ISTQB Agile Tester

Table of Contents

1 Agile methodology

Agile software development methodology is an process for developing software (like other software development methodologies – Waterfall model, V-Model, Iterative model etc.) However, Agile methodology differs significantly from other methodologies. In English, Agile means 'ability to move quickly and easily' and responding swiftly to change – this is a key aspect of Agile software development as well.

1.1 Brief overview of Agile Methodology

- In traditional software development methodologies like Waterfall model, a project can take several months or years to complete and the customer may not get to see the end product until the completion of the project.
- At a high level, non-Agile projects allocate extensive periods of time for Requirements gathering, design, development, testing and UAT, before finally deploying the project.
- In contrast to this, Agile projects have Sprints or iterations which are shorter in duration (Sprints/iterations can vary from 2 weeks to 2 months) during which pre-determined features are developed and delivered.
- Agile projects can have one or more iterations and deliver the complete product at the end of the final iteration.

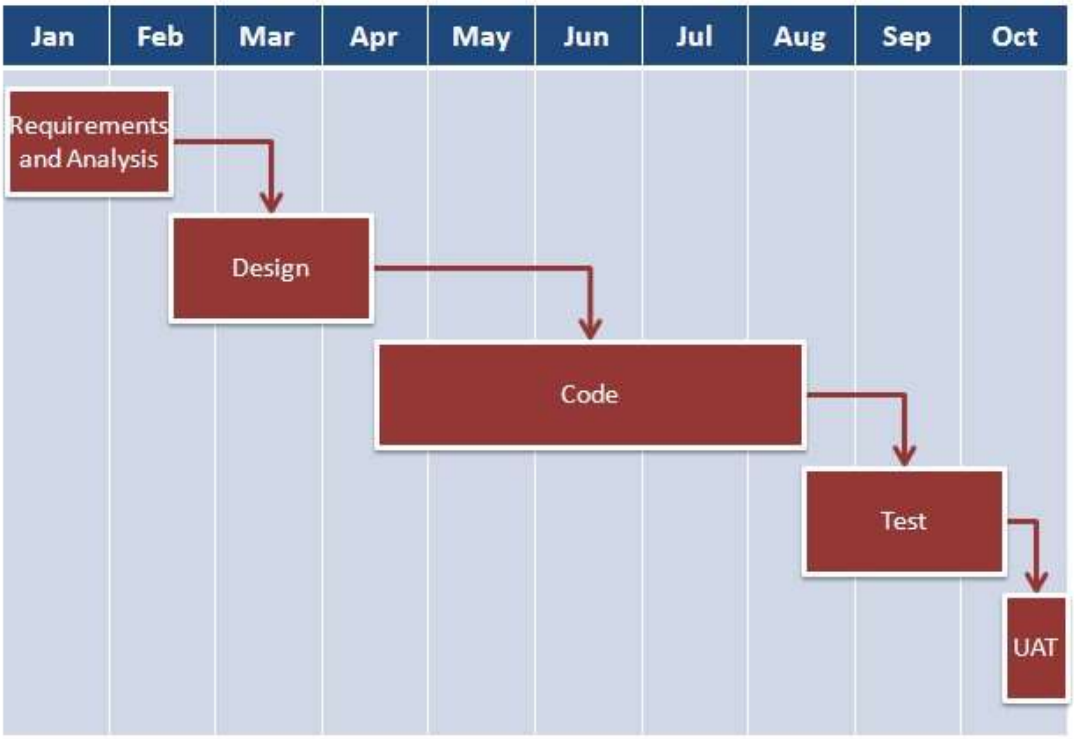
1.2 Example of Agile software development

Example: Google is working on project to come up with a competing product for MS Word, which provides all the features provided by MS Word and any other features requested by the marketing team. The final product needs to be ready in 10 months of time. Let us see how this project is executed in traditional and Agile methodologies.

In traditional Waterfall model -

- At a high level, the project teams would spend 15% of their time on gathering requirements and analysis (1.5 months)
- 20% of their time on design (2 months)
- 40% on coding (4 months) and unit testing
- 20% on System and Integration testing (2 months).
- At the end of this cycle, the project may also have 2 weeks of User Acceptance testing by marketing teams.
- In this approach, the customer does not get to see the end product until the end of the project, when it becomes too late to make significant changes.

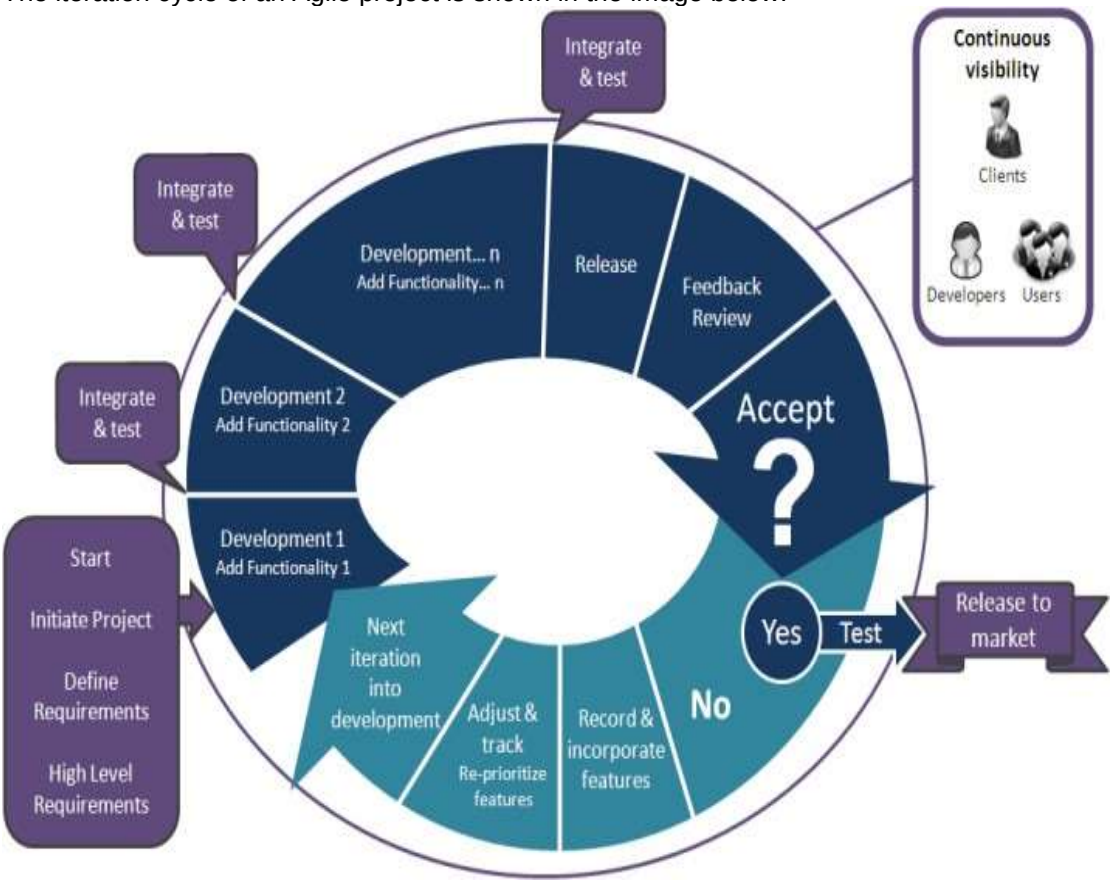
The image below shows how these activities align with the project schedule in traditional software development.

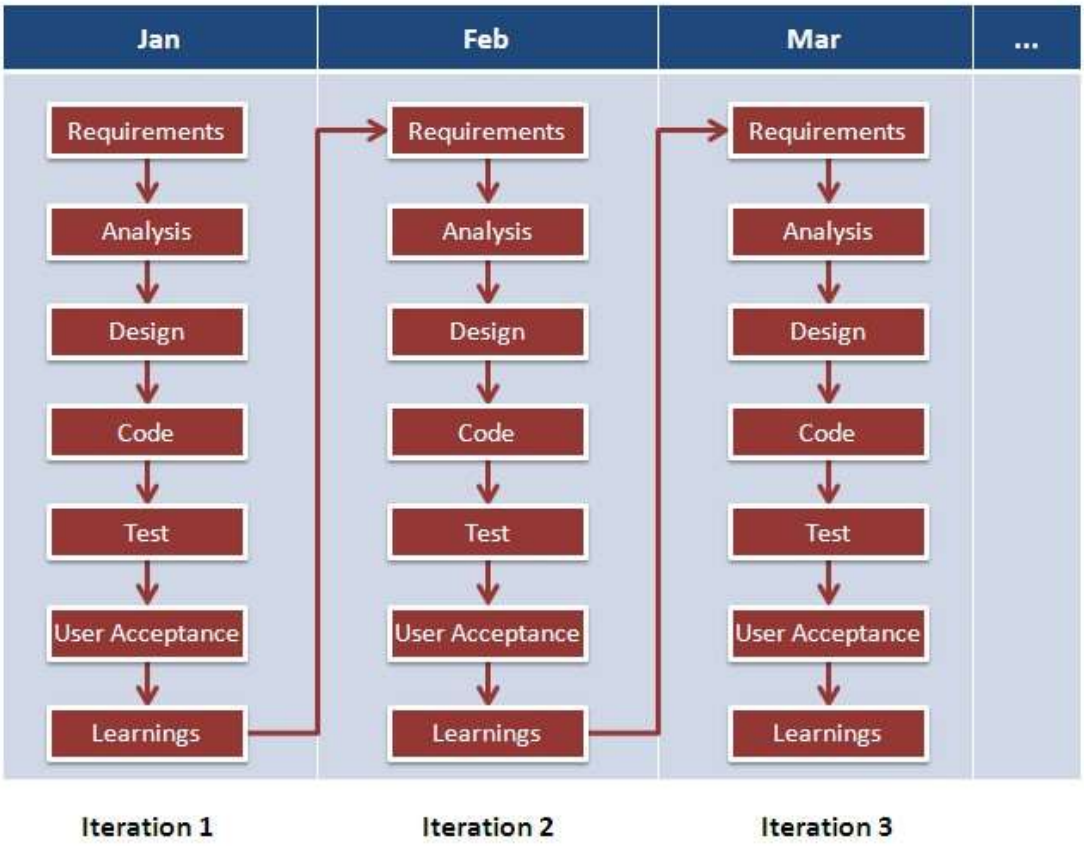


With **Agile development** methodology –

- In the Agile approach, each project is broken up into several 'Iterations'.
- All Iterations should be of the same time duration (between 2 to 8 weeks).
- At the end of each iteration, a working product should be delivered.
- In simple terms, in the Agile approach the project will be broken up into 10 releases (assuming each iteration is set to last 4 weeks).
- Rather than spending 1.5 months on requirements gathering, in Agile software development, the team will decide the basic core features that are required in the product and decide which of these features can be developed in the first iteration.
- Any remaining features that cannot be delivered in the first iteration will be taken up in the next iteration or subsequent iterations, based on priority.
- At the end of the first iterations, the team will deliver a working software with the features that were finalized for that iteration.
- There will be 10 iterations and at the end of each iteration the customer is delivered a working software that is incrementally enhanced and updated with the features that were shortlisted for that iteration.

The iteration cycle of an Agile project is shown in the image below.





This approach allows the customer to interact and work with functioning software at the end of each iteration and provide feedback on it. This approach allows teams to take up changes more easily and make course corrections if needed. In the Agile approach, software is developed and released incrementally in the iterations. An example of how software may evolve through iterations is shown in the image below.

Agile methodology gives more importance to collaboration within the team, collaboration with the customer, responding to change and delivering working software.

Agile development has become common place in IT industry. In a recent survey over 52% of respondents said that their company practiced Agile development in one form or another. Irrespective of your role in the organization, it has become essential to understand how Agile development works and how it differs from other forms of software development.

In traditional approach each job function does its job and hands over to the next job function. The previous job functions have to signoff before it is handed over the next job function authenticating that the job is full and complete in all aspects. For example, Requirement gathering is completed and handed over to design phase and it is subsequently handed over to development and later to testing and rework. Each job function is a phase by itself.

In Agile way of working, each feature is completed in terms of design, development, code, testing and rework, before the feature is called done. There are no separate phases and all the work is done in single phase only.

1.3 Difference between Agile model and Non-Agile models

Parameters	Agile Model	Non-Agile Models
Approach of this methodology	This methodology is very flexible and adjustable and can adapt to the project needs.	This methodology is not as flexible as Agile model and it's tough to accommodate changes in the project.
Measurement of Success	The success of the project in Agile model is measured by the Business value delivered.	In this methodology the success of the project is measured by the Conformation to plan.
Size of the Project	The Project size is small in Agile model.	The project size is Large in non- Agile models.
Style of Management	The style of management in Agile model is not centralized. It is distributed among the team members.	The management style in the non-Agile models is dictatorial. Only one person is the decision maker and rest of the people follows him.
Ability to adapt to change	In Agile model the changes are accepted and adapted as per the project needs.	But in non-Agile models the changes are not accepted easily in the later stages of the development.
Documentation required	Less documentation is required in Agile.	More documentation is required in non-Agile models.

Importance of	In Agile model more emphasis is given to the people that means it's People- Oriented.	In non-Agile models the more importance is given to the process hence it's Process- Oreinted.
Cycles or iterations	Agile methodology has many cycles or iterations which is also known as Sprints.	But, in Non-Agile methodology the cycles are limited.
Planning in Advance	There is minimal upfront planning in Agile methodology.	In Non-Agile models the planning should be complete before the development starts.
Revenue	In Agile method the return on investment is early in the project cycle.	In non-Agile methods the return on investment is at the end of the project.
Size of the team	The size of the team in Agile methodology is usually small and creative.	But in Non-Agile models the team size is large.

1.4 The Fundamentals of Agile Software Development

Project Attributes	Agile Model	Non-Agile Model
Requirement of the Project	Requirements in Agile model can change as per the customer requirement. Sometimes requirements are not very clear.	In Non-Agile models the requirements are very clear before entering into the development phases. Any change in the requirement is not easily accepted during the development phases.
Size of the Project	The Project size is small in Agile model hence small team is required.	But in Non-Agile models the Project size is usually big hence big team is required.
Design of the Project	In Agile model the architecture is made as per the current requirements.	In Non-Agile models the architecture is made as per the current requirements as well as for future requirements.
Planning and Control of the Project	In Agile model the planning of the project is Internalized and has qualitative control.	But in Non-Agile models the plans are documented properly and have quantitative control.
Type of Customers	Agile methodology is followed by the collaborated, dedicated collated and knowledgeable customers.	In Non-Agile models the customers are of Contract provisions.
Developers required	In Agile model the developers should be knowledgeable, analytically strong, collated and collaborative.	In Non-Agile models the developers should be more Plan Oriented.
Refactoring	In Agile model refactoring is not costly.	But in Non-Agile models the refactoring is very costly.
Risks involved	Usually in Agile models the chances of occurrence of unknown risks are more which can have major impact in the project.	In Non-Agile models the risks are understood clearly and the impact of the risk in the project is very less.

1.5 Advantages of Agile Methodology

- ✓ *In Agile methodology the delivery of software is unremitting.*

- ✓ *The customers are satisfied because after every Sprint working feature of the software is delivered to them.*
- ✓ *Customers can have a look of the working feature which fulfilled their expectations.*
- ✓ *If the customer has any feedback or any change in the feature then it can be accommodated in the current release of the product.*
- ✓ *In Agile methodology the daily interactions are required between the business people and the developers.*
- ✓ *In this methodology attention is paid to the good design of the product.*
- ✓ *Changes in the requirements are accepted even in the later stages of the development.*

1.6 Disadvantages of the Agile Methodology

- *In Agile methodology the documentation is less.*
- *Sometimes in Agile methodology the requirement is not very clear hence it's difficult to predict the expected result.*
- *In few of the projects at the starting of the software development life cycle it's difficult to estimate the actual effort required.*
- *The projects following the Agile methodology may have to face some unknown risks which can affect the development of the project.*

2 Agile software development

2.1 The Fundamentals of Agile Software Development

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

Testers in agile team are called as team members, not identified by his/her skill specialization. In fact, testers and developers are together are called as Development team. The word "Development team" not only contains developers, but also testers who are actively involved in the development of the product increment. The testers form part of fully cross functional Agile team where they work with business representatives and developers in tandem. The team members of an Agile team communicate more frequently and informally to develop product with good quality.

2.1.1 Agile Software Development and the Agile Manifesto

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

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

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

Individuals and Interactions

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

People form the core of Agile software development and teamwork is given high importance than using a specific processes and tools. Any processes and tools should be used as a enabler to increase the team work rather than replacing it. Communication has also plays a vital role in enhancing the Teamwork.

Working Software

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

It has been a practice in Waterfall to deliver design documents, architecture documents or test plans, test cases etc to the customer before we deliver the real piece of working software.

- However, in Agile way of working, this practice should be substituted with delivering piece of working software increment every iteration.
- It doesn't mean that we have to avoid documentation completely; but only necessary documentation is produced.

- The valuable software should be delivered early and frequently during the development.
- Agile development in a way helps in reducing the time taken for the product/software to reach the market giving scope for frequent feedback.

Customer Collaboration

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

Customers often do not know what they really want until they see something working.

- Even more, those days are gone when customers gave specifications in thick binders.
- In a rapidly changing era, upfront requirement specification seems to be a big challenge.
- Frequent customer collaboration may help the development team to understand the needs of the customer in greater detail.
- Writing contracts with the customers is a profit motive and collaboration is a purpose motive.
- Purpose motive should be given more importance than profit motive.

Responding to Change

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

Change is constant and unavoidable in software development projects.

- Most of the software organizations need to respond to changing business needs and regulation amendments for their survival.
- In the current market scenario of digital world, the business priorities change at a faster pace frequently.
- It is predominantly important that software development should adopt to the changing needs to meet the business goals.
- The flexibility to accommodate changes into the plan is more significant than just writing a plan and following it.

Principles

The core Agile Manifesto values are captured in twelve principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, at intervals of between a few weeks to a few months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.

- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

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

2.1.2 Whole-Team Approach

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

- A Product Owner who is the customer representative is also part of the team.
- The Agile teams are small usually not exceeding 7 team members plus or minus 2 team members, stable and cross functional.
- Everyone in the team is seated at same physical location, ideally no barriers between them to enable smooth communication.
- All the Scrum ceremonies are attended by everyone where the progress of the assigned task is discussed and any potential road blocks are raised.
- In this way the whole-team approach helps in building healthier and stronger teams for improving team bonding and synergy.

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

- ✓ *Enhancing communication and collaboration within the team*
- ✓ *Helps team in building strong working relationships through effective cooperation, teamwork and communication.*
- ✓ *Enabling the various skill sets within the team to be leveraged to the benefit of the project*
- ✓ *Helps team members to learn and share knowledge from each other.*
- ✓ *Making quality everyone's responsibility*
- ✓ *Making everyone collectively responsible for the outcome*

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

The developers and testers can often have informal meetings to work out the strategy to create a new product increment. The whole team is involved in any consultations or meetings in which product features are presented, analysed, or estimated. The concept of involving testers, developers, and business representatives in all feature discussions is known as the power of three.

For example: Testers can log defects in a tracking tool and collaborate with developers to replicate it and get it fixed in the same iteration, which will always add more value to the product quality, rather than testers just think that their only job is to just raise defects.

2.1.3 Early and Frequent Feedback

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

During Waterfall development, each activity like design, development, testing is considered as a phase, but in agile all the activities is done in small chunks every iteration.

- In waterfall, the customer can see the software working at the end of the project, and at that stage the changes are very costly and involve significant rework.
- Instead, if the feedback is obtained at the end of every iteration, it may be very easy for the team to make up the feedback and go along.
- It also serves as an important tool to adopt modifications in the application.
- Highest business value features are delivered first to the customer by development teams through the use of early and frequent feedback.

It also in a way helps the team to measure its own capacity so that they do not grossly over commit thus building high degree of transparency into the planning process.

