley Professional, 2004.

[Goucher09] Adam Goucher and Tim Reilly, editors, “Beautiful Testing: Leading Professionals Reveal How They Improve Software,” O'Reilly Media, 2009.

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

[Jones11] Capers Jones and Olivier Bonsignour, “The Economics of Software Quality,” AddisonWesley Professional, 2011.

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

[Schwaber01] Ken Schwaber and Mike Beedle, “Agile Software Development with Scrum,” Prentice Hall, 2001.

[Steen01] Steen Lerche-Jensen, “Scrum Master”, Amazon Kindle, 2014

[Steen02] Steen Lerche-Jensen, “Product Owner”, Amazon Kindle, 2014

[vanVeenendaal12] Erik van Veenendaal, “The PRISMA approach”, Uitgeverij Tutein Nolthenius, 2012.

[Wiegers13] Karl Wiegers and Joy Beatty, “Software Requirements, 3e,” Microsoft Press, 2013.