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

The benefits of early and frequent feedback include:

- ✓ *Avoiding requirements misunderstandings, which may not have been detected until later in the development cycle when they are more expensive to fix.*
- ✓ *Customer availability for any questions to the team makes the product development robust, so that team exactly build what customer wants.*
- ✓ *Clarifying customer feature requests, making them available for customer use early. This way, the product better reflects what the customer wants.*
- ✓ *Discovering (via continuous integration), isolating, and resolving quality problems early.*
- ✓ *Providing information to the Agile team regarding its productivity and ability to deliver.*
- ✓ *Sharing agile productivity metrics helps the team to understand the gaps better so that they can find ways to improve themselves.*

- ✓ *Promoting consistent project momentum. The team will be able to deliver at a constant and sustainable pace.*

2.2 Aspects of Agile Approaches

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

2.2.1 Agile Software Development Approaches

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

Extreme Programming

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

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

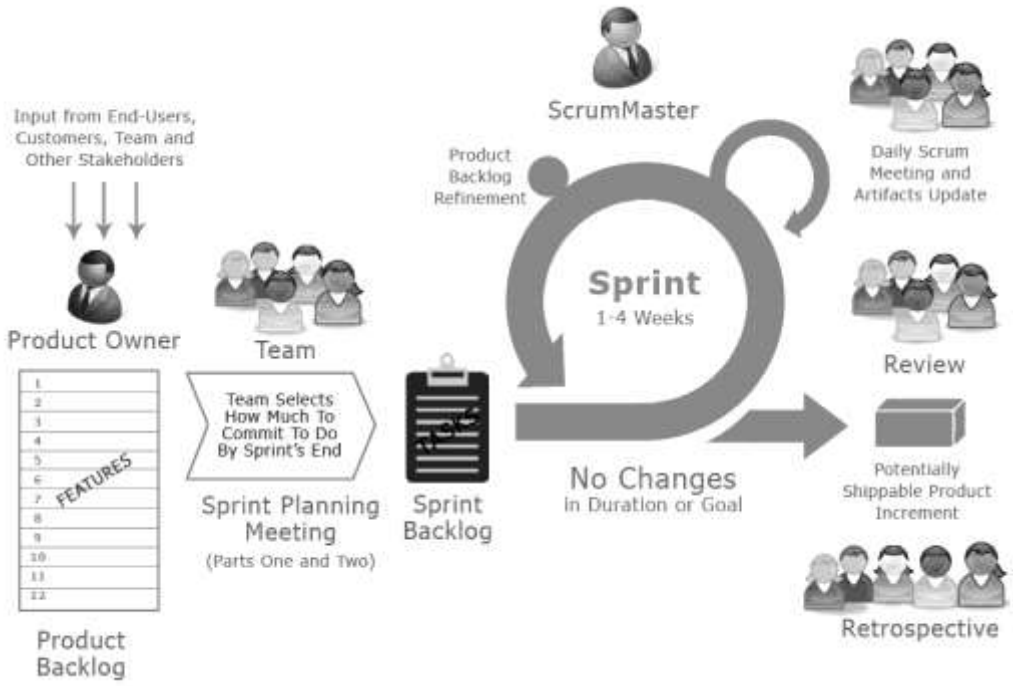
XP describes a set of principles as additional guidelines: humanity, economics, mutual benefit, self-similarity, improvement, diversity, reflection, flow, opportunity, redundancy, failure, quality, baby steps, and accepted responsibility.

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

Many of the Agile software development approaches in use today are influenced by XP and its values and principles. For example, Scrum team often reuses Extreme Programming engineering practices to build quality into the product increment.

Scrum

Scrum is another Agile Development framework and has several key features that are shown the Figure given below and explained in detail.



It contains the following constituent instruments and practices:

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

Scrum defines three roles:

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

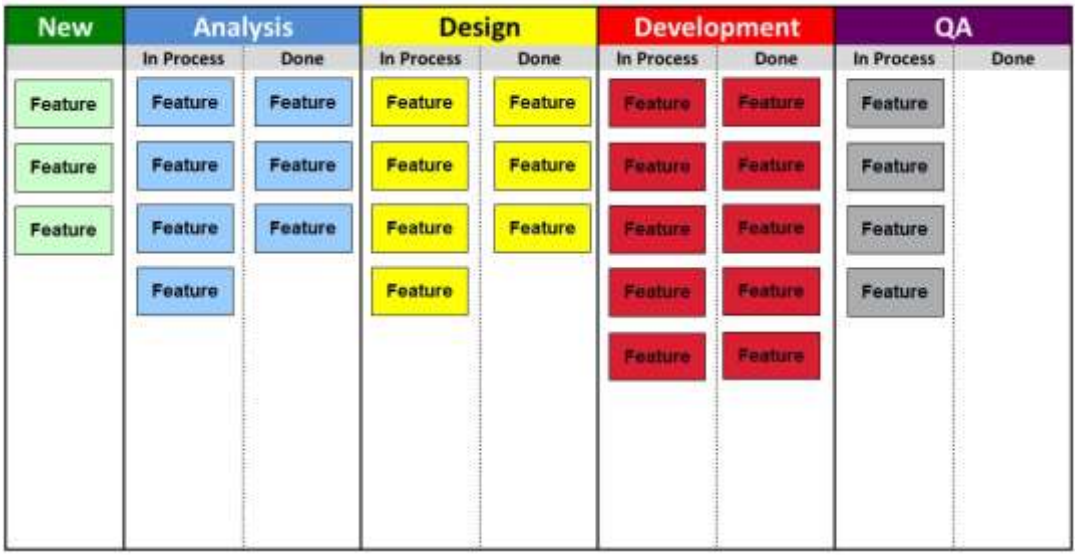
Scrum (as opposed to XP) does not dictate specific software development techniques (e.g., test first programming). In addition, Scrum does not provide guidance on how testing has to be done in a Scrum project.

Kanban

Kanban is a management approach that is sometimes used in Agile projects. The word 'Kan' means Visual 'ban' means Card. So the meaning of Kanban translates into Visual Card, which refers to a signal card which is emerging as part of Lean Software Development. The primary purpose of Kanban is to visualize and optimize the flow of work within a value-added chain thereby reducing the cycle time of delivering fully completed features.

Kanban utilizes three instruments:

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



Kanban board and its states

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

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

2.2.2 Collaborative User Story Creation

In Agile Software development, requirements are captured from the point of value of the user. There are usually several actors that are acting on the system called User Personas (example: User, Admin, Customer, Supplier etc). It should not be written from a developer or tester or manager's perspective. The Product Owner along with team usually writes user stories and the Product Owner explains the User Story to the development team, and clarifies any questions the team might have.

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

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

User stories can be written for functional characteristics with acceptance criteria which should be initially written by the Product Owner and later reviewed along with the team. User stories can be also written for technical stories which indirectly derive a value to the functional user stories (example: Java Update, Hibernate Upgrade etc).

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

The tester can also be part of the team discussion along with the Product Owner and help team to bring up any missing points. He also take help of Product Owner to review the test cases he writes for specific story to validate his understanding.

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

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

Example of a User Story with acceptance criteria is given below:

Story: As a customer, **I would like to** have an email sent to my normal email address **when** my account goes into overdraft so that I know that I need to put money into my account.

Acceptance criteria: When and account goes into overdraft, the system will issue a secure message to the user's account, then the system will:

1. Check to determine if the user has a valid email account in their file
   - If there is a valid email address on file, the system will send a message to that address indicating that a message is available
   - If there is not an email address on file, the system will flag the account as incomplete
2. The message sent to the customer's email address will match the text provided by marketing
3. The email sent by the system will be formatted for HTML viewing and will match the graphic design specifications sent by marketing
4. The email will contain a hyperlink which allows the user to jump instantly to the online banking site

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

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

- **Conversation**: The conversation explains how the software will be used. The conversation can be documented or verbal. Testers, having a different point of view than developers and business representatives, bring valuable input to the exchange of thoughts, opinions, and experiences. Conversation begins during the release-planning phase and continues when the story is scheduled.

- **Confirmation**: The acceptance criteria, discussed in the conversation, are used to confirm that the story is done. These acceptance criteria may span multiple user stories. Both positive and negative tests should be used to cover the criteria. During confirmation, various participants play the role of a tester. These can include developers as well as specialists focused on performance, security, interoperability, and other quality characteristics. To confirm a story as done, the defined acceptance criteria should be tested and shown to be satisfied.

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

2.2.3 Retrospectives

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

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

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

Retrospectives ideally may be held at the same place and same location every iteration. It is highly recommended that Scrum Master, Product Owner and the team only attend the retrospective. Managers and senior management may be barred for entering a retro meeting; the team may not speak up due to inherent fear of management.

Sprint Retrospective meetings can be facilitated by asking each person in the team to answer a variation on the above questions. Instead of asking what went well, what didn't go well some teams prefer -

- What should we start doing
- What should we stop doing
- What should we continue to do

Teams are asked to be specific in their answers so that effective actions can be taken. The retrospective meeting is usually conducted immediately after the Sprint review meeting. It is recommended that the entire team participate in this exercise so that any issues or concerns that the teams face during the previous Sprint are addressed during the teaming and avoided in upcoming Sprints.

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

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

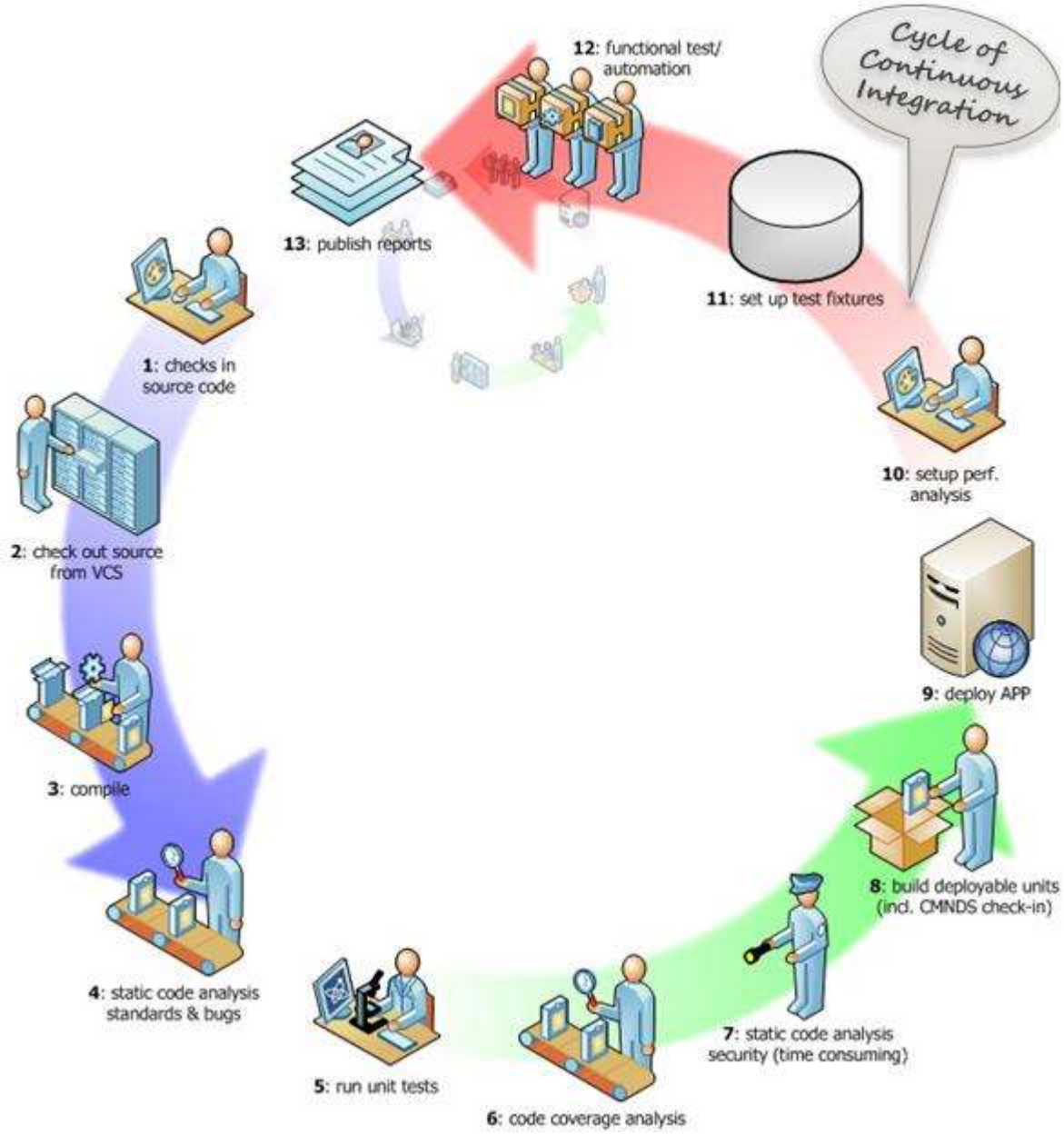
2.2.4 Continuous Integration

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

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

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

The Cycle of CI has been shown below in the figure below



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

CI helps testers to perform automation effectively to uncover defects at a faster rate and improve the regression results. The automation will also give the percentage of test coverage area. The automation report also covers the number of user stories and functionality mapped to the product increment. Automation also reduces the manual testing effort of the testers and makes their life easy. Testers can also manually test the areas of failure and collaborate with the developers to fix the defects.

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

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

There are several tools that are being used in organizations as a part of continuous integration to build the automation scripts. Few examples of the tools are JUnit, Selenium, SONAR etc. CI automation will reduce the time to deliver, since it replaces the traditional manual testing. Build tools are linked to automation server and build server and deploy to testing, staging and production environments. Organizations started using these build tools to replace the traditional quality control.

Continuous integration can provide the following benefits:

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

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

- Continuous integration tools have to be introduced and maintained
- The continuous integration process must be defined and established
- Test automation requires additional resources and can be complex to establish
- Thorough test coverage is essential to achieve automated testing advantages
- Teams sometimes over-rely on unit tests and perform too little system and acceptance testing

Continuous integration requires the use of tools, including tools for testing, tools for automating the build process, and tools for version control. So in summary, CI is advantageous to a project team, if the required tools are put in place for automating the build process.

2.2.5 Release and Iteration Planning

There are 3 levels of planning in Agile. They are Release Planning, Iteration Planning and Daily Planning. These planning meetings help the Scrum Master, Product Owner and the rest of the team in understanding how the product will be delivered, the complexity involved and their day to day responsibility in the delivery of the product, among other things.

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

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

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

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

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

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

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

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

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

Release plans may change as the project proceeds, including changes to individual user stories in the product backlog. These changes may be triggered by internal or external factors. Internal factors include delivery capabilities, velocity, and technical issues. External factors include the discovery of new markets and opportunities, new competitors, or business threats that may change release objectives and/or target dates. In addition, iteration plans may change during an iteration. For example, a particular user story that was considered relatively simple during estimation might prove more complex than expected.

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

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

- The scope of testing, the extent of testing for those areas in scope, the test goals, and the reasons for these decisions.
- The team members who will carry out the test activities.

- The test environment and test data needed, when they are needed, and whether any additions or changes to the test environment and/or data will occur prior to or during the project.
- The timing, sequencing, dependencies, and prerequisites for the functional and non-functional test activities (e.g., how frequently to run regression tests, which features depend on other features or test data, etc.), including how the test activities relate to and depend on development activities.
- The project and quality risks to be addressed.

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

Daily Planning meeting in agile are also called as ‘Stand up meetings’ where the development team and testing team meets daily to discuss ‘What progress in the assigned task they made yesterday’, ‘What tasks they have planned to do today and what tasks they will do tomorrow’. In addition to these details each member has to tell that how long they are going to take to complete the assigned task within the sprint. If any remaining or pending task of any member of the team is not completed in that particular sprint then they notify such details in the daily meetings. Accordingly the pending task(s) from the previous sprint are taken care in the next sprints. Likewise if there are any changes in the requirement from the customers or any build issues or any blocking issues then such details are also discussed in the meetings.

3 Fundamental Agile Testing Principles, Practices, and Processes

3.1 The Differences between Testing in Traditional and Agile Approaches

Test activities are related to development activities, and thus testing varies in different lifecycles.

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

Testers should remember that organizations vary considerably in their implementation of lifecycles. Deviation from the ideals of Agile lifecycles 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.

3.1.1 Testing and Development Activities

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

Examples of iteration goals in an eCommerce application:

- Implement payment gateway functionality to accept Master card and Visa card and reject other cards.
- Develop the order checkout process: pay for an order, cancel shipment, pick shipping, gift wrapping, etc

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

In some cases, hardening or stabilization iterations occur periodically to resolve any lingering defects and other forms of technical debt. However, the best practice is that no feature is considered done until it has been integrated and tested with the system. Another good practice is to address defects remaining from the previous iteration at the beginning of the next iteration, as part of the backlog for that iteration (referred to as “fix bugs first”).

However, some complain that this practice results in a situation where the total work to be done in the iteration is unknown and it will be more difficult to estimate when the remaining features can be done. At the end of the sequence of iterations, there can be a set of release activities to get the software ready for delivery, though in some cases delivery occurs at the end of each iteration.

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

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

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

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

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

3.1.2 Project Work Products

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

I. Business-oriented work products that describe what is needed (e.g., requirements specifications) and how to use it (e.g., user documentation)
II. 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)
III. Test work products that describe how the system is tested (e.g., test strategies and plans), that actually test the system (e.g., manual and automated tests), or that present test results (e.g., test dashboards)

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

Typical business-oriented work products on Agile projects include user stories and acceptance criteria. User stories are the Agile form of requirements specifications, and should explain how the system should behave with respect to a single, coherent feature or function.

A user story should define a feature small enough to be completed in a single iteration. Larger collections of related features, or a collection of sub-features that make up a single complex

feature, may be referred to as “epics”. Epics may include user stories for different development teams. For example, one user story can describe what is required at the API-level (middleware) while another story describes what is needed at the UI-level (application). These collections may be developed over a series of sprints. Each epic and its user stories should have associated acceptance criteria.

Examples of Epics are:

1. As an Amazon customer, I would like to be able to order a new blender and have it shipped to my house because my blender broke.
2. As a customer of ABC auto insurance company, I would like to be able to report the fact that my neighbor’s son hit my truck with their car so that I can get the claim process started.

Looking at the Epics above, they can be easily broken down into user stories by slicing them vertically and they comply to INVEST criteria. Each Epic and user story contains acceptance criteria with necessary business rules and mockups attached to them.

Typical developer work products on Agile projects include code. Agile developers also often create automated unit tests. These tests might be created after the development of code. In some cases, though, developers create tests incrementally, before each portion of the code is written, in order to provide a way of verifying, once that portion of code is written, whether it works as expected. While this approach is referred to as test first or test-driven development, in reality the tests are more a form of executable low-level design specifications rather than tests.

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

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

In some regulated agile projects, more inspection of the documentation that is being produced during the project execution may be required.

For example: Federal drug administration (FDA) projects require lot of audit reports, strategy reports and compliance reports that need to be produced during the project execution.

3.1.3 Test Levels

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

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

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

- Unit testing, typically done by the developer

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

Example of Feature acceptance testing:

As a new depositor, in order to protect my money, I want to save my money a bank account. Since, I am a new depositor; my default opening balance in a new bank account is INR 0.00

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

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

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

3.1.4 Testing and Configuration Management

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

The tools usage in Agile projects must supplement people interaction but should not replace it. In Agile software development projects, there are many tools that are being used in various situations. Few examples of tools are: Developers use tools like SONAR for static analysis and code coverage, JUnit for code coverage. Developers also use configuration management tools like SVN, where they can check in source code, unit tests. The configuration management tool will compile and build the code with test frameworks.

Examples of few tools that are commonly used with build automation are:

- Ant
- Jenkins
- PMD
- FindBug
- Checkstyle
- JSHint
- JUnit
- Jacoco
- Subversion

These common toolset and build scripts allows for customization when needed, giving their customers flexibility to configure the tools to their individual needs. They support centralized controls and reporting, decentralized execution (Identity-based security). They can be easily deployed on the existing infrastructure where possible.

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

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

The goal of automated tests may also be to check the build function and its stabilization. Any build failure immediately warrants an action from the developers and testers. This can quickly help other developers not to get delayed due to build failure.

- There are 3 kinds of builds, namely Fast builds, Full build, and Push-To-QA build.
- Fast build is triggered when it detects the source code changes, and it also detects complication, unit testing or packaging errors. It also performs the static code analysis and provides a web based build report on the code coverage. This build can be customized as per the team
- Full Build is scheduled usually twice a day, it is same as fast build, and it deploys into DEV server
- Push-To-QA build is scheduled On-demand and it deploy the latest DEV build to QA

Automated testing and build tools help to manage the regression risk associated with the frequent change that often occurs in Agile projects. However, over-reliance on automated unit testing alone to manage these risks can be a problem, as unit testing often has limited defect detection effectiveness. Automated tests at the integration and system levels are also required. There are frequent changes to software/application in Agile development, effective use of build tools and automated testing helps in managing and minimizing risks related to regression issues.

3.1.5 Organizational Options for Independent Testing

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

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

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

3.2 Status of Testing in Agile Projects

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

3.2.1 Communicating Test Status, Progress, and Product Quality

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

There are lot of simple tools and techniques in place by which agile teams can monitor the status of iteration. Different tools and techniques available for Agile teams to record their status are:

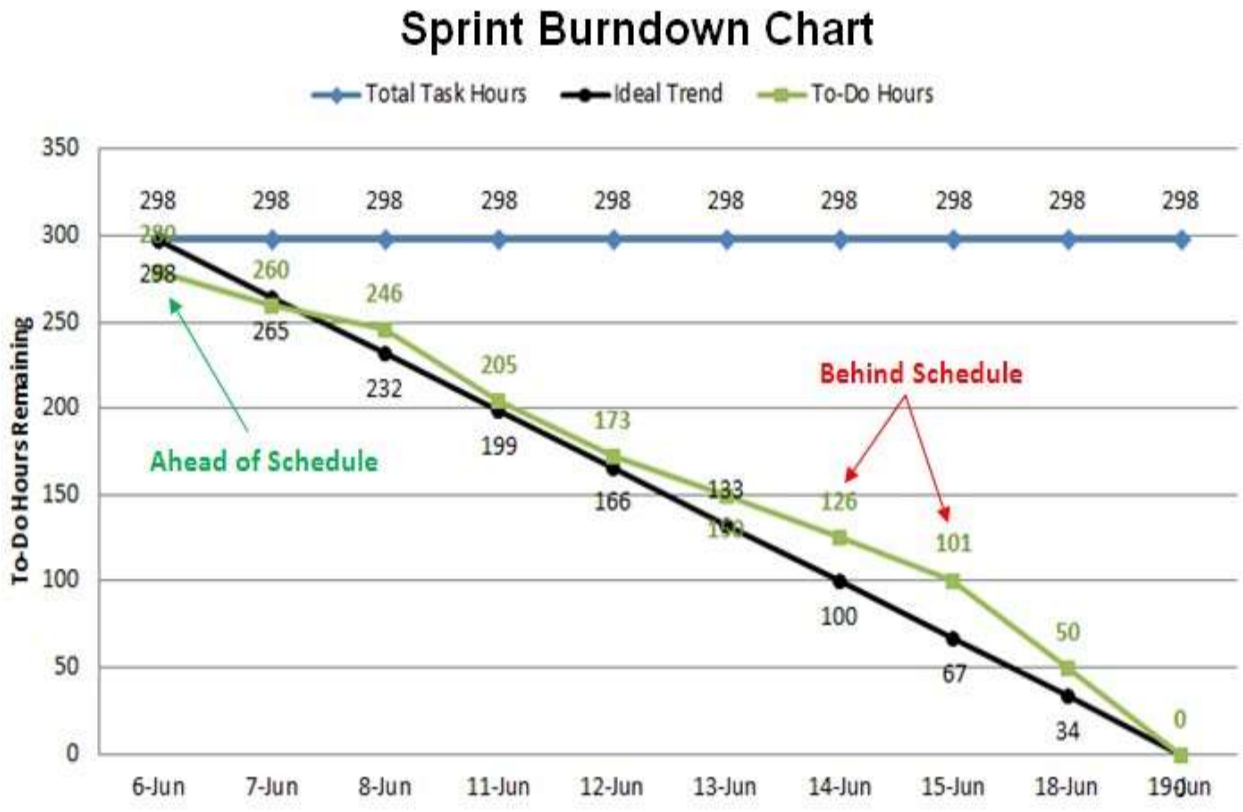
- Daily standup, it is a daily planning meeting; this meeting is not be exactly meant to update daily status
- Various soft dashboards.
- Sprint Burndown charts.
- Big visible task boards at the team location.
- Wiki portal where everyone can update the daily progress.
- Lastly, emails in simple language that everyone can understand.

Agile teams may also use tools like Rally, Version One, Jira etc to know the status of the sprint or release. These tools generate great metrics where several inferences can be drawn. The testers can also generate update the tools on automation status and testing progress for the current sprint.

Teams may use burndown charts 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.

Agile team use **burn down charts** to track progress in every iteration. The burn down always gives the "To Do" work in the current iteration. It may not focus on what was accomplished. A burn down chart can also be measured for sprints to track scope versus story points. The image of daily burn down is shown in Figure below.

The task board serves as a visual representation of the agile team, including development and testing. There are story cards on the left to do and development and testing tasks shown in progress. There may be columns of "To do", "In Progress" and "Done" states in the task board. We can use multiple color Post-it's to make the task board attractive.



It is recommended that the whole team do their daily standup before the task board and give their standup update. The team can move the tasks back and forth, if the team feels that they are in the wrong place. Any tasks including testing tasks are moving as per the expectations. The team can take a collective decision to address them by pairing up. There can be various testing tasks related to automation, test plans, test strategy, exploratory and manual tests in that task board.

The daily standup includes all agile team members, where each team member is supposed to give update on the three questions.

- What did I do yesterday that helped the Development Team meet the Sprint Goal?
- What will I do today to help the Development Team meet the Sprint Goal?
- Do I see any impediment that prevents me or the Development Team from meeting the Sprint Goal?

The Scrum Master will make a note of any impediments and that come in the way and tries to address them offline. Few examples of impediments that may come up during standup are

- Server outage due to which cannot move forward in the development.
- A tester local database instance got corrupted due to which the data base is not working
- A problem that is related to accessing the third party database.

In order to improve the overall quality of the product that is being developed, agile teams take lot of informal feedback from the Product Owner. The teams may also initiate a customer feedback survey to capture the voice of the customer. They can also use several other metrics, the defect density, the test coverage percentage, the number of team members per story, the regression test results and code churn. Metrics must be carefully measured, since the management get what they measure

For example:

- If the management sets velocity targets to the team, the team may inflate story points of all stories

- If the management sets 100% utilization, the team may mark up the hours and show more than 100% utilization, but no value delivered to the customer.

To provide an instant, detailed visual representation of the whole team's current status, including the status of testing, teams may use **Agile task boards**. The story cards, development tasks, test tasks, and other tasks created during iteration planning 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.

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

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

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

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

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

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

3.2.2 Managing Regression Risk with Evolving Manual and Automated Test Cases

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

Testers often do routine job every iteration; however, there might be excessive work load during some iteration it may be difficult to juggle around several issues. The testers at the beginning of the iteration will have to get together to prepare a test strategy to deal with the unknowns and known and proceed accordingly. They testers team should also review the

tests written earlier and discuss the probably touch points on where they need to add or delete the scripts.

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

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

It is critical that testers have the ability to quickly identify and update test cases from previous iterations and/or releases that are affected by the changes made in the current iteration. Defining how the team designs, writes, and stores test cases should occur during release planning. Good practices for test design and implementation need to be adopted early and applied consistently. The shorter timeframes for testing and the constant change in each iteration will increase the impact of poor test design and implementation practices.
Use of test automation, at all test levels, allows Agile teams to provide rapid feedback on product quality. Well-written automated tests provide a living document of system functionality. By checking the automated tests and their corresponding test results into the configuration management system, aligned with the versioning of the product builds, Agile teams can review the functionality tested and the test results for any given build at any given point in time.

Automated unit tests are run before source code is checked into the mainline of the configuration management system to ensure the code changes do not break the software build. To reduce build breaks, which can slow down the progress of the whole team, code should not be checked in unless all automated unit tests pass. Automated unit test results provide immediate feedback on code and build quality, but not on product quality.
Automated acceptance tests are run regularly as part of the continuous integration full system build. These tests are run against a complete system build at least daily, but are generally not run with each code check-in as they take longer to run than automated unit tests and could slow down code check-ins. The test results from automated acceptance tests provide feedback on product quality with respect to regression since the last build, but they do not provide status of overall product quality.

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

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

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

- Test data generation

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

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

3.3 Role and Skills of a Tester in an Agile Team

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

3.3.1 Agile Tester Skills

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

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

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

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

3.3.2 The Role of a Tester in an Agile Team

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

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

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

Agile organizations may encounter some test-related organizational risks:

- Testers work so closely to developers that they lose the appropriate tester mindset
- Testers become tolerant of or silent about inefficient, ineffective, or low-quality practices within the team
- Testers cannot keep pace with the incoming changes in time-constrained iterations
- To mitigate these risks, organizations may consider variations for preserving independence.

4 Agile Testing Methods, Techniques, and Tools

4.1 Agile Testing Methods

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

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

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

Test-Driven Development

Test-driven development (TDD) is an advanced technique of using automated unit tests to drive the design of software and force decoupling of dependencies. In other words, it is used to develop code guided by automated test cases. The process for test-driven development is:

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

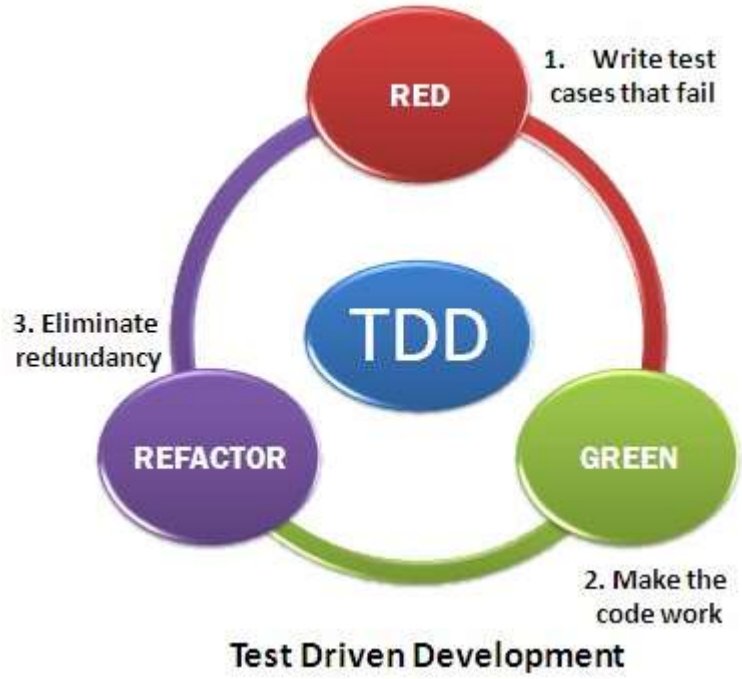
The tests written are primarily unit level and are code-focused, though tests may also be written at the integration or system levels. Test-driven development gained its popularity through Extreme Programming, but is also used in other Agile methodologies and sometimes in sequential lifecycles. It helps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

The result of using this practice is a comprehensive suite of unit tests that can be run at any time to provide feedback that the software is still working. This technique is heavily emphasized by those using Agile development methodologies. Creating and running automated tests inside.

- Abstracting dependencies in an object-oriented world
- Refactoring new and old features to remove duplication in code
- How to Author a Unit Test
- How to Organize Tests into Test Lists
- How to Run Selected Tests

The motto of test-driven development is "Red, Green and Refactor."

- Red: Create a test and make it fail.
- Green: Make the test pass by any means necessary.
- Refactor: Change the code to remove duplication in your project and to improve the design while ensuring that all tests still pass.



The Red/Green/Refactor cycle is repeated very quickly for each new unit of code

Benefits of Test Driven Development

- ✓ The suite of unit tests provides constant feedback that each component is still working.
- ✓ The unit tests act as documentation that cannot go out-of-date, unlike separate documentation, which can and frequently does.
- ✓ When the test passes and the production code is refactored to remove duplication, it is clear that the code is finished, and the developer can move on to a new test.
- ✓ Test-driven development forces critical analysis and design because the developer cannot create the production code without truly understanding what the desired result should be and how to test it.
- ✓ The software tends to be better designed, that is, loosely coupled and easily maintainable, because the developer is free to make design decisions and refactor at any time with confidence that the software is still working. This confidence is gained by running the tests. The need for a design pattern may emerge, and the code can be changed at that time.
- ✓ The test suite acts as a regression safety net on bugs: If a bug is found, the developer should create a test to reveal the bug and then modify the production code so that the bug goes away and all other tests still pass. On each successive test run, all previous bug fixes are verified.

An example of Test Driven Development process is explained below using the Visual Studio.

When Visual Studio Team System is used, the following steps can be performed while processing a work item that is already assigned. Make sure that a Test Project in the solution available for creating new tests. This project should reference the class library in which you intend to add new functionality.

Follow these below steps

1. Understand the requirements of the story, work item, or feature that is being worked on.

2. **Red**: Create a test and make it fail.

i. Imagine how the new code should be called and write the test as if the code already existed. We may not get IntelliSense because the new method does not yet exist.
ii. Create the new production code stub. Write just enough code so that it compiles.
iii. Run the test. It should fail. This is a calibration measure to ensure that your test is calling the correct code and that the code is not working by accident. This is a meaningful failure, and you expect it to fail.

3. **Green**: Make the test pass by any means necessary.
   i. Write the production code to make the test pass. Keep it simple.
   ii. Some advocate the hard-coding of the expected return value first to verify that the test correctly detects success. This varies from practitioner to practitioner.
   iii. If the code is written so that the test passes as intended, you are finished. Code need not be written more speculatively. The test is the objective definition of "done." If new functionality is still needed, then another test is needed. Make this one test pass and continue.
   iv. When the test passes, it might want to run all tests up to this point to build confidence that everything else is still working.

4. **Refactor**: Change the code to remove duplication in your project and to improve the design while ensuring that all tests still pass.
   i. Remove duplication caused by the addition of the new functionality.
   ii. Make design changes to improve the overall solution.
   iii. After each refactoring, rerun all the tests to ensure that they all still pass.

5. Repeat the cycle. Each cycle should be very short, and a typical hour should contain many Red/Green/Refactor cycles.

Acceptance Test-Driven Development

Acceptance test-driven development defines acceptance criteria and tests during the creation of user stories. Acceptance test-driven development is a collaborative approach that allows every stakeholder to understand how the software component has to behave and what the developers, testers, and business representatives need to ensure this behavior.

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

It is a test-first approach in which acceptance criteria is well understood by the development team and test cases are created based on the bullet points stated in the acceptance criteria; any modification to the acceptance criteria may done in prior planning meeting in a negotiation with the Product Owner.

The idea of Acceptance test driven development is a set of tests that must pass before an application can be considered finished. The value of testing an application before delivering it is relatively well established.

The team collaborate together to create test cases, with a business representative validating the test cases. The test cases are essentially the characteristics of a user story. The test cases are also called as examples.

The examples include positive tests and scenarios that affirm the right behavior of the story and exception handlers based on sequence of activities. Subsequently, negative tests that cover negative validation flows and non-functional requirements like usability and performance may also be covered. Tests are written in simple language providing necessary inputs, throughputs and the expected output. The tests may also cover certain boundary

contains that may or may not be part of the story to establish the story dependency to the other stories. In addition, it's not recommended to duplicate the test cases with similar characteristics.

Example acceptance test case is given below

scenario "System approves an card swipe for an amount less than the preset maximum limit"

{

given "the System has selected an open card swipe",

and "the System has chosen to approve the swipe",

and "the card swipe amount is less than the agreed maximum limit",

when "the System completes the action",

then "the card swipe should be successfully approved",

}

Behavior-Driven Development

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

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

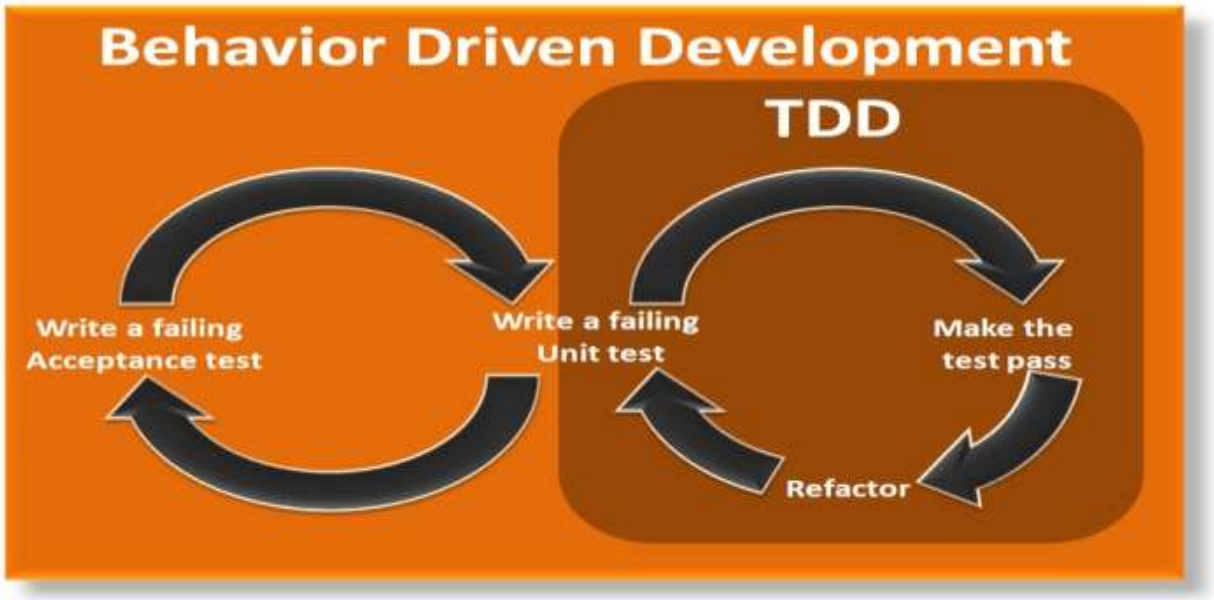
Given some initial context,
When an event occurs,
Then ensure some outcomes.

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

Benefits of BDD:

- ✓ The suite of unit tests provides constant feedback that each component is still working.
- ✓ The unit tests act as documentation that cannot go out-of-date, unlike separate documentation, which can and frequently does.
- ✓ When the test passes and the production code is refactored to remove duplication, it is clear that the code is finished, and the developer can move on to a new test.
- ✓ Test-driven development forces critical analysis and design because the developer cannot create the production code without truly understanding what the desired result should be and how to test it.
- ✓ The software tends to be better designed, that is, loosely coupled and easily maintainable, because the developer is free to make design decisions and refactor at any time with confidence that the software is still working. This confidence is gained by running the tests. The need for a design pattern may emerge, and the code can be changed at that time.
- ✓ The test suite acts as a regression safety net on bugs: If a bug is found, the developer should create a test to reveal the bug and then modify the production code so that the bug goes away and all other tests still pass. On each successive test run, all previous bug fixes are verified.

- ✓ It also reduces debugging time.



4.1.2 The Test Pyramid

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

4.1.3 Testing Quadrants, Test Levels, and Testing Types

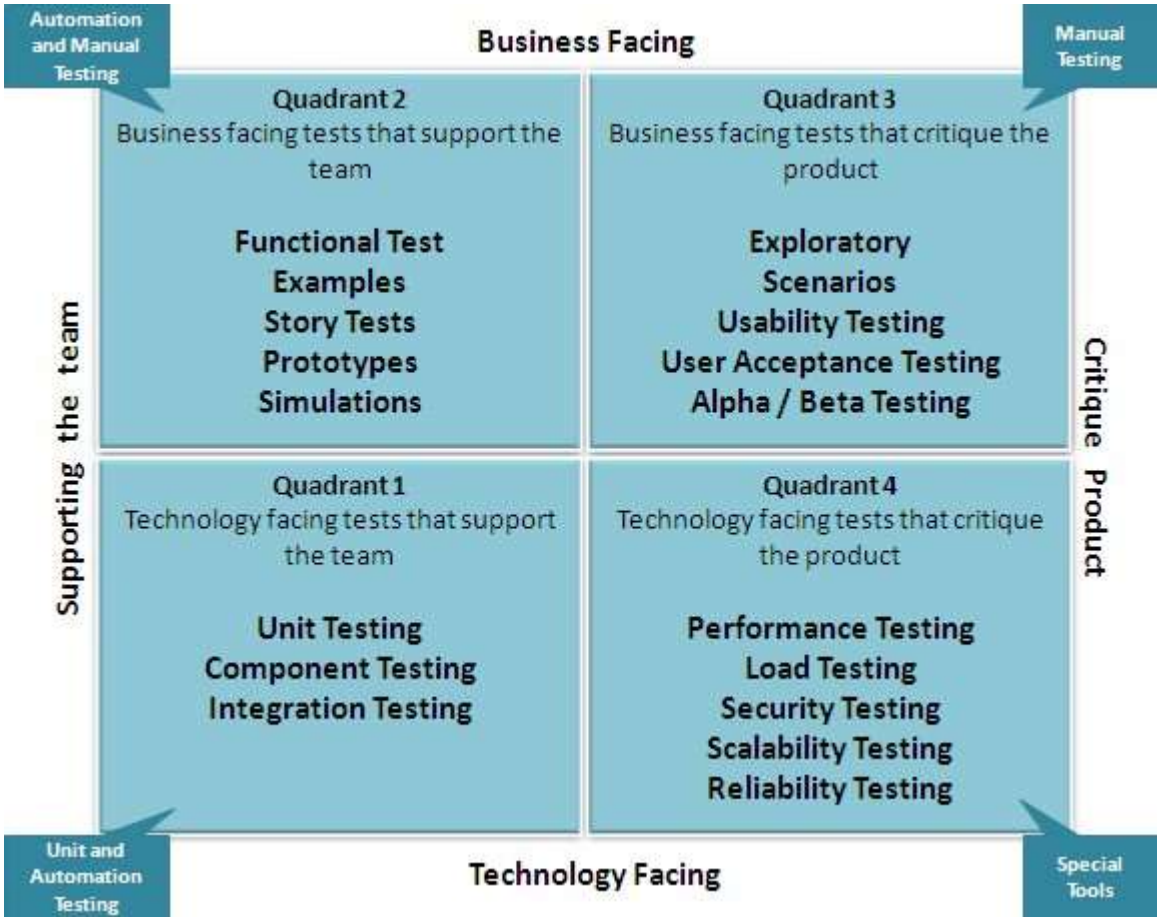
Testing quadrants, defined by Brian Marick, align the test levels with the appropriate test types in the Agile methodology. The testing quadrants model, and its variants, helps to ensure that all important test types and test levels are included in the development lifecycle. This model also provides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

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

- **Quadrant Q1** is unit level, technology facing, and supports the developers. This quadrant contains unit tests. These tests should be automated and included in the continuous integration process.
- **Quadrant Q2** is system level, business facing, and confirms product behavior. This quadrant contains functional tests, examples, story tests, user experience prototypes, and simulations. These tests check the acceptance criteria and can be manual or automated. They are often created during the user story development and thus improve the quality of the stories. They are useful when creating automated regression test suites.
- **Quadrant Q3** is system or user acceptance level, business facing, and contains tests that critique the product, using realistic scenarios and data. This quadrant contains exploratory testing, scenarios, process flows, usability testing, user acceptance

testing, alpha testing, and beta testing. These tests are often manual and are user-oriented.

- **Quadrant Q4** is system or operational acceptance level, technology facing, and contains tests that critique the product. This quadrant contains performance, load, stress, and scalability tests, security tests, maintainability, memory management, compatibility and interoperability, data migration, infrastructure, and recovery testing. These tests are often automated.



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

4.1.4 The Role of a Tester

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

Teamwork

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

- **Cross-functional**: Each team member brings a different set of skills to the team. The team works together on test strategy, test planning, test specification, test execution, test evaluation, and test results reporting.
- **Self-organizing**: The team may consist only of developers, but ideally there would be one or more testers.
- **Co-located**: Testers sit together with the developers and the product owner.

- **Collaborative**: Testers collaborate with their team members, other teams, the stakeholders, the product owner, and the Scrum Master.
- **Empowered**: Technical decisions regarding design and testing are made by the team as a whole (developers, testers, and Scrum Master), in collaboration with the product owner and other teams if needed.
- **Committed**: The tester is committed to question and evaluate the product's behavior and characteristics with respect to the expectations and needs of the customers and users.
- **Transparent**: Development and testing progress is visible on the Agile task board.
- **Credible**: The tester must ensure the credibility of the strategy for testing, its implementation, and execution; otherwise the stakeholders will not trust the test results. This is often done by providing information to the stakeholders about the testing process.
- **Open to feedback**: Feedback is an important aspect of being successful in any project, especially in Agile projects. Retrospectives allow teams to learn from successes and from failures.
- **Resilient**: Testing must be able to respond to change, like all other activities in Agile projects.

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

Sprint Zero

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

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

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

Integration

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

Test Planning

Since testing is fully integrated into the Agile team, test planning should start during the release planning session and be updated during each sprint. Test planning for the release and each sprint should address the issues.
Sprint planning results in a set of tasks to put on the task board, where each task should have a length of one or two days of work. In addition, any testing issues should be tracked to keep a steady flow of testing.

Agile Testing Practices

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

- **Pairing**: Two team members (e.g., a tester and a developer, two testers, or a tester and a product owner) sit together at one workstation to perform a testing or other sprint task.
- **Incremental test design**: Test cases and charters are gradually built from user stories and other test bases, starting with simple tests and moving toward more complex ones.
- **Mind mapping**: 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.

In Agile team, all developers and testers closely collaborate with each other to deliver business value. Every tester has to bring versatile skills on board to become a cross function team member.

Agile testers must have primary skills in automating test cases, test-driven development, and acceptance test-driven development, manual testing both white box and black-box testing.

Agile methods ask for greater collaboration, effective communication and interaction between team members and teams outside as well as stake holders who can directly or indirectly influence the product development.

Following are the few attributes expected of Agile testers:

- ✓ Great interpersonal skills with clarity in expression
- ✓ Result oriented and positive attitude
- ✓ Go getter and get it done attitude
- ✓ Make test results visible to make a collective decision
- ✓ Quality first mindset, intolerance to any imperfections
- ✓ Someone who is good at organizing their task boards
- ✓ Good analytical capabilities to evaluate test results
- ✓ Understand the user stories and their acceptance criteria
- ✓ Respond to change without impacting the business
- ✓ Those who are die-hard learners on everything related to testing
- ✓ Those who believe in team wins than individual wins
- ✓ Constantly upgrade their skills using new tools and techniques

4.2 Assessing Quality Risks and Estimating Test Effort

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

4.2.1 Assessing Quality Risks in Agile Projects

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

Risk is the possibility of a negative or undesirable outcome or event. The level of risk is found by assessing the likelihood of occurrence of the risk and the impact of the risk. When the primary effect of the potential problem is on product quality, potential problems are referred to

as quality risks or product risks. When the primary effect of the potential problem is on project success, potential problems are referred to as project risks or planning risks.

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

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

Examples of quality risks for a system include:

- Incorrect calculations in reports (a functional risk related to accuracy)
- Slow response to user input (a non-functional risk related to efficiency and response time)
- Difficulty in understanding screens and fields (a non-functional risk related to usability and understandability)

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

An example of how the quality risk analysis process in an Agile project may be carried out during iteration planning is outlined in the following steps:

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

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

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

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

4.2.2 Estimating Testing Effort Based on Content and Risk

During release planning, the Agile team estimates the effort required to complete the release. The estimate addresses the testing effort as well. A common estimation technique used in Agile projects is planning poker, a consensus-based technique. The product owner or customer reads a user story to the estimators. Each estimator has a deck of cards with values similar to the Fibonacci sequence (i.e., 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …) or any other progression of choice (e.g., shirt sizes ranging from extra-small to extra-extra-large). The values represent the number of story points, effort days, or other units in which the team estimates. The Fibonacci sequence is recommended because the numbers in the sequence reflect that uncertainty grows proportionally with the size of the story. A high estimate usually

means that the story is not well understood or should be broken down into multiple smaller stories.

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

4.3 Techniques in Agile Projects

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

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

Agile projects outline initial requirements as user stories in a prioritized backlog at the start of the project. Initial requirements are short and usually follow a predefined format. Non-functional requirements, such as usability and performance, are also important and can be specified as unique user stories or connected to other functional user stories.

Non-functional requirements may follow a predefined format or standard, such as, or an industry specific standard.
The user stories serve as an important test basis. Other possible test bases include:

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

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

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

- External interfaces: Descriptions of the connections between the system to be developed and the outside world. External interfaces can be divided into different types (user interface, interface to other systems, etc.).
- Constraints: Any design and implementation constraint that will restrict the options for the developer. Devices with embedded software must often respect physical constraints such as size, weight, and interface connections.
- Data definitions: The customer may describe the format, data type, allowed values, and default values for a data item in the composition of a complex business data structure (e.g., the ZIP code in a U.S. mail address).
- In addition to the user stories and their associated acceptance criteria, other information is relevant for the tester, including:
- How the system is supposed to work and be used
- The system interfaces that can be used/accessed to test the system
- Whether current tool support is sufficient
- Whether the tester has enough knowledge and skill to perform the necessary tests

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

Test Levels

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

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

User Story

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

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

Feature

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

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

Iteration

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

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

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

Release

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

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

4.3.2 Applying Acceptance Test-Driven Development

Acceptance test-driven development is a test-first approach. Test cases are created prior to implementing the user story. The test cases are created by the Agile team, including the

developer, the tester, and the business representatives and may be manual or automated. The first step is a specification workshop where the user story is analyzed, discussed, and written by developers, testers, and business representatives. Any incompleteness, ambiguities, or errors in the user story are fixed during this process.

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

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

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

4.3.3 Functional and Non-Functional Black Box Test Design

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

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

4.3.4 Exploratory Testing and Agile Testing

Exploratory testing is important in Agile projects due to the limited time available for test analysis and the limited details of the user stories. In order to achieve the best results, exploratory testing should be combined with other experience-based techniques as part of a reactive testing strategy, blended with other testing strategies such as analytical risk-based testing, analytical requirements-based testing, model-based testing, and regression-averse testing. Test strategies and test strategy blending is discussed in the Foundation Level syllabus.

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

A test charter may include the following information:

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

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

- Survey session (to learn how it works)
- Analysis session (evaluation of the functionality or characteristics)
- Deep coverage (corner cases, scenarios, interactions)

The quality of the tests depends on the testers' ability to ask relevant questions about what to test. Examples include the following:

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

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

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

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

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

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

- Actual behavior: recording of actual behavior of the system that needs to be saved (e.g., video, screen captures, output data files)

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

4.4 Tools in Agile Projects

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

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

4.4.1 Task Management and Tracking Tools

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

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

4.4.2 Communication and Information Sharing Tools

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

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

- Product feature diagrams, feature discussions, prototype diagrams, photos of whiteboard discussions, and other information
- Tools and/or techniques for developing and testing found to be useful by other members of the team

- Metrics, charts, and dashboards on product status, which is especially useful when the wiki is integrated with other tools such as the build server and task management system, since the tool can update product status automatically
- Conversations between team members, similar to instant messaging and email, but in a way that is shared with everyone else on the team

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

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

Desktop sharing and capturing tools provide the following benefits:

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

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

4.4.3 Software Build and Distribution Tools

As discussed earlier in this syllabus, daily build and deployment of software is a key practice in Agile teams. This requires the use of continuous integration tools and builds distribution tools.

4.4.4 Configuration Management Tools

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

4.4.5 Test Design, Implementation, and Execution Tools

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

- **Test design tools**: Use of tools such as mind maps has become more popular to quickly design and define tests for a new feature.
- **Test case management tools**: The type of test case management tools used in Agile may be part of the whole team's application lifecycle management or task management tool.
- **Test data preparation and generation tools**: Tools that generate data to populate an application's database are very beneficial when a lot of data and combinations of data are necessary to test the application. These tools can also help re-define the database structure as the product undergoes changes during an Agile project and refactor the scripts to generate the data. This allows quick updating of test data as changes occur. Some test data preparation tools use production data sources as a raw material and use scripts to remove or anonymize sensitive data. Other test data preparation tools can help with validating large data inputs or outputs.
- **Test data load tools**: After data has been generated for testing, it needs to be loaded into the application. Manual data entry is often time consuming and error

prone, but data load tools are available to make the process reliable and efficient. In fact, many of the data generator tools include an integrated data load component. In other cases, bulk-loading using the database management systems is also possible.

- **Automated test execution tools**: There are test execution tools which are more aligned to Agile testing. Specific tools are available via both commercial and open source avenues to support test first approaches, such as behavior-driven development, test-driven development, and acceptance test-driven development. These tools allow testers and business staff to express the expected system behavior in tables or natural language using keywords.
- **Exploratory test tools**: Tools that capture and log activities performed on an application during an exploratory test session are beneficial to the tester and developer, as they record the actions taken. This is useful when a defect is found, as the actions taken before the failure occurred have been captured and can be used to report the defect to the developers. Logging steps performed in an exploratory test session may prove to be beneficial if the test is ultimately included in the automated regression test suite.

4.4.6 Cloud Computing and Virtualization Tools

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

5 Sample Exams

5.1 Questions – SET 1

Question 1

The Agile Manifesto has 4 statements of values. Match the agile value on the left (1-4) with its traditional counterpart on the right (i-iv).

1) Customer collaboration over	i) Processes and tools
2) Responding to change over	ii) Following a plan
3) Individuals and interactions over	iii) Contract negotiation
4) Working software over	iv) Comprehensive documentation

Answer Set:
A. 1 – iii, 2 – iv, 3 – ii, 4 – i
B. 1 – iii, 2 – ii, 3 – i, 4 – iv
C. 1 – iv, 2 – ii, 3 – i, 4 – iii
D. 1 – ii, 2 – iii, 3 – iv, 4 – i

Question 2

Which of the following statements best reflects one of the values of the Agile Manifesto?

Answer Set:
A. Working software allows the customer to provide rapid feedback to the developer.
B. Developers should use unit testing tools to support the testing process.
C. Business representatives should provide a backlog of user stories and their estimates to the team.
D. Adopting plans to change adds no real value to an agile project.

Question 3

Which **TWO** activities below best represent responsibilities that are consistent with agile development's Whole Team approach?

Select TWO options.
Answer Set:
A. Testers are responsible for developing unit tests which they pass on to the developers for testing
B. Business representatives are expected to select the tools the team will use during the project
C. Testers are expected to work with customer representatives to create acceptance tests
D. The whole team, not just testers, has responsibility for the quality of the product
E. Developers are expected to test non-functional requirements (performance, usability, security, etc.)

Question 4

Which of the following is an advantage of having the whole team responsible for quality?

Answer Set:
A. Companies no longer need to recruit and train software testing specialists.
B. Test automation tasks are now the responsibility of the development team instead of the test team.

C. Role barriers are eliminated and team members contribute to project success based on their unique skills and perspectives.
D. Project costs are lower because the need for a specialized test team is eliminated.

Question 5

Which TWO of the following statements are true?
1) Early feedback gives the developers more time to develop new system features because they spend less time reworking features expected in a given iteration.
2) Early feedback enables agile teams to deliver features with the highest business value first, because the customer maintains focus on features with the highest system value.
3) Early feedback reduces costs because it decreases the amount of time needed for system testing.
4) Early feedback makes it more likely that the system built is what the customer wanted because they are given the opportunity to make changes throughout the iteration.

Answer Set:
A. 1 and 4
B. 2 and 3
C. 2 and 4
D. 1 and 3

Question 6

Which of the following is a benefit of the agile process promoting early and frequent feedback?

Answer Set:
A. The total number of defects found during the project is much higher than on traditional software development projects such as waterfall.
B. There is less rework because customers see the product regularly.
C. It is easy to determine the developer who introduces the most defects when integrating code.
D. There is enough time to complete all features scheduled for the given iteration.

Question 7

Match the following agile software development approaches on the top with their corresponding descriptions on the bottom.

1) Extreme Programming
2) Scrum
3) Kanban

I. Embraces 5 values to guide development: Communication, Simplicity, Feedback, Courage and Respect
II. Divides the project into short iterations called sprints.
III. Optimizes the ‘flow’ of work in a value-added chain.

Answer Set:
A. 1-i, 2-iii, 3-ii
B. 1-i, 2-ii, 3-iii
C. 1-i, 2-ii, 3-iii
D. 1-iii, 2-ii, 3-i

Question 8

During an iteration planning meeting, the team is sharing their thoughts about a user story. The product owner advises that the customer should have one screen to enter information. The developer explains that there are technical limitations for the feature, due to the amount of information needed to be captured on the screen. Another developer says that there are risks about performance as the information will be stored in an external offsite database. Which of the following would best represent a tester's contribution to this discussion?

Answer Set:
A. The tester advises that the screen for the user story needs to be a single page to reduce test automation effort.
B. The tester advises that usability is more important than performance.
C. The tester advises that performance acceptance criteria should standard maximum of 1 second for data storage.
D. The tester advises that the user story needs acceptance criteria to be testable.

Question 9

Which of the following BEST describes a tester participating in a retrospective meeting?

Answer Set:
A. As a tester participating in a retrospective meeting, I should bring in topics that are related to testing only. All other topics will be covered by different participants.
B. As a tester, I participate in a retrospective meeting as an observer, ensuring that the meeting follows the retrospective rules and agile values.
C. As a tester participating in a retrospective meeting, I should provide feedback and input on all activities conducted by the team during the sprint.
D. As a tester, I should only attend and participate in a retrospective meeting if I have any feedback and input related to activities conducted by the team during the sprint.

Question 10

Which of the following items should NOT be raised during a retrospective meeting?

Answer Set:
A. There should be more emphasis on unit testing in the future, to improve overall quality.
B. The build process is manual and takes too long. Research and implementation of an automated build framework should be done.
C. Tester XYZ is struggling to find defects. Test design training is required for this resource.
D. Automated regression test suites are taking too long to run. A review of the tests, to eliminate redundant or unnecessary tests, is required.

Question 11

Which of the following is NOT a principle of continuous integration?

Answer Set:
A. Continuous integration helps to build changed software regularly, including testing and deploying, in an automated way.
B. Continuous integration allows new builds to be available frequently to testers and stakeholders.
C. Continuous integration helps to identify new integration defects early and makes the analysis of these defects easier.
D. Continuous integration ensures that testing of builds is done manually, as this generates more reliable results than automated scripts.

Question 12

Which of the following activities would a tester do during release planning?

Answer Set:
A. Produce a list of acceptance tests for user stories
B. Help break down user stories into smaller and more detailed tasks.
C. Estimate testing tasks generated by new features planned for this iteration.
D. Support the clarification of the user stories and ensure that they are testable

Question 13

What is the most appropriate explanation of a ‘user story’?

Answer Set:
A. An artifact that the tester must review and sign off before testing can begin.
B. An artifact used to detail only the functional requirements of the system.
C. An artifact documented by business representatives to help developers and testers understand the system requirements.
D. An artifact written collaboratively by developers, testers, and business representatives to capture requirements.

Question 14

Which of the following test activities is typically done during agile projects, but is not as common on traditional projects?

Answer Set:
A. Testers write detailed test plans so all team members can understand what will be tested during each iteration.
B. Testers are heavily involved in the creation of automated test cases which are then used to verify the implementation of the requirements.
C. Testers perform exploratory testing in order to find important defects quickly.
D. Testers collaborate with developers to better understand what needs to be tested.

Question 15

Consider the following activities:

i. Strict enforcement of system test level entry and exit criteria.
ii. Collaboration between tester, developer, and business stakeholders to define acceptance criteria.
iii. Functional verification testing of user stories developed in the previous iteration.

Which of the following combination of these activities should occur in an agile project?

Answer Set:
A. ii only
B. i and ii
C. ii and iii
D. iii only

Question 16

Which TWO of the following statements are true on agile projects?

Select TWO options.

Answer Set:

A. Testers should work closely with developers while retaining an objective outlook.
B. Test managers do not exist in organizations doing agile development.
C. There is no difference between what testers and developers do on agile projects.
D. Developers should rely on testers to create the automated regression tests.
E. A selection of users may perform beta testing on the product after the completion of a series of iterations.

Question 17

Which of the following statements about independent testing on agile projects is FALSE?

Answer Set:
A. There can be a risk of losing test independence for organizations introducing agile.
B. Independent testers will find more defects than developers regardless of test level.
C. Independent testing can be introduced at the end of a sprint.
D. The independent test team can be part of another team

Question 18

In an agile project, which of the following would best denote product quality at the end of iteration 6 of a new system release consisting of 8 iterations?

Answer Set:
A. No severity 1 or 2 defects were detected during system testing of iteration 6, which allowed the teams to move into iteration 7.
B. The results of a customer beta test on the iteration 6 software release indicate that the system works correctly and that it has improved productivity.
C. The agile team has been successfully tracking to estimates, with limited variance showing on the burndown charts for all iterations to date.
D. All story cards in scope for each iteration, up to the current iteration, have been marked as “Done”, but with some technical debt being incurred.

Question 19

Which of the following is best at showing the team’s progress against estimates?

Answer Set:
A. Burndown charts
B. Automation logs
C. The agile task board showing user story and task progress
D. Defect tracking tools

Question 20

The business advises during iteration 5 planning that they require changes to the system delivered in iteration 3. Of the following activities, which would need to be done first to minimize the introduction of regression risk when this feature is changed?

Answer Set:
A. Review and update all manual and automated tests impacted by this change to meet the new acceptance criteria.
B. Write new manual and automated tests for the feature and add them to the regression test suite.
C. Automate all test cases from the previous iteration and add them to the automated regression test suite.
D. Increase the amount of test automation around the system to include more detailed test conditions.

Question 21

Which TWO of the following are reasons why automation is essential within agile projects?
i. So that teams maintain or increase their velocity
ii. To prevent the test team from becoming bored with manual, repetitive tasks
iii. To retest all test cases from previous iterations
iv. To eliminate regression in the product due to high code churn
v. To ensure that code changes do not break the software build

Answer Set:
A. i and iv
B. i and v
C. iii and iv
D. ii and v

Question 22

In agile projects there is more need for testers to understand and develop test automation scripts than in traditional projects. Of the following, which are the TWO reasons why this is a necessary skill on agile projects?

i. Requirements change daily and have to be regression tested. This rapid change requires automated tests because manual testing is too slow.
ii. The tests should generate feedback on product quality as early as possible. So all acceptance tests should be executed in each iteration, ideally as modifications are made. In practice that can only be realized by automated tests.
iii. Test-First and Continuous Integration Practice require that the regression test suite is executed whenever changed code is checked-in. In practice that can only be realized by automated tests.
iv. Iterations or sprints are of fixed length. The team has to guarantee that all tests can be completely executed at the last day of each iteration/sprint. In practice, that can only be realized by automated tests.
v. Agile projects rely on unit testing rather than on systems testing. Since unit tests cannot be executed manually, all tests have to be automated tests.

Answer Set:
A. i & iii
B. ii & v
C. iv & v
D. ii and iii

Question 23

Which tasks are typically expected of a tester on an agile project?

i. decide on user acceptance
ii. design, create and execute appropriate tests
iii. schedule defect reports for analysis
iv. automate and maintain tests
v. improve program logic by pair programming

Answer Set:
A. i & iii
B. ii & iii
C. ii & iv
D. ii & v

Question 24

Which of the following is NOT a typical task performed by the tester within an agile team?

Answer Set:
A. To automate tests and maintain them
B. To mentor and coach other team members
C. To produce and update burndown charts
D. To participate in code analysing activities

Question 25

The term “burndown” refers to which of the following?

Answer Set:
A. A chart showing which team members are working the most, and are likely to be under stress
B. A chart showing the progress of each user story, and when they are likely to be completed
C. A chart showing the amount of work left to be done, versus the time allocated for the iteration
D. A chart showing defects that have been fixed, and when the remaining defects are likely to be fixed

Question 26

Which of the following statements about Test Driven Development (TDD) is FALSE?

Answer Set:
A. TDD is a ”test first” approach to develop reusable automated tests.
B. The TDD cycle is continuously used until the software product is released.
C. TDD helps to document the code for future maintenance efforts.
D. The result of TDD are test classes used by the developer to develop test cases

Question 27

What does the term ‘Test Pyramid’ refers to and illustrate situations for?

Answer Set:
A. The team’s testing workload increases from sprint to sprint
B. The backlog size, and thus the number of tests, decreases
C. The number of automated unit tests is higher than the number of automated tests for higher test levels.
D. The number of automated tests in place increases from sprint to sprint

Question 28

Which of the following demonstrates effective use of the testing quadrants?

Answer Set:
A. When communicating test ideas, the tester can refer to the matching test quadrant, so that the rest of the team will better understand the purpose of the test.
B. The tester can use the types of tests described in the testing quadrants as a coverage metric, the more tests covered from each quadrant, the higher the test coverage.

C. The team should pick a number of tests expected from each quadrant, and the tester should design and execute those tests to ensure all levels and types of tests have been executed.
D. The tester can use the testing quadrants during risk analysis; with the lower level quadrants representing lower risk to customer.

Question 29

Given the following user stories:

"As a bank teller, I can easily navigate through the system menu and links, and find the information I am looking for"
"For all users, the system must display all queries in less than 2 seconds, 90% of the time"

And the associated test cases:

TC1: Login as bank teller. Enter customer ID. Verify that the customer transaction history is easy to find, and that navigating through the menus is intuitive.
TC2: Login as bank teller: Enter customer Name. Verify that the customer accounts are easy to find and that navigating through the menus is intuitive.
TC3: Simulate expected traffic on system and validate the time for customer transaction history to display is less than 2 seconds.

Which TWO test quadrants would the above test cases be part of?

Answer Set:
A. Q1 unit level, technology facing & Q2 system level, business facing
B. Q2 system level, business facing & Q3 system or user acceptance level, business facing
C. Q3 system or user acceptance level, business facing & Q4 system or operation acceptance level, technology facing
D. Q2 system level, business facing & Q4 system or operation acceptance level, technology facing

Question 30

At the beginning of the 5th iteration of a project, a new requirement was introduced to support a new type of browser. The tester realizes that the existing test automation framework and scripts will not support the new type of browser. What is the best course of action for the tester on this team to take?

Answer Set:
A. The tester should notify the team that they are planning on working extra hours throughout the next 2 sprints in order to update the existing test automation framework and scripts to support the new type of browser so as not to disturb the existing sprint plan.
B. The tester will notify the team of the issue. A risk analysis is done, and the team decides that regression testing must be performed on the new type of browser in addition to the other supported browsers. The tester will update the sprint plan by adding tasks to modify the framework and scripts to support the new type of browser.
C. The tester does some research and concludes that the risk is low that any new defects would be introduced in the new type of browser that have not already been found in other supported browsers. The tester continues with the existing sprint plan and makes no changes to test automation framework or scripts.
D. The tester will stop what they are doing, design specific tests for compatibility testing of the new type of browser, and communicate with the team that any other testing work for the sprint will have to be pushed to the next iteration.

Question 31

Given the following results from a product risk analysis that occurred at the beginning of an iteration:

- User story 1(Performance): likelihood: high, impact: high
- User story 2 (Security): likelihood: high, impact: high
- User story 3 (Functional): likelihood: medium, impact: high
- User story 4 (Functional): likelihood: high, impact: medium
- User story 5 (Compatibility): likelihood: low, impact: low
- User story 6 (Recoverability): likelihood: low, impact: low

Which TWO of the following describes best what the team should do with this information?

Select TWO options.

Answer Set:
A. Move onto planning poker session to estimate effort for user stories, and determine what can be done in the current iteration, and what needs to be added to backlog.
B. Remove user stories 5 and 6 from the current iteration and move to a later iteration.
C. Because of the number of high likelihood, high impact risks slotted for this iteration, the team has no choice but to extend the timeframe of the iteration by 2 weeks.
D. The team should collaborate on effective ways to mitigate the high likelihood, high impact risks.
E. The team should plan to complete all items in the current sprint, but save the lower risk items for the end of the sprint, and only test these items if there is time.

Question 32

Given the following user story: “As the president, any data I upload should not be viewable by any other user of the system”

During the first poker planning session, the following story points were given based on risk, effort, complexity, and proper extent of testing:

Customers: 5
Developers: 5
Testers: 20

What is the best outcome following this planning session?

Answer Set:
A. Because the customer’s and developer’s size estimates match, the team can be confident that this estimate is good and should move onto the next user story.
B. The team should hold a conversation to understand why the testers felt this user story was significantly more work. Another round of the planning poker session should occur following that discussion.
C. Because the customer owns the system in the end, the customers’ estimates should be taken as correct when there is a conflict.
D. The poker planning sessions should continue until all estimated story points are an exact match between customers, developers, and testers.

Question 33

An agile team is assigned to a project to update an existing medical device to newer technologies. Since the last release of the existing medical device, a new version of the medical device standard has been released. User access to the device is changing and will be documented in user stories.

Based on this information, and in addition to the user stories, which of the following would best provide relevant information to support your testing activities?

i. Updated version of standards document for medical system.
ii. Existing defects or typical defect areas in existing system.
iii. Obsolete user access test cases and results for existing application.
iv. Performance metrics for existing application.
v. Defects logged during other similar conversion projects for medical devices.

Answer Set:
A. i, ii, iii, iv
B. ii, iv, v
C. i, ii, v
D. All of the above

Question 34

Which alternative is the BEST description of when to stop testing (release criteria) in an agile project?

Answer Set:
A. All test cases have been executed.
B. The probability of remaining faults has been reduced to a level that can be accepted by the customer
C. The achieved test coverage is considered enough. The coverage limit is justified by the complexity of the included functionality, its implementation, and the risks involved.
D. The iteration/sprint is finished

Question 35

Which TWO of the following are examples of testable acceptance criteria for test related activities?

Select TWO options.

Answer Set:
A. Structure based testing: White box testing in addition to black box testing is used.
B. System testing: At least 80% of functional regression tests are automated.
C. Security testing: A threat risk analysis scan is completed with no faults identified.
D. Performance testing: The application is responding in a reasonable amount of time with 5000 users.
E. Compatibility testing: The application is working on all major browsers.

Question 36

Given the following User Story:

“As a bank teller, I would like to be able to view all of my customer’s bank transactions on the screen, so I can answer his/her questions”.

Which of the following can be considered as relevant acceptance test cases?

i. Login as a bank teller, get the customer’s account balance for all open accounts.
ii. Login as a bank teller, enter a customer account ID, get his transactions history on the screen
iii. Login as a bank teller, request customer account ID by using name abbreviations, and get his transaction history on the screen

iv. Login as a bank teller, enter a customer IBAN (international bank account number), get his transaction history on the screen
v. Login as a Bank Teller, enter a customer Account ID, get the Transactions history in less than 3 seconds on screen.

Answer Set:
A. i, ii, iv
B. i, iii, iv
C. ii, iv, v
D. ii, iii, iv

Question 37

Given the following user story: “An online application charges customers to ship purchased items, based on the following criteria**:**

- Standard shipping costs for under 6 items
- Shipping is $5 for 6-10 items.
- Shipping is free for more than 10 items.

Which of the following is the best black box test design technique for the user story?

Answer Set:
A. State Transition testing: Test the following states – browsing, logged in, selecting, purchasing, confirming, and exiting.
B. Decision tables: Test the following conditions – User logged in; At least 1 item in cart; Purchase confirmed; Funding approved; with the resulting action of – Ship Item.
C. Boundary Value Analysis: Test the following inputs – 0,5,6,10,11,max
D. Use Case Testing: Actor=customer; Prerequisites=customer logs in, selects and purchases items; Postconditions= items are shipped.

Question 38

Your manager would like to introduce exploratory testing to your agile team. He has received the following suggestions on how to proceed from previous colleagues:

i. User stories are assigned to testers who are completely new to the user story. There is allotted 120 minutes allocated to complete exploratory testing on the user story. Testers do not need to document tests, or test results, but do need to log defects if any are encountered.

ii. User stories are assigned to testers who have already completed risk based testing on the same areas. There is allotted 120 minutes allocated to complete exploratory testing for this user story. The team expects at the end of the 120 minutes to have a list of test ideas, including data and actors, results and issues encountered, and list of defects to be logged in the defect management tool.

iii. A user story is assigned to business representative. The business representative is told to use the system like the customer would on a day-to-day basis. If issues are encountered, the business representative is told to inform the tester, so that they can prioritize and log the defect.

iv. A user story is assigned to a tester for exploratory testing. Tester is told to learn the functionality of the user story, to make sure the functionality is correct and to include negative testing. There is no set deadline for this exploratory testing to be complete; it depends on what is found by the tester. Documentation is not necessary, but defects need to be logged in defect tracking tool. Your manager presents you with his conclusions about how best to introduce exploratory testing to an agile team.

Which one of your manager's conclusions is correct?

Answer Set:
A. Scenario i IS NOT the best way because: In exploratory testing, test design and test execution
happen at the same time but are guided by a documented test charter that includes actors, test conditions, test data, etc. Test results are also documented and will guide the next test.
B. Scenario ii IS the best way because: In this case, the testers have knowledge of the user story already, which will help them come up with test conditions and ideas. The team is using timeboxed exploratory test sessions. The team is expected to document test conditions, data, and user information, and to log results of the test. Issues are logged in a defect tracking tool just like any other test technique.
C. Scenario iii IS NOT the best way because: This could be describing system acceptance testing, but not exploratory testing.
D. Scenario iv IS NOT the best way because: Documentation is necessary for exploratory testing, and testers must log test ideas and results of testing. The results of testing are used to guide future exploratory testing.

Question 39

Which of the following is one of the purposes of an Application Lifecycle Management (ALM) tool on an agile project?

Answer Set:
A. An ALM tool allows teams to build up a knowledge base on tools and techniques for development and testing activities
B. An ALM tool provides quick response about the build quality and details about code changes
C. An ALM tool provides visibility into the current state of the application, especially with distributed teams
D. An ALM tool generates and loads large volumes and combinations of data to use for testing

Question 40

Which of the following statements is FALSE with respect to exploratory testing?

Answer Set:
A. Exploratory testing encompasses concurrent learning, test design, and execution.
B. Exploratory testing eliminates the need for testers to prepare test ideas prior to test execution.
C. Best results are achieved when exploratory testing is combined with other test strategies.
D. Exploratory testers need to have a solid understanding of the system under test.

5.2 Answers – SET 1

Question	Answer	Comments
1	B	The Manifesto consists of 4 key values: Individuals and Interactions over processes and tools; Working software over comprehensive documentation; Customer collaboration over contract negotiation; Responding to change over following a plan.
2	A	From a customer perspective, working software is much more useful and valuable than overly detailed documentation, and it provides an opportunity to provide the development team rapid feedback.
3	C	Testers support & collaborate with business representatives to help them create suitable acceptance tests.
4	C	Enables a variety of skillsets to be leveraged as needed for the project.
5	C	2) Frequent customer feedback maintains a focus on the features with the highest business value. 4) Customers indicate if requirements are missed or misinterpreted, and modify functionality if they desire.
6	B	Clarifying customer feature requests, early and regularly throughout development, making it more likely that key features will be available for customer use earlier and the product will better reflect what the customer wants.
7	B	Extreme Programming embraces 5 values to guide development: Communication, Simplicity, Feedback, Courage, and Respect. Scrum divides the project into short iterations called sprints. Kanban has no iterations or sprints; it is used to optimize continuous flow of tasks and minimize throughput time of each task.
8	D	The tester contributes by ensuring that the team creates acceptance criteria for the user story.
9	C	All team members, both testers and non-testers, can provide input on both testing and non-testing activities.
10	C	The retrospective meeting is not meant to single out individuals, but to focus on improvements of the process, and the team as a whole.
11	D	Testing should be automated at the unit and integration level to allow for quick feedback on the quality of the build.
12	D	This is expected during release planning.
13	D	In an Agile environment, user stories are written to capture requirements from the perspectives of developers, testers, and business representatives. The collaborative authorship of the user story can use techniques such as brainstorming and mind mapping.
14	B	Test automation at all levels occurs in many agile teams. As the developers focus on automating tests on unit level testers should focus on automating tests on integration, system, and acceptance level. In traditional projects it is not as common to have the same focus on automation. Sometimes automation is done once the system testing is completed in order to work with a stable system or just to automate regression tests for maintenance purposes after the system is deployed to production.
15	A	These three perspectives (tester, developer and business representative) are important to define when a feature is done.
16	A	This is one of the hallmarks of agile projects.
	E	Agile teams can employ various forms of acceptance testing.
17	B	This is a false statement. Independent testers CAN find more defects than developers, but this is dependent on the level of testing being performed and also the expertise of the independent tester.
18	B	Positive customer feedback and working software are key indicators to product quality.
19	A	Burndown charts show the planned progress and release date together with the actual progress of the user stories.
20	A	**A.** Correct – As this feature has previously been delivered, a review of all test assets is required, which should result in the updating of test cases to meet new acceptance criteria, to ensure false negatives (i.e. invalid failing tests) do not occur. This is the initial task to be performed before a decision about any other changes can be made. **B.** Incorrect – This would not be the initial task to perform, as the tester would not

		know what new tests would be required for these changes without reviewing the current tests first. There may not be a need to add new tests – updates to existing tests may suffice. **C.** Incorrect – While this is good practice, it does not address the specific regression risk identified in this scenario. **D.** Incorrect – Same as with choice B. Without reviewing the current tests for this feature, it is unknown if additional automation is required.
21	B	i - Agile expects and manages change and each iteration will require more and more regression testing. If automation was not used, then the team's velocity would be reduced. v - Automation tools are linked to continuous integration tools that will execute and will highlight instantaneously if the new code breaks the build.
22	D	ii - The earlier the agile team gets feedback on quality, the better. iii - Automation tools are linked to continuous integration tools that will execute and will highlight instantaneously if the new code breaks the build.
23	C	ii - This activity is expected of the agile tester. iv - This activity is typical for an agile tester.
24	C	It is the Scrum Master's role (or what the equivalent role is called in other agile methodologies) to produce and update the burndown chart from the information supplied by the rest of the team.
25	C	The burndown chart shows progress of the user stories that are complete (done) and an estimate of the remaining time to complete the rest of the user stories in the sprint.
26	D	This is true of BDD – not TDD.
27	C	The test pyramid emphasizes having more tests at the lower levels and a decreasing number of tests at the higher levels.
28	A	The testing quadrants can be used as an aid to describe the types of tests to all stakeholders.
29	C	Q1 – Incorrect – These test cases are not technology-facing component tests. Q2 – Incorrect – Usability and performance tests are not part of the 2nd quadrant. Q3 – Correct – Usability testing is part of the 3rd quadrant. Q4 – Correct – Performance testing is part of the 4th quadrant.
30	B	The decision to modify the test automation framework and scripts should be done collaboratively with the whole team. The tester is then responsible to make changes to the iteration plan as required.
31	A	The information from the risk analysis is used during poker planning sessions to determine priorities of items to be completed in the iteration. Only after the poker planning sessions, would items be added to the backlog if it is determined that not all items can be completed in the iteration.
	D	Risk mitigation can be done before test execution occurs to reduce the level of risk.
32	B	Planning poker sessions should continue for the user story, until the entire team is satisfied with the estimated effort.
33	C	i. This is helpful since we know there is a new version of the standard; existing test cases will need to be modified or new ones will need to be added. ii. This is helpful during the risk analysis phase. v. This is helpful during the risk analysis phase.
34	C	The obtained test coverage with supporting information makes it the best choice, even if more information would be needed. This includes information about found defects, their severity of occurrence, and taxonomy (how many serious problems in each area). This information gives a more complete basis for a release decision. You would also need information regarding the evaluated characteristics and how they affect the total picture regarding the completion of the system, and the related testing.
35	B, C	**A.** Incorrect – not testable, there are no details on the type of white box testing or the coverage expected. **B.** Correct – this is testable. **C.** Correct – this is testable. **D.** Incorrect – not testable, we do not know what is a reasonable response time. **E.** Incorrect – not testable, need to specify which browsers. One could make assumptions on what the major browsers are.

36	D	i. Incorrect – User story is specific to customers' transaction history. ii. Correct – This test is specific to a bank teller role and results in viewing customer's bank transactions. iii. Correct – This test is specific to a bank teller role and results in viewing customer's bank transactions. iv. Correct – This test is specific to a bank teller role and results in viewing customer's bank transactions. v. Incorrect – User story does not mention performance requirements.
37	C	**A.** Incorrect – The focus of this user story is not on the state of the system; instead the expectation is to test shipping costs. **B.** Incorrect – The focus of this user story is not on whether the item is shipped as expected; the expectation is to test shipping costs. **C.** Correct – BVA is the best option when testing shipping costs. **D.** Incorrect – The focus of this user story is not on whether the item is shipped as expected, the expectation is to test shipping costs.
38	A	**A.** Correct – This is not a valid reason because exploratory testing cannot prevent defects from occurring due to the concurrent, reactionary nature of analysis, design and execution of the tests.
39	C	**C.** Correct – This is one of many purposes of an ALM tool, but using the tool allows more collaboration with distributed teams than physical task boards.
40	B	**B.** Correct – Test charters are created prior to execution which include test objectives and test ideas.

5.3 Questions – SET 2

1. Which of the following best describes the approach for determining the iteration (timebox) length?
A. Iterations (timeboxes) should always be 30 days
B. The team determines iteration (timebox) length by dividing the total number of story points by the average velocity of the team
C. Iterations (timeboxes) should always be two weeks
D. The team should agree on the length of the iteration (timebox), taking the size and complexity of the project into consideration

2. Which of the following is a characteristic of an Agile leader?
A. Task focused
B. Process oriented
C. Supportive
D. Disengaged

3. Who is responsible for prioritizing the product backlog?
A. Product Owner
B. Project Manager
C. Lead Developer
D. Business Analyst

4. What are the advantages of maintaining consistent iteration (timebox) length throughout the project?
A. It helps to establish a consistent pattern of delivery
B. It helps the team to objectively measure progress
C. It provides a consistent means of measuring team velocity
D. All of the above

5. Tracking project issues in an Agile project is the primary responsibility of the…
A. Tester
B. Project Leader
C. Functional Manager
D. Developer

6. Why is it important to trust the team?
A. High trust teams do not have to be accountable to each other
B. High trust teams do not require a user representative
C. The Project Manager does not then have to keep a project schedule
D. The presence of trust is positively correlated with the team performance

7. An effective workshop facilitator will always ...
A. Involve the whole project team in all project workshops
B. Agree the process and participants of the workshop with the workshop owner before the workshop
C. Involve only those team members who will commit to doing further work after the workshop
D. Act as a proxy for any invited participant who is unable to attend the workshop on the day

8. Which of the following best represents the Agile approach to planning?
A. Planning is not part of an Agile approach, because Agile is exploratory
B. Planning should be done in detail at the outset of a project and not revisited
C. Planning should involve the whole team, not just the Project Manager
D. Planning should all be done by the Project Manager

9. Who should define the business value of a Feature within an Agile project?
A. The individual end-users
B. The Product Owner
C. The Business Analyst
D. The Business Sponsor

10. If a timebox (iteration) plan needs to be reprioritised in a hurry, who should re-prioritise?
A. The developers alone (they know what the customer wants)
B. The Product Owner (the developers would only choose the easy things as top priority)
C. The Project Leader (they can give an independent, pragmatic view)
D. The whole team including Product Owner and developers (together they can consider both business value and practicality)

11. What is the effect of having a large visible project plan on a wall?
A. It removes the need to create any other reports for management
B. It continuously communicates progress within the team and to other stakeholders
C. It allows the Project Manager to allocate tasks to specific team members
D. It is restrictive, as it does not allow the team to innovate and change

12. How should work be allocated to the team in an Agile project?
A. The Team Leader (Scrum Master) should allocate specific tasks to individuals
B. Tasks should be randomly allocated to team members, using Planning Poker
C. Team members should self-select tasks appropriate to their skills
D. The most complex tasks should be allocated by the Team Leader (Scrum Master)

13. What should the developers do if the customer representative is repeatedly too busy to be available?
A. Continue the work, record the assumptions and ask the customer later for input.
B. Send the customer a written warning that the end product will be completed on time, but may not meet their needs
C. Allow the Business Analyst to take on the role of Proxy Customer Representative
D. Draw the problem to the attention of the Scrum Master (Team Leader)

14. Which one of the following is a key feature of documentation that you would expect to find in an Agile project?
A. System documentation created at the end of each increment, at the start of the deployment
B. User Stories held in a spreadsheet or specialist database, where full details of user conversations are recorded for future purposes, like handover to maintenance or support
C. User Story cards containing only enough detail for planning and development, which will need to be supplemented by further face-to-face conversations
D. No written documentation, as all good communication is face-to-face

15. When handling team dynamics, the Agile Leader should …
A Empower the team members, within appropriate limits
B. Encourage an environment of competition and personal advantage
C. Give clear directives to the team about what they should do and how
D. Expect team members to be proactive and each work to their own priorities and objectives

16. Which one of the following statements is correct regarding acceptance of any deliverables on an Agile Project?
A. The team should allow only senior managers to sign off deliverables
B. The team should get acceptance of project deliverables from the appropriate stakeholders at least at the end of every timebox / iteration
C. The team should get acceptance of project deliverables from the users during a UAT phase at the end of the project
D. Acceptance of any particular deliverable on the project is gained from all stakeholders at the same time.

17. Which one of the following statements is correct regarding quality of deliverables from an Agile Project?
A. The products produced by an Agile project should be cheaper than those produced by any other approach, but quality will suffer
B. The products will be more expensive than by any other approach but will be top quality
C. The products will be fit for purpose, but may not do what the customer wanted

D. The products will be of appropriate quality, as guided by the customer representative involved throughout the development process

18. What is the Agile approach to doing design early in a project?
A. A big design up front is always a good idea
B. Just enough design up front gives a good foundation to start from and helps to mitigate risk, without wasting unnecessarily time
C. No design up front is the best approach as most of the fun of a project is in discovery of the unexpected
D. Design has no place in an Agile project

19. An Agile approach advocates which of the following approaches?
A. Get something “quick and dirty” delivered, to save time
B. Get something simple released as quickly as possible
C. Get something business-valuable delivered as quickly as possible, consistent with the right level of quality
D. Get something delivered once it has been fully documented and the documentation has been signed off as complete

20. Which of these best describes the Agile approach to team-working?
A. The team should plan to work a small amount of overtime regularly throughout the project
B. The team should expect to work longer hours towards the end of the sprint (timebox), in order to deliver all that was committed to
C. The team should strive for a sustainable pace and a normal working week
D. The team will “burn out” if they have to work overtime for more than two sprints (timeboxes, iterations) in a row

21. Which one of the following statements about workshops is true for Agile projects?
A. All project stakeholders should attend requirements workshops
B. Retrospectives are only run at the end of a project
C. It is best if the Project Manager facilitates the project’s workshops
D. An independent facilitator will manage the structure of a facilitated workshop but not input to the content

22. Which one of the following is an important feature of the daily stand-up / wash up / Scrum meeting?
A. Everyone is expected to stand for the whole time, to keep the meeting short
B. The meeting must be kept short and well structured
C. The meeting should ensure that it is clear to all which team members are not performing
D. No-one is allowed to leave the stand-up meeting until all problems raised have been solved

23. Who should attend the stand-up meetings?
A. Sponsor and Executive Management only
B. Project Manager and Technical Leads only
C. Project Leader and Customer Representatives only
D. The entire team

24. One of the development stages you would expect to see a team go through is:
A. Storming
B. Warming
C. Cloning
D. Yawning

25. When estimating is done for a project, the developers should:
A. Be fully involved in the estimating process
B. Be in total control of the estimating process
C. Be consulted after the Team Leader (Scrum Master) has made the estimates for the team’s work

D. Not make estimates unless velocity is already known

26. During an iteration (sprint) (timebox) the developers should be:
A. Able to contact the customer to clarify aspects of the work
B. Completely uninterrupted by the customer
C. In twice-daily contact with the customer
D. Able to work without needing to disturb the customer

27. The end result of an Agile development is:
A. A product of a professional quality which fits the business need
B. A product of almost as good a quality as a Waterfall development
C. A product which is barely sufficient for its purpose and deliberately not maintainable
D. A technically-perfect, re-factored solution

28. An Agile customer …
A. Must have a thorough understanding of Agile techniques, for Agile to work
B. Will always receive lower-quality products than their non-Agile counterparts
C. Will typically get business value delivered early and often
D. Will need to understand the technical aspects of development, to contribute effectively

29. An Agile team …
A. Is self-organizing, with each member having the same technical skills
B. Collaborates and supports its team members
C. Ensures that weak members of the team are allocated the simpler tasks
D. Ensures blame is allocated fairly

30. The Agile process …
A. Encourages the team to meet regularly
B. Has no meetings
C. Has lengthy reporting requirements
D. Has no reporting requirements

31. The Agile Leader …
A. Should allocate tasks to the team members each day at the stand-up meeting
B. Should involve the team in their own work-allocation
C. Should give detailed work-plans to the team each day
D. Should direct the work of the team, if they are inexperienced

32. What is Kan Ban?
A. A list of activities banned by the team, in relation to Team Norms
B. The set of Can Have stories for a project
C. A visible chart of work to do, work in progress and work done
D. A graph of tasks partially-completed by the team

33. What is meant by “Yesterday’s Weather” in an Agile project?
A. Teams work less well when it rains
B. Keeping metrics of earlier work to help with future estimates
C. Retrospectives should include less important topics, such as the weather, as ice-breakers
D. Estimating is as futile as predicting the weather

34. In Agile projects, we plan to “learn as we go” because…
A. It creates a better relationship between the developers and customer representatives
B. Many projects are evolutionary, and a better solution emerges this way
C. It is time-consuming to analyse everything at the beginning of a project
D. It prevents late delivery of the project

35. The recommended approach to design in an Agile project is:
A. No design up front
B. Big design up front
C. Just enough design up front

D. Use a previous design – it will be “good enough”

36. What is the personal risk that an Agile Leader takes in empowering the team?
A. The Agile Leader might lose their job, as the team is doing all the work
B. If the team fails, the Agile leader will not get a performance bonus
C. The Agile Leader has less direct control over the team’s work, but still has the responsibility for their outcomes
D. The Agile Leader cannot share the glory of team success

37. The Agile approach to documentation is:
A. Do no documentation because it is a waste of time
B. Do sufficient documentation to prove you have done a good job
C. Do the necessary documentation to support the development and use of the product
D. Do more documentation than usual, because Agile is risky

38. The Agile way is:
A. To produce working product of the right quality, early and incrementally
B. To produce working product after documentation has been signed off
C. To produce simple prototypes early, but no finished product until the end of the project
D. To produce products without technical integrity, but re-engineer later

39. The customer in an Agile project ….
A. Has no control over the prioritization of delivered features
B. Has total control over the prioritization of features
C. Collaborates with the developers over prioritization of features, but the developers have the final decision
D. Collaborates with the developers over prioritization of features, but the business has the final decision

40. In the popular prioritization technique called “MoSCoW”, the “M” stands for ...
A. May have
B. Major
C. Must Have
D. Mandatory

41. The working culture of an Agile team is …
A. Collective
B. Collaborative
C. Connective
D. Contemplative

42. The leadership style of an Agile Leader is …
A. Directive
B. Assertive
C. Facilitative
D. Feature-based

43. The Agile Manifesto states the following values:
A. People are more important than contracts
B. Working software should have priority over comprehensive documentation
C. Plans should have priority over ability to respond
D. Contracts should be negotiated which allow control over the people

44. Which of the following are attributes of an Agile team?
A. Courage to change and adapt
B. Trust of fellow team members to do the work
C. Responsiveness to change
D. All of these

45. A sustainable pace means …

A. If the team members work long hours regularly they will get used to it, and be able to sustain it
B. A 40 hour week is only for the weaker members of the team. Others can do more.
C. The team should establish a velocity which can be sustained within normal working hours
D. Working long hours is the only way to deliver on time

46. A burn-down chart shows …
A. The energy level and velocity of the team
B. The remaining work (effort, points) to complete before the iteration (timebox) end
C. The number of hours worked by each team member during the iteration (timebox)
D. The rate of spending of the budget for a project

47. The reason for holding regular Retrospectives is:
A. It allows the team to take a necessary break from work
B. It gives management information to use in team members' performance reviews
C. It allows learning which can be used to improve team performance during the project
D. It prevents deviation from the process which the team has been following

48. Once a project is underway, the approach to planning is:
A. Plans should never be changed
B. It is normal to need to plan and re-plan as the project progresses
C. Plans should only be changed with full approval of all stakeholders
D. Plans are not required as Agile is incremental

49. An Agile project …
A. Should have no control over its progress
B. Should be able to demonstrate control of its progress
C. Is always out of control
D. Is controlled by the velocity of the most junior team member

50. An Agile project should have ...
A. Occasional early deliveries, if the business is prepared to accept lower quality
B. A regular pattern of delivery of developer-focused products
C. A regular pattern of delivery of business-valued increments
D. An irregular and unpredictable delivery of products

51. When an Agile team is successful …
A. It should be encouraged to celebrate success only when the project is over
B. It should be encouraged to celebrate even small successes immediately
C. It should not celebrate success, as this wastes project budget
D. It should not celebrate success, as this makes less successful teams feel bad

52. In order to communicate well, the Agile project should ...
A. Keep team-size large, to avoid stakeholders feeling left out
B. Break the project into small, mixed-skill, self-organising teams
C. Operate with one team of less than 10 people
D. Operate with separate customer, developer and test teams

53. If a new requirement emerges once an Agile project is running, it should be:
A. Automatically included in the work of the project
B. Automatically excluded and left until a later project or increment
C. Assessed for importance and, if sufficiently important to the business, included in the project, displacing less important requirements
D. Put on the backlog for consideration by the wider group of stakeholders after the project has been completed

54. You have been engaged as the Technical Coordinator in a product development team. The customer (Product Owner) and Team Leader (Scrum Master) are happy because the team always delivers business value on time. However, you worry that the technical debt is increasing. What would be your primary goal to ensure the right quality?

A. Ensure testers define their entry criteria for quality, which they impose on the development team
B. Nothing. Prescribed roles such as technical coordinators are not part of an Agile team
C. Make sure that the maintainability quality attribute is addressed
D. On time delivery and happy end users are the only quality measures in Agile development

55. How could maintainability of the developing product be improved in a development team?
A. Apply standard design patterns
B. All of these
C. Make refactoring a common practice
D. Ensure unit testing is included in the “done” criteria

56. Agile methods are described as “adaptive” because…
A. Agile teams have the empowerment to frequently respond to change and to learn on a project by changing the plan
B. The rate of development progress on an Agile project is constantly tracked to allow adaptation
C. Project Managers are not needed in Agile methods because teams are self-organising
D. Workshops held at the beginning and the end of every iteration (timebox) allow the team to adapt the product specification

57. What do all Agile approaches have in common?
A. A prescribed, fixed iteration (timebox) length
B. Iterative development and incremental delivery
C. A strict focus on on-time delivery
D. A large set of clearly defined roles

58. What is one difference in responsibility between a Project Manager and a Scrum Master (Team Leader) in an Agile project?
A. None. It's basically the same. Scrum Master (or Team Leader) is just a better term than Project Manager in an Agile project
B. The Project Manager creates the detailed delivery plans while the Team Leader monitors execution within the team
C. Project Manager communicates with project governance authorities when necessary
D. The Project Manager monitors the realisation of benefits in the business case.

59. How could you benefit from having an End User Representative in your development team?
A. End users should NOT be in the development team. Requirements are always communicated to the developers by the Product Owner, who is part of the customer team
B. The End User Representative will be solely responsible for acceptance tests within the team
C. The End User Representative assures that user stories are documented properly
D. The End User Representative will be able to clearly tell the developers what will work for an end user

60. The responsibilities of a Product Owner will include …
A. Business processes diagramming
B. Prioritizing requirements
C. Managing the project budget
D. All of these

5.4 Answers – SET 2

Question	Answer
1	D
2	C
3	A
4	D
5	B
6	D
7	B
8	C
9	B
10	D
11	B
12	C
13	D
14	C
15	A
16	B
17	D
18	B
19	C
20	C
21	D
22	B
23	D
24	A
25	A
26	A
27	A
28	C
29	B
30	A
31	B
32	C
33	B
34	B
35	C
36	C
37	C
38	A
39	D
40	C
41	B
42	C
43	B
44	D
45	C

46	B
47	C
48	B
49	B
50	C
51	B
52	B
53	C
54	C
55	B
56	A
57	B
58	C
59	D
60	B

6 Lessons learned

- Even though not mentioned as one of the values in the Agile Manifesto, Developers using unit testing tools to support the testing process is normal practice, especially in test driven development.
- Testers are not always responsible for developing unit tests which they pass on to the developers for testing, as depending on the skillset of the team; developers may take on this task.
- Developers are not expected to test non-functional requirements (performance, usability, security, etc.) but they may help with these tasks depending on the skillset of the team and individual workload.
- Specialized testers are still needed and are an important resource on agile projects.
- Developers only implement features that are requested by business and are part of an iteration. If they complete their tasks, they will help out with other tasks assigned to the iteration.
- The same number of defects may be found using any software development process. The benefit with agile is the ability to find and fix defects faster.
- There may not be enough time to complete all features for a given iteration, but the agile process does allow the team to focus on those features that have the highest business value.
- Agile does not single out individuals; it is about the whole team.
- It is important to consider testability and automation, but designing the application so that it reduces testing effort may not result in a suitable solution for the end-user.
- Product Owner is responsible for creating and prioritising the acceptance criteria.
- Testers should make it a point to attend the retrospective meeting so that they can learn valuable information to apply in subsequent iterations.
- Retrospective meeting can be used as a process improvement measure.
- One of the principles of continuous integration is that builds are done at least once per day with automatic deploy and execution of automated unit & integration tests.
- Continuous integration allows for constant availability of an executable software at any time and any place, for testing, demonstration, or education purposes.
- The Continuous Integration practice enables developers to integrate work constantly, and test constantly, so errors in code can be detected rapidly.
- The tester participates in the creation of the user story.
- The user story should include both functional and non-functional requirements.
- The user story is written collaboratively by the developers, testers, and business representatives.
- Agile testing promotes lightweight documentation.
- Features should be verified in the same iteration in which they are developed.
- Test level entry and exit criteria are more closely associated with traditional lifecycles.
- Testers should work closely with developers while retaining an objective outlook.
- Developers and testers work collaboratively to develop and test a feature.
- A selection of users may perform beta testing on the product after the completion of a series of iterations.
- There can be a risk of losing test independence for organizations introducing agile.
- The independent test team can be part of another team when there are some specialized testers working on non-sprint or long term activities.
- Independent testing can be introduced at the end of a sprint which preserves a level of independence where there are separate test and development teams and testers are assigned on-demand at the end of a sprint.
- Positive customer feedback and working software are key indicators to product quality.
- Burndown charts is best at showing the team’s progress against estimates as it shows the planned progress and release date together with the actual progress of the user stories.
- Agile task board shows progress, this information is then used in the burndown chart. But the task board showing the progress of the user stories and tasks cannot show progress against estimates.
- Automation is essential within agile projects so that teams maintain or increase their velocity.

- Automation tools are linked to continuous integration tools that will execute and will highlight instantaneously if the new code breaks the build. Hence Automation is needed to ensure that code changes do not break the software build.
- Agile projects embrace and expect change; however this does not mean requirements change daily.
- Pair programming is typically done using two developers; testers are not expected to improve program logic although could review code for testability or maintainability.
- In agile, defects are communicated regularly with stakeholders.
- The tester should coach all other team members in any testing related aspect.
- Within agile, the tester will provide feedback on the product at all stages, which might include code analysing activities.
- The burndown chart shows progress of the user stories that are complete (done), and an estimate of the remaining time to complete the rest of the user stories in the sprint.
- Test-Driven Development (TDD) is a technique used to develop code guided by automated test cases. It is also known as test first programming, since tests are written before the code. The tests are automated and are used in continuous integration.
- The process for TDD is repeated for each small piece of code, running the previous tests as well as the added tests. The tests serve as a form of executable design specification for future maintenance efforts.
- The test pyramid emphasizes having more tests at the lower levels and a decreasing number of tests at the higher levels. It refers to situation where the number of automated unit tests is higher than the number of automated tests for higher test levels.
- The testing quadrants can be used as an aid to describe the types of tests to all stakeholders. It should not be used as metric for risk level or test coverage. In some situations, there may not be any tests for a quadrant.
- Planning poker sessions are held first to determine what can be accomplished in the given iteration. If it is determined that there is not enough time to complete all items, it is probable that the lower risk items will be added to the backlog for future sprints.
- The entire team must agree on the estimate for the user story. The customer alone does not understand the complexity of developing or testing the functionality.
- Exploratory testing is known as an experienced based approach to testing, which will be as effective as the tester running the tests. The benefit of this approach is that the tests that will be designed and executed will influence the next set of tests that are designed and executed.
- Exploratory testing is not a technique but an approach to testing that can use other techniques such as pairwise, classification trees, boundary value analysis etc.
- One of the benefits of using exploratory testing is when there are requirements that are less than perfect, and within agile projects there is limited analysis, depth and detail of requirements.
- Exploratory testing encompasses concurrent learning, test design, and execution.
- Best results are achieved when exploratory testing is combined with other test strategies.
- Exploratory tester needs good understanding of how the system is used and how to determine when it fails.
- A Wiki tool allows teams to build up a knowledge base on tools and techniques for development and testing activities.
- A Continuous Integration (CI) tool provides quick response about the build quality and details about code changes.
- A data generation and data load tool generates and loads large volumes and combinations of data to use for testing.

7 Glossary

A
abstract test case: See *high level test case*.
acceptance: See *acceptance testing*.
acceptance criteria: The specific criteria identified by the customer for each functional requirement. The acceptance criteria, written in easily to understand terms and from the customer's perspective, provides additional detail into how a feature should work and assesses the ability of the feature to perform its intended function.
acceptance testing: Formal testing with respect to user needs, requirements, and business
processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.
accessibility testing: Testing to determine the ease by which users with disabilities can use a component or system.
accuracy: The capability of the software product to provide the right or agreed results or effects with the needed degree of precision. See also *functionality*.
accuracy testing: The process of testing to determine the accuracy of a software product. See also *accuracy.*
acting (IDEAL): The phase within the IDEAL model where the improvements are developed, put into practice, and deployed across the organization. The acting phase consists of the activities: create solution, pilot/test solution, refine solution and implement solution. See also *IDEAL*.
action word driven testing: See *keyword-driven testing*
actor: User or any other person or system that interacts with the system under test in a specific way.
actual outcome: See *actual result*.
actual result: The behavior produced/observed when a component or system is tested.
ad hoc review: See *informal review*.
Actual Time Estimation: A time-based method of estimating development work. The intent of this method is to best approximate the amount of time required to complete a given development task. Generally, these estimates are calculated using Ideal Engineering Hours.

Examples:
This task will be complete in 10 days. Or...
This task will be complete by January 10th. Or...
This task will require 25 development hours for completion

ad hoc testing: Testing carried out informally; no formal test preparation takes place, no recognized test design technique is used, there are no expectations for results and arbitrariness guides the test execution activity.
adaptability: The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered. See also *portability*.
agile: Often used as the abbreviation for Agile Software Development or Agile Methods. Agile is a generic term which refers to a collection of lightweight software development methodologies that value and support evolving requirements through iterative development, direct Customer/Developer communication and collaboration, self organizing cross-functional teams and continuous improvement through frequent inspection and adaptation.
agile manifesto: A statement of the principles and values that support the ideals of Agile Software Development. The manifesto was drafted in February 2001 at the Snowbird Ski Resort located in the state of Utah. Users of Agile can become signatories of this manifesto at http://www.agilemanifesto.org

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

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

That is, while there is value in the items on the right, we value the items on the left more.
agile software development: A group of software development methodologies based on iterative incremental development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.
agile testing: Testing practice for a project using agile software development methodologies, incorporating techniques and methods, such as extreme programming (XP), treating development as the customer of testing and emphasizing the test-first design paradigm. See also *test-driven development*.
AgileUP: Short for Agile Unified Process, it refers to a simplified version of the IBM Rational Unified Process (RUP). Much like RUP, AgileUP is an iterative process, that utilizes several Agile techniques and ideas for the purpose of developing business applications. The most noticeable difference between the two processes is that AgileUP only defines 7 key disciplines to RUP's 9. Those disciplines are:

1. Model: The goal of this discipline is to understand the business of the organization, the problem domain being addressed by the project, and to identify a viable solution to address the problem domain.
2. Implementation: The goal of this discipline is to transform your model(s) into executable code and to perform a basic level of testing, in particular unit testing.
3. Test: The goal of this discipline is to perform an objective evaluation to ensure quality. This includes finding defects, validating that the system works as designed, and verifying that the requirements are met.
4. Deployment: The goal of this discipline is to plan for the delivery of the system and to execute the plan to make the system available to end users.
5. Configuration Management: The goal of this discipline is to manage access to your project artifacts. This includes not only tracking artifact versions over time but also controlling and managing changes to them.
6. Project Management: The goal of this discipline is to direct the activities that take place on the project. This includes managing risks, directing people (assigning tasks, tracking progress, etc.) and coordinating with people and systems outside the scope of the project to be sure that it is delivered on time and within budget
7. Environment: The goal of this discipline is to support the rest of the effort by ensuring that the proper process, guidance (standards and guidelines), and tools (hardware, software, etc.) are available for the team as needed.

algorithm test: See *branch testing*.
alpha testing: Simulated or actual operational testing by potential users/customers or an independent test team at the developers' site, but outside the development organization. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing.
analytical testing: Testing based on a systematic analysis of e.g., product risks or requirements.
analyzability: The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified. See also *maintainability*.
analyzer: See *static analyzer*.
anomaly: Any condition that deviates from expectation based on requirements specifications, design documents, user documents, standards, etc. or from someone's perception or experience. Anomalies may be found during, but not limited to, reviewing, testing, analysis, compilation, or use of software products or applicable documentation. See also *bug, defect, deviation, error, fault, failure, incident, problem*.
anti-pattern: Repeated action, process, structure or reusable solution that initially appears to be beneficial and is commonly used but is ineffective and/or counterproductive in practice.
API: Acronym for Application Programming Interface.
API testing: Testing performed by submitting commands to the software under test using programming interfaces of the application directly.
Application Lifecycle Management: Often abbreviated as ALM, it is generally used in reference to tools that help facilitate the coordination of business management and software engineering practices by integrating features related to requirements management, architecture management, coding, testing, defect tracking and release management into a single solution.
arc testing: See *branch testing*.

assessment report: A document summarizing the assessment results, e.g. conclusions, recommendations and findings. See also *process assessment.*
assessor: A person who conducts an assessment; any member of an assessment team.
atomic condition: A condition that cannot be decomposed, i.e., a condition that does not contain two or more single conditions joined by a logical operator (AND, OR, XOR).
attack: Directed and focused attempt to evaluate the quality, especially reliability, of a test object by attempting to force specific failures to occur. See also *negative testing.*
attack-based testing: An experience-based testing technique that uses software attacks to induce failures, particularly security related failures. *See also* attack.
attractiveness: The capability of the software product to be attractive to the user. See also *usability*.
audit: An independent evaluation of software products or processes to ascertain compliance to standards, guidelines, specifications, and/or procedures based on objective criteria, including documents that specify:
(1) the form or content of the products to be produced
(2) the process by which the products shall be produced
(3) how compliance to standards or guidelines shall be measured.
audit trail: A path by which the original input to a process (e.g. data) can be traced back through the process, taking the process output as a starting point. This facilitates defect analysis and allows a process audit to be carried out.
automated testware: Testware used in automated testing, such as tool scripts.
availability: The degree to which a component or system is operational and accessible when required for use. Often expressed as a percentage.

B
back-to-back testing: Testing in which two or more variants of a component or system are executed with the same inputs, the outputs compared, and analyzed in cases of discrepancies.
balanced scorecard: A strategic tool for measuring whether the operational activities of a company are aligned with its objectives in terms of business vision and strategy. See also *corporate dashboard, scorecard.*
baseline: A specification or software product that has been formally reviewed or agreed upon, that thereafter serves as the basis for further development, and that can be changed only through a formal change control process.
basic block: A sequence of one or more consecutive executable statements containing no branches. Note: A node in a control flow graph represents a basic block.
basis test set: A set of test cases derived from the internal structure of a component or specification to ensure that 100% of a specified coverage criterion will be achieved.
bebugging: See *fault seeding*.
behavior: The response of a component or system to a set of input values and preconditions.
benchmark test: (1) A standard against which measurements or comparisons can be made. (2) A test that is be used to compare components or systems to each other or to a standard as in (1).
bespoke software: Software developed specifically for a set of users or customers. The opposite is off-the-shelf software.
best practice: A superior method or innovative practice that contributes to the improved performance of an organization under given context, usually recognized as 'best' by other peer organizations.
beta testing: Operational testing by potential and/or existing users/customers at an external site not otherwise involved with the developers, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes. Beta testing is often employed as a form of external acceptance testing for off-the-shelf software in order to acquire feedback from the market.
big-bang testing: An integration testing approach in which software elements, hardware elements, or both are combined all at once into a component or an overall system, rather than in stages.See also *integration testing*.
black box technique: See *black box test design technique*.
black box test design technique: Procedure to derive and/or select test cases based on an
analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

black box testing: Testing, either functional or non-functional, without reference to the internal structure of the component or system.
blocked test case: A test case that cannot be executed because the preconditions for its execution are not fulfilled.
bottom-up testing: An incremental approach to integration testing where the lowest level components are tested first, and then used to facilitate the testing of higher level components. This process is repeated until the component at the top of the hierarchy is tested. See also *integration testing*.
boundary value: An input value or output value which is on the edge of an equivalence partition or at the smallest incremental distance on either side of an edge, for example the minimum or maximum value of a range.
boundary value analysis: A black box test design technique in which test cases are designed based on boundary values. See also *boundary value.*
boundary value coverage: The percentage of boundary values that have been exercised by a test suite.
boundary value testing: See *boundary value analysis*.
branch: A basic block that can be selected for execution based on a program construct in which one of two or more alternative program paths is available, e.g. case, jump, go to, if-then-else.
branch condition: See *condition*.
branch condition combination coverage: See *multiple condition coverage*.
branch condition combination testing: See *multiple condition testing*.
branch condition coverage: See *condition coverage*.
branch coverage: The percentage of branches that have been exercised by a test suite. 100% branch coverage implies both 100% decision coverage and 100% statement coverage.
branch testing: A white box test design technique in which test cases are designed to execute branches.
buffer: A device or storage area used to store data temporarily for differences in rates of data flow, time or occurrence of events, or amounts of data that can be handled by the devices or processes involved in the transfer or use of the data.
buffer overflow: A memory access failure due to the attempt by a process to store data beyond the boundaries of a fixed length buffer, resulting in overwriting of adjacent memory areas or the raising of an overflow exception. See also *buffer.*
bug: See *defect*.
bug report: See *defect report.*
bug taxonomy: *See defect taxonomy.*
bug tracking tool: See *defect management tool.*
build verification test: A set of automated tests which validates the integrity of each new
build and verifies its key/core functionality, stability and testability. It is an industry practice when a high frequency of build releases occurs (e.g., agile projects) and it is run on every new build before the build is released for further testing. See also *regression testing, smoke test*.
burndown chart: A publicly displayed chart that depicts the outstanding effort versus time in an iteration. It shows the status and trend of completing the tasks of the iteration. The X-axis typically represents days in the sprint, while the Y-axis is the remaining effort (usually either in ideal engineering hours or story points). A burn down chart is a simple, easy to understand graphical representation of "Work Remaining" versus "Time Remaining". Generally, "Work Remaining" will be represented on the vertical axis while "Time Remaining" is displayed along the horizontal axis. Burn down charts are effective tools for communicating progress and predicting when work will be completed. The burn down chart is also an effective means for teams to make adjustments in order to meet product/project delivery expectations.
business process-based testing: An approach to testing in which test cases are designed based on descriptions and/or knowledge of business processes.
BVT: See *build verification test*.

C
Cadence: Cadence, by definition is a noun that represents the flow or rhythm of events and the pattern in which something is experienced. In verb form, it is used to describe the idea of making something rhythmical. Cadence is something that Agile teams strive to achieve as it allows teams to operate efficiently and sustainably within the iterative cycles

that most Agile methods promote. In its simplest form, cadence allows Agile teams to focus on development and delivery of the product rather than on process.
call graph: An abstract representation of calling relationships between subroutines in a program.
Capability Maturity Model Integration: A framework that describes the key elements of an effective product development and maintenance process. The Capability Maturity Model Integration covers best-practices for planning, engineering and managing product development and maintenance.
Capacity: The measurement of how much work can be completed within a given, fixed time frame by estimating the number of available, productive work hours for an individual or team. To accurately estimate capacity, it is important to factor in all known variables such as meetings, holidays and vacations, as well as the effects of multi-tasking and normal administrative tasks.
capture/playback: A test automation approach, where inputs to the test object are recorded during manual testing in order to generate automated test scripts that could be executed later (i.e. replayed).
capture/playback tool: A type of test execution tool where inputs are recorded during manual testing in order to generate automated test scripts that can be executed later (i.e. replayed). These tools are often used to support automated regression testing.
capture/replay tool: See *capture/playback tool*.
CASE: Acronym for Computer Aided Software Engineering.
CAST: Acronym for Computer Aided Software Testing. See also *test automation*.
causal analysis: The analysis of defects to determine their root cause.
cause-effect analysis: See *cause-effect graphing*.
cause-effect decision table: See *decision table*.
cause-effect diagram: A graphical representation used to organize and display the interrelationships of various possible root causes of a problem. Possible causes of a real or potential defect or failure are organized in categories and subcategories in a horizontal tree-structure, with the (potential) defect or failure as the root node.
cause-effect graph: A graphical representation of inputs and/or stimuli (causes) with their associated outputs (effects), which can be used to design test cases.
cause-effect graphing: A black box test design technique in which test cases are designed from cause-effect graphs.
certification: The process of confirming that a component, system or person complies with its specified requirements, e.g. by passing an exam.
change control: See *configuration control.*
change control board: See *configuration control board*.
change management: (1) A structured approach to transitioning individuals, and organizations from a current state to a desired future state. (2) Controlled way to effect a change, or a proposed change, to a product or service. See also *configuration management.*
changeability: The capability of the software product to enable specified modifications to be implemented. See also *maintainability*.
charter: See *test charter.*
checker: See *reviewer*.
checklist-based testing: An experience-based test design technique whereby the experienced tester uses a high-level list of items to be noted, checked, or remembered, or a set of rules or criteria against which a product has to be verified.
Chickens and Pigs: From the popular Chickens and Pigs story by Ken Schwaber(see below). A "Chicken" is used to describe someone who, while involved in the process or project, is not committed and accountable for any specific deliverables. Chickens are often interested stake holders, managers and executives. As these individuals are not directly involved or accountable, it is encouraged that Chickens participation in the process is limited observation only. A "Pig", however, is an individual who is committed as they are directly accountable for specific project and product deliverables. Pigs are encouraged to wholly participate in the process as they will be accountable for the expectations set by their involvement and estimates.

The Chicken and Pig Story

A pig and a chicken are walking down a road. The chicken looks at the pig and says, "Hey, why don't we open a restaurant?" The pig looks back at the chicken and says, "Good idea, what do you want to call it?" The chicken thinks about it and says, "Why don't we call it 'Ham and Eggs'?" "I don't think so," says the pig, "I'd be committed, but you'd only be involved."

Chow's coverage metrics: See *N-switch coverage*.

classification tree: A tree showing equivalence partitions hierarchically ordered, which is used to design test cases in the classification tree method. See also *classification tree method.*
classification tree method: A black box test design technique in which test cases, described by means of a classification tree, are designed to execute combinations of representatives of input and/or output domains. See also *combinatorial testing.*
clear-box testing: See *white-box testing*.
CLI: Acronym for Command-Line Interface.
CLI testing: Testing performed by submitting commands to the software under test using a dedicated command-line interface.
CMMI: See *Capability Maturity Model Integration.*
code: Computer instructions and data definitions expressed in a programming language or in a form output by an assembler, compiler or other translator.
code analyzer: See *static code analyzer*.
code coverage: An analysis method that determines which parts of the software have been executed (covered) by the test suite and which parts have not been executed, e.g. statement coverage, decision coverage or condition coverage.
code-based testing: See *white box testing*.
codependent behavior: Excessive emotional or psychological dependence on another person, specifically in trying to change that person's current (undesirable) behavior while supporting them in continuing that behavior. For example, in software testing, complaining about late delivery to test and yet enjoying the necessary "heroism" working additional hours to make up time when delivery is running late, therefore reinforcing the lateness.
co-existence: The capability of the software product to co-exist with other independent software in a common environment sharing common resources. See also *portability.*
combinatorial testing: A means to identify a suitable subset of test combinations to achieve a predetermined level of coverage when testing an object with multiple parameters and where those parameters themselves each have several values, which gives rise to more combinations than are feasible to test in the time allowed. See also *classification tree method*, *n-wise testing, pairwise testing*, *orthogonal array testing*.
Commercial Off-The-Shelf software: See *off-the-shelf software*.
comparator: See *test comparator*.
compatibility testing: See *interoperability testing*.
compiler: A software tool that translates programs expressed in a high order language into their machine language equivalents.
complete testing: See *exhaustive testing*.
completion criteria: See *exit criteria*.
complexity: The degree to which a component or system has a design and/or internal structure that is difficult to understand, maintain and verify. See also *cyclomatic complexity.*
Complexity Points: Complexity points are units of measure, based on relative sizing, used to estimate development work in terms of complexity and/or size, versus traditional time based methods which attempt to measure the duration of time required to complete some unit of work. Complexity Points are similar to 'Story Points' (see Story Points) but the scale used for complexity points may vary based on the differences in implementation of this sizing approach.
compliance: The capability of the software product to adhere to standards, conventions or regulations in laws and similar prescriptions.
compliance testing: The process of testing to determine the compliance of the component or system.
component: A minimal software item that can be tested in isolation.
component integration testing: Testing performed to expose defects in the interfaces and
interaction between integrated components.
component specification: A description of a component's function in terms of its output values for specified input values under specified conditions, and required non-functional behavior (e.g.resource-utilization).
component testing: The testing of individual software components.
compound condition: Two or more single conditions joined by means of a logical operator (AND, OR or XOR), e.g. 'A>B AND C>1000'.
concrete test case: See *low level test case*.

concurrency testing: Testing to determine how the occurrence of two or more activities within the same interval of time, achieved either by interleaving the activities or by simultaneous execution, is
handled by the component or system.
condition: A logical expression that can be evaluated as True or False, e.g. A>B. See also *condition testing.*
condition combination coverage: See *multiple condition coverage*.
condition combination testing: See *multiple condition testing*.
condition coverage: The percentage of condition outcomes that have been exercised by a test suite. 100% condition coverage requires each single condition in every decision statement to be tested as True and False.
condition determination coverage: See *modified condition decision coverage*.
condition determination testing: See *modified condition decision testing.*
condition outcome: The evaluation of a condition to True or False.
condition testing: A white box test design technique in which test cases are designed to execute condition outcomes.
confidence interval: In managing project risks, the period of time within which a contingency action must be implemented in order to be effective in reducing the impact of the risk.
confidence test: See *smoke test*.
configuration: The composition of a component or system as defined by the number, nature, and interconnections of its constituent parts.
configuration auditing: The function to check on the contents of libraries of configuration items, e.g. for standards compliance. " "
configuration control: An element of configuration management, consisting of the evaluation, coordination, approval or disapproval, and implementation of changes to configuration items after formal establishment of their configuration identification.
configuration control board (CCB): A group of people responsible for evaluating and approving or disapproving proposed changes to configuration items, and for ensuring implementation of approved changes.
configuration identification: An element of configuration management, consisting of selecting the configuration items for a system and recording their functional and physical characteristics in technical documentation.
configuration item: An aggregation of hardware, software or both, that is designated for configuration management and treated as a single entity in the configuration management process.
configuration management: A discipline applying technical and administrative direction and
surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements.
configuration management tool: A tool that provides support for the identification and control of configuration items, their status over changes and versions, and the release of baselines consisting of configuration items.
configuration testing: See *portability testing*.
confirmation testing: Testing that runs test cases that failed the last time they were run, in order to verify the success of corrective actions.
conformance testing: See *compliance testing*.
consistency: The degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a component or system.
consultative testing: Testing driven by the advice and guidance of appropriate experts from outside the test team (e.g., technology experts and/or business domain experts).
content-based model: A process model providing a detailed description of good engineering practices, e.g. test practices.
content reference model: See *content-based model.*
Continuous Integration: The practice of continuously integrating new development code into the existing codebases. Continuous integration allows the development team to ensure that the code repository always reflects the latest working build of the software. As developers complete the coding of a feature, the feature is applied to the latest software build where it is validated for defects and integrated into the codebase previously delivered. Continuous integration practices generally include testing and build automation, resulting in an end-to-end integration suite.

continuous representation: A capability maturity model structure wherein capability levels provide a recommended order for approaching process improvement within specified process areas.
control chart: A statistical process control tool used to monitor a process and determine whether it is statistically controlled. It graphically depicts the average value and the upper and lower control limits (the highest and lowest values) of a process.
control flow: A sequence of events (paths) in the execution through a component or system.
control flow analysis: A form of static analysis based on a representation of unique paths
(sequences of events) in the execution through a component or system. Control flow analysis
evaluates the integrity of control flow structures, looking for possible control flow anomalies such as closed loops or logically unreachable process steps.
control flow graph: An abstract representation of all possible sequences of events (paths) in the execution through a component or system.
control flow path: See *path*.
control flow testing: An approach to structure-based testing in which test cases are designed to execute specific sequences of events. Various techniques exist for control flow testing, e.g., decision testing, condition testing, and path testing, that each have their specific approach and level of control flow coverage. *See also* decision testing, condition testing, path testing.
convergence metric: A metric that shows progress toward a defined criterion, e.g., convergence of the total number of test executed to the total number of tests planned for execution.
conversion testing: Testing of software used to convert data from existing systems for use in replacement systems.
corporate dashboard: A dashboard-style representation of the status of corporate performance data. See also *balanced scorecard, dashboard.*
cost of quality: The total costs incurred on quality activities and issues and often split into prevention costs, appraisal costs, internal failure costs and external failure costs.
COTS: Acronym for Commercial Off-The-Shelf software. See *off-the-shelf software.*
ETAE **coverage:** The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite.
coverage analysis: Measurement of achieved coverage to a specified coverage item during test execution referring to predetermined criteria to determine whether additional testing is required and if so, which test cases are needed.
coverage item: An entity or property used as a basis for test coverage, e.g. equivalence partitions or code statements.
coverage measurement tool: See *coverage tool.*
coverage tool: A tool that provides objective measures of what structural elements, e.g. statements, branches have been exercised by a test suite.
critical success factor: An element necessary for an organization or project to achieve its mission. Critical success factors are the critical factors or activities required for ensuring the success.
Critical Testing Processes: A content-based model for test process improvement built around twelve critical processes. These include highly visible processes, by which peers and management judge competence and mission-critical processes in which performance affects the company's profits and reputation. See also *content-based model.*
Crystal: Crystal (sometimes referred to as Crystal Clear) is a lightweight, Agile software development framework developed originally by Alistair Cockburn. Crystal, as a basic matter of principle, is primarily focused on the collaboration and interactions between teams of people rather than the processes and artifacts of traditional methodologies. Crystal methods value:

- Frequent delivery of working software
- Continuous, reflective improvement
- Osmotic communication via team colocation
- Technical excellence by utilizing automated testing, configuration management and frequent integration

CTP: See *Critical Testing Processes.*
custom software: See *bespoke software*.
custom tool: A software tool developed specifically for a set of users or customers.

Customer Unit: The Customer Unit refers to the people and roles that define and/or represent the voice and expectations of the primary consumers of the deliverables produced throughout the course of the project. Product Managers, Sales, Marketing, Executives and End-User Customers are all examples of roles and titles that often comprise the Customer Unit. In Agile projects, the Customer Unit is typically responsible for setting the vision, project charter and roadmap, creating and maintaining product backlog requirements and priorities, defining user acceptance criteria and communicating with the Developer Unit.
cyclomatic complexity: The maximum number of linear, independent paths through a program.
Cyclomatic complexity may be computed as: L – N + 2P, where
- L = the number of edges/links in a graph
- N = the number of nodes in a graph
- P = the number of disconnected parts of the graph (e.g. a called graph or subroutine)
cyclomatic number: See *cyclomatic complexity*.

D
daily build: A development activity whereby a complete system is compiled and linked every day (often overnight), so that a consistent system is available at any time including all latest changes.

Daily Scrum: The Daily Scrum, also referred to as 'the daily stand-up', is a brief, daily communication and planning forum, in which Agile/Scrum teams come together to evaluate the health and progress of the iteration/sprint. It is also considered to be the fifth and final level of the Agile planning process. As a daily, team planning meeting, an effective daily scrum should be a tightly focused and time boxed meeting that occurs at the same time and place, on a daily basis. The intent of the daily scrum is to better understand the progress of the iteration, by all contributing team members honestly answering the following three questions:

1. What did I accomplish yesterday?
2. What will I commit to, or complete, today?
3. What impediments or obstacles are preventing me from meeting my commitments?

Conversation in these meetings should remain focused on answering these three questions only. For the sake of brevity and greater team efficiency, additional discussion stemming from these three questions should be handled independently of the daily scrum, and should be limited to those team members who are directly involved.
dashboard: A representation of dynamic measurements of operational performance for some
organization or activity, using metrics represented via metaphors such as visual 'dials', 'counters', and other devices resembling those on the dashboard of an automobile, so that the effects of events or activities can be easily understood and related to operational goals. See also *corporate dashboard, scorecard.*
data definition: An executable statement where a variable is assigned a value.
data-driven testing: A scripting technique that stores test input and expected results in a ATT table or spreadsheet, so that a single control script can execute all of the tests in the table. Data-driven testing is often used to support the application of test execution tools such as capture/playback tools. See also *keyword-driven testing*.
data flow: An abstract representation of the sequence and possible changes of the state of data objects, where the state of an object is any of: creation, usage, or destruction.
data flow analysis: A form of static analysis based on the definition and usage of variables.
data flow coverage: The percentage of definition-use pairs that have been exercised by a test suite.
data flow testing: A white box test design technique in which test cases are designed to execute definition-use pairs of variables.
data integrity testing: See *database integrity testing*.
data quality; An attribute of data that indicates correctness with respect to some pre-defined criteria, e.g., business expectations, requirements on data integrity, data consistency.

database integrity testing: Testing the methods and processes used to access and manage the data(base), to ensure access methods, processes and data rules function as expected and that during access to the database, data is not corrupted or unexpectedly deleted, updated or created.
dd-path: A path between two decisions of an algorithm, or two decision nodes of a corresponding graph, that includes no other decisions. See also *path.*
dead code: See *unreachable code*.
debugger: See *debugging tool*.
debugging: The process of finding, analyzing and removing the causes of failures in software.
debugging tool: A tool used by programmers to reproduce failures, investigate the state of
programs and find the corresponding defect. Debuggers enable programmers to execute programs step by step, to halt a program at any program statement and to set and examine program variables.
decision: A program point at which the control flow has two or more alternative routes. A node with two or more links to separate branches.
decision condition coverage: The percentage of all condition outcomes and decision outcomes that have been exercised by a test suite. 100% decision condition coverage implies both 100% condition coverage and 100% decision coverage.
decision condition testing: A white box test design technique in which test cases are designed to execute condition outcomes and decision outcomes.
decision coverage: The percentage of decision outcomes that have been exercised by a test suite. 100% decision coverage implies both 100% branch coverage and 100% statement coverage.
decision outcome: The result of a decision (which therefore determines the branches to be taken).
decision table: A table showing combinations of inputs and/or stimuli (causes) with their associated outputs and/or actions (effects), which can be used to design test cases.
decision table testing: A black box test design technique in which test cases are designed to execute the combinations of inputs and/or stimuli (causes) shown in a decision table.
See also *decision table.*
decision testing: A white box test design technique in which test cases are designed to execute decision outcomes.
defect: A flaw in a component or system that can cause the component or system to fail to
perform its required function, e.g. an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system. defect-based technique: See *defect-based test design technique*.
defect-based test design technique: A procedure to derive and/or select test cases targeted at one or more defect types, with tests being developed from what is known about the specific defect type. See also *defect taxonomy.*
defect category: See *defect type.*
defect density: The number of defects identified in a component or system divided by the size of the component or system (expressed in standard measurement terms, e.g. lines-of-code, number of classes or function points).
Defect Detection Percentage (DDP): The number of defects found by a test level, divided by the number found by that test level and any other means afterwards. See also *escaped defects.*
defect management: The process of recognizing, investigating, taking action and disposing of defects. It involves recording defects, classifying them and identifying the impact.
defect management committee: A cross-functional team of stakeholders who manage reported defects from initial detection to ultimate resolution (defect removal, defect deferral, or report cancellation). In some cases, the same team as the configuration control board. *See also* configuration control board.
defect management tool: A tool that facilitates the recording and status tracking of defects and changes. They often have workflow-oriented facilities to track and control the allocation, correction and re-testing of defects and provide reporting facilities. See also *incident management tool*.

defect masking: An occurrence in which one defect prevents the detection of another.
defect report: A document reporting on any flaw in a component or system that can cause the component or system to fail to perform its required function.
defect taxonomy: A system of (hierarchical) categories designed to be a useful aid for reproducibly classifying defects.
defect tracking tool: See *defect management tool*.
defect triage committee: *See* defect management committee.
defect type: An element in a taxonomy of defects. Defect taxonomies can be identified with respect to a variety of considerations, including, but not limited to:
- Phase or development activity in which the defect is created, e.g., a specification error or a
coding error
- Characterization of defects, e.g., an "off-by-one" defect
- Incorrectness, e.g., an incorrect relational operator, a programming language syntax error, or
an invalid assumption
- Performance issues, e.g., excessive execution time, insufficient availability.
definition-use pair: The association of a definition of a variable with the subsequent use of that variable. Variable uses include computational (e.g. multiplication) or to direct the execution of a path ("predicate" use).
deliverable: Any (work) product that must be delivered to someone other than the (work) product's author.
Demo (Demonstration): At the end of each iteration, the development unit performs a demo of the functionality completed during the iteration. The demo is a forum for the customer to provide feedback on the product's development to influence the evolution of the product.
Deming cycle: An iterative four-step problem-solving process, (plan-do-check-act), typically used in process improvement.
design-based testing: An approach to testing in which test cases are designed based on the
architecture and/or detailed design of a component or system (e.g. tests of interfaces between components or systems).
desk checking: Testing of software or a specification by manual simulation of its execution. See also *static testing.*
Developer Unit: The Developer Unit refers to the people that are responsible for delivering working software that meets requirements by collaborating with the customer throughout the development lifecycle. Typically, the Development Unit is comprised of individuals fulfilling the technical roles of development (developers, QA, tech writer, project manager, DBA and others), working together in one or more, cross-functional agile teams. In agile projects, the developer unit is responsible for estimating the backlog, working with the customer unit to plan the iterations, iteration execution, demonstration and ultimate delivery of working software.
development testing: Formal or informal testing conducted during the implementation of a component or system, usually in the development environment by developers.
deviation: See *incident*.
deviation report: See *incident report*.
diagnosing (IDEAL): The phase within the IDEAL model where it is determined where one is, relative to where one wants to be. The diagnosing phase consists of the activities: characterize current and desired states and develop recommendations. See also *IDEAL*.
dirty testing: See *negative testing*.
documentation testing: Testing the quality of the documentation, e.g. user guide or installation guide.
domain: The set from which valid input and/or output values can be selected.
domain analysis: A black box test design technique that is used to identify efficient and effective test cases when multiple variables can or should be tested together. It builds on and generalizes equivalence partitioning and boundary values analysis. *See also* boundary value analysis, equivalence partitioning.
driver: A software component or test tool that replaces a component that takes care of the
control and/or the calling of a component or system.
dynamic analysis: The process of evaluating behavior, e.g. memory performance, CPU usage, of a system or component during execution.

dynamic analysis tool: A tool that provides run-time information on the state of the software code. These tools are most commonly used to identify unassigned pointers, check pointer arithmetic and to monitor the allocation, use and de-allocation of memory and to flag memory leaks.
dynamic comparison: Comparison of actual and expected results, performed while the software is being executed, for example by a test execution tool.

Dynamic Systems Development Method: Often abbreviated as DSDM, it is another example of a lightweight, Agile software development methodology. DSDM is an iterative and incremental approach that is largely based on the Rapid Application Development (RAD) methodology. The method provides a 4 staged/phased framework consisting of:

1. **Feasibility & Business Study**
2. **Functional Model / Prototype Iteration**
3. **Design and Build Iteration**
4. **Implementation**

Within each of the different phases, DSDM relies on several different activities and techniques that are all based on the following key, underlying principles:

- Projects best evolve through direct and co-located collaboration between the developers and the users.
- Self-managed and empowered teams must have the authority to make time sensitive and critical project level decisions.
- Design and Development is incremental and evolutionary in nature and is largely driven by regular and iterative user feedback.
- Working software deliverables are defined as systems that address the critical and current business needs versus systems that address less critical and future needs.
- Frequent and incremental delivery of working software is valued over infrequent delivery of perfectly working software.
- All changes introduced during development must be reversible.
- Continuous integration and QA Testing is conducted in-line, throughout the project lifecycle.
- Visibility and transparency is encouraged through regular communication and collaboration amongst all project stakeholders.

dynamic testing: Testing that involves the execution of the software of a component or system**.**

E
effectiveness: The capability of producing an intended result. See also *efficiency*.
efficiency: (1) The capability of the software product to provide appropriate performance, relative to the amount of resources used under stated conditions.
(2) The capability of a process to produce the intended outcome, relative to the amount of resources used
efficiency testing: The process of testing to determine the efficiency of a software product.
EFQM (European Foundation for Quality Management) excellence model: A non-prescriptive framework for an organisation's quality management system, defined and owned by the European Foundation for Quality Management, based on five 'Enabling' criteria (covering what an organisation does), and four 'Results' criteria (covering what an organisation achieves).
elementary comparison testing: A black box test design technique in which test cases are designed to execute combinations of inputs using the concept of modified condition decision coverage
embedded iterative development model: A development lifecycle sub-model that applies an iterative approach to detailed design, coding and testing within an overall sequential model. In this case, the high level design documents are prepared and approved for the entire project but the actual detailed design, code development and testing are conducted in iterations.
emotional intelligence: The ability, capacity, and skill to identify, assess, and manage the emotions of one's self, of others, and of groups.
EMTE: Acronym for Equivalent Manual Test Effort.
emulator: A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system. See also *simulator*.

entry criteria: The set of generic and specific conditions for permitting a process to go forward with a defined task, e.g. test phase. The purpose of entry criteria is to prevent a task from starting which would entail more (wasted) effort compared to the effort needed to remove the failed entry criteria.
entry point: An executable statement or process step which defines a point at which a given process is intended to begin.
Epic Stories: Epic stories are user stories whose scope is so large as to make them difficult to complete in a single iteration or accurately estimate the level of effort to deliver. Epic stories, while common when first defining the product backlog (see product backlog), should be decomposed into smaller user stories where the requirements of the story are defined much more narrowly in scope.
equivalence class: See *equivalence partition*.
equivalence partition: A portion of an input or output domain for which the behavior of a component or system is assumed to be the same, based on the specification.
equivalence partition coverage: The percentage of equivalence partitions that have been exercised by a test suite.
equivalence partitioning: A black box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle test cases are designed to cover each partition at least once.
equivalent manual test effort: Effort required for running tests manually.
error: A human action that produces an incorrect result.
error guessing: A test design technique where the experience of the tester is used to anticipate what defects might be present in the component or system under test as a result of errors made, and to design tests specifically to expose them.
error seeding: See *fault seeding.*
error seeding tool: See *fault seeding tool.*
error tolerance: The ability of a system or component to continue normal operation despite the presence of erroneous inputs.
escaped defect: A defect that was not detected in a previous test level which is supposed to find such type of defects. See also *Defect Detection Percentage.*
establishing (IDEAL): The phase within the IDEAL model where the specifics of how an organization will reach its destination are planned. The establishing phase consists of the activities: set priorities, develop approach and plan actions. See also *IDEAL*.
evaluation: See *testing*.
exception handling: Behavior of a component or system in response to erroneous input, from either a human user or from another component or system, or to an internal failure.
executable statement: A statement which, when compiled, is translated into object code, and which will be executed procedurally when the program is running and may perform an action on data.
exercised: A program element is said to be exercised by a test case when the input value causes the execution of that element, such as a statement, decision, or other structural element.
exhaustive testing: A test approach in which the test suite comprises all combinations of input values and preconditions.
exit criteria: The set of generic and specific conditions, agreed upon with the stakeholders
for permitting a process to be officially completed. The purpose of exit criteria
is to prevent a task from being considered completed when there are still outstanding parts of the task which have not been finished. Exit criteria are used to report against and to plan when to stop testing.
exit point: An executable statement or process step which defines a point at which a given process is intended to cease..
expected outcome: See *expected result*.
expected result: The behavior predicted by the specification, or another source, of the component or system under specified conditions.
experience-based technique: See *experience-based test design technique.*
experience-based test design technique: Procedure to derive and/or select test cases based on the tester's experience, knowledge and intuition.
experience-based testing: Testing based on the tester's experience, knowledge and intuition.
exploratory testing: An informal test design technique where the tester actively controls the

design of the tests as those tests are performed and uses information gained while testing to design new and better tests.
extreme programming (XP): A software engineering methodology used within agile software development whereby core practices are programming in pairs, doing extensive code review, unit testing of all code, and simplicity and clarity in code. See also *agile software development.*

Often abbreviated as XP, it is a popular example of a lightweight, Agile software development method. XP seeks to improve software quality by focusing on technical excellence, while improving project agility and responsiveness to changing requirements by valuing small yet frequent, time-boxed releases. XP provides a basic framework for managing projects based on the following key values:

- **Communication:** The most effective way to communicate requirements is by direct communication between the user and the developer
- **Simplicity:** Focus on building the simplest solution that meets the needs of today.
- **Feedback:** Inspect, adapt and evolve the system by responding to feedback from system tests, user acceptance tests and team input.
- **Courage:** By having the courage to refactor in the future, we can focus on only building what we need today.
- **Respect:** Do no harm to others by striving for the highest degree of quality in the solutions you build today.

F
factory acceptance testing: Acceptance testing conducted at the site at which the product is
developed and performed by employees of the supplier organization, to determine whether or not a component or system satisfies the requirements, normally including hardware as well as software. See also *alpha testing.*
fail: A test is deemed to fail if its actual result does not match its expected result.
failover testing: Testing by simulating failure modes or actually causing failures in a controlled environment. Following a failure, the failover mechanism is tested to ensure that data is not lost or corrupted and that any agreed service levels are maintained (e.g., function availability or response times). See also *recoverability testing*.
failure: Deviation of the component or system from its expected delivery, service or result.
failure mode: The physical or functional manifestation of a failure. For example, a system in failure mode may be characterized by slow operation, incorrect outputs, or complete termination of execution.
Failure Mode and Effect Analysis (FMEA): A systematic approach to risk identification and analysis of identifying possible modes of failure and attempting to prevent their occurrence. See also *Failure Mode, Effect and Criticality Analysis (FMECA).*
Failure Mode, Effects, and Criticality Analysis (FMECA): An extension of FMEA, as in addition to the basic FMEA, it includes a criticality analysis, which is used to chart the probability of failure modes against the severity of their consequences. The result highlights failure modes with relatively high probability and severity of consequences, allowing remedial effort to be directed where it will produce the greatest value. See also *Failure Mode and Effect Analysis (FMEA).*
failure rate: The ratio of the number of failures of a given category to a given unit of measure, e.g. failures per unit of time, failures per number of transactions, failures per number of computer runs.
false-fail result: A test result in which a defect is reported although no such defect actually exists in the test object.
false-negative result: See *false-pass result*.
false-pass result: A test result which fails to identify the presence of a defect that is actually present in the test object.
false-positive result: See *false-fail result*.
fault: See *defect*.
fault attack: See *attack.*
fault density: See *defect density*.
Fault Detection Percentage (FDP): See *Defect Detection Percentage (DDP).*
fault injection: The process of intentionally adding defects to a system for the purpose of finding out whether the system can detect, and possibly recover from, a defect. Fault injection intended to mimic failures that might occur in the field. See also *fault tolerance.*
fault masking: See *defect masking*.

fault seeding: The process of intentionally adding defects to those already in the component or system for the purpose of monitoring the rate of detection and removal, and estimating the number of remaining defects. Fault seeding is typically part of development (pre-release) testing and can be performed at any test level (component, integration, or system).
fault seeding tool: A tool for seeding (i.e. intentionally inserting) faults in a component or system.
fault tolerance: The capability of the software product to maintain a specified level of performance in cases of software faults (defects) or of infringement of its specified interface. See also *reliability, robustness*.
Fault Tree Analysis (FTA): A technique used to analyze the causes of faults (defects). The technique visually models how logical relationships between failures, human errors, and external events can combine to cause specific faults to disclose.
feasible path: A path for which a set of input values and preconditions exists which causes it to be executed.
feature: An attribute of a component or system specified or implied by requirements documentation (for example reliability, usability or design constraints).
Feature Based Planning: Feature based planning is an approach used in release planning, where features and scope take priority over date. Release plans (see Release Planning) created utilizing this approach are created by estimating the amount of time that will be required to complete a certain defined amount of scope. Often this approach is utilized for new product launches where a minimal or critical amount of feature/function must be delivered in order for the completed product to be considered market worthy.
Feature-driven development: An iterative and incremental software development process driven from a client-valued functionality (feature) perspective. Feature-driven development is mostly used in agile software development. See also *agile software development.*

Feature driven development, or FDD, is an example of a lightweight, Agile approach to development. Like many other Agile methods, FDD seeks to deliver valuable, working software frequently in an iterative manner. FDD utilizes an incremental, model driven approach that is based on the following 5 key activities:

1. **Develop the Overall Model:** Define the high level technical and functional architecture and project vision.
2. **Build the Feature List:** Define the individual features that are required to meet the needs of the defined model.
3. **Plan by Feature:** Create the actual working development plans by grouping the defined requirements into feature groups or themes.
4. **Design by Feature:** Based on the feature based plans, design packages are created that seek to group together small feature sets that can be developed within 2 week, iterative periods.
5. **Develop by Feature:** Development feature teams, execute against the design packages and promote completed and tested code to the main build branch within the 2 week, time-boxed iteration.

Feature Teams: Feature teams are small, cross-functional teams of development resources, focused on designing and building specific feature groupings or areas of system functionality. The feature team concept is utilized by FDD and other agile methods.

Fibonacci Sequence: Discovered in the 12th century by Leonardo Pisano, the Fibonacci sequence is a mathematically recursive sequence, in which the result of each subsequent term is determined by the sum of the two previous terms. A classic example of this concept is illustrated in the following string of numbers:

1, 1, 2, 3, 5, 8, 13, 21…

Using this example above, 1+1=2, 1+2=3, 2+3=5 and so on. The Fibonacci sequence serves as the basis of popular agile estimating technique known as Planning Poker.
field testing: See *beta testing*.
finite state machine: A computational model consisting of a finite number of states and transitions between those states, possibly with accompanying actions.
finite state testing: See *state transition testing*.
fishbone diagram: See *cause-effect diagram.*
Five Levels of Agile Planning: The five levels of Agile planning are Vision, Roadmap, Release, Iteration (or Sprint), and Daily. The top level (Vision) represents the "big picture" of the overall effort and thus the planning at this level encompasses more strategic

product information and fewer details on the product specifics. Working through to the bottom level, more details are included in the produced plans, so that in whole, the five levels of Agile planning represents a holistic understanding of what we are building, why we are undertaking the effort, and how we plan to deliver.
formal review: A review characterized by documented procedures and requirements, e.g. inspection.
frozen test basis: A test basis document that can only be amended by a formal change control process. See also *baseline*.
Function Point Analysis (FPA): Method aiming to measure the size of the functionality of an information system. The measurement is independent of the technology. This measurement may be used as a basis for the measurement of productivity, the estimation of the needed resources, and project control.
functional integration: An integration approach that combines the components or systems for the purpose of getting a basic functionality working early. See also *integration testing*.
Functional requirement: A requirement that specifies a function that a component or system must perform.
functional test design technique: Procedure to derive and/or select test cases based on an analysis of the specification of the functionality of a component or system without reference to its internal structure. See also *black box test design technique*.
functional testing: Testing based on an analysis of the specification of the functionality of a component or system. See also *black box testing*.
functionality: The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.
functionality testing: The process of testing to determine the functionality of a software product.

G
generic test automation architecture: Representation of the layers, components, and interfaces of a test automation architecture, allowing for a structured and modular approach to implement test automation.
glass box testing: See *white box testing*.
Goal Question Metric: An approach to software measurement using a three-level model: conceptual level (goal), operational level (question) and quantitative level (metric).
GQM: See Goal Question Metric.
GUI: Acronym for Graphical User Interface.
GUI testing: Testing performed by interacting with the software under test via the graphical user interface.

H
hardware-software integration testing: Testing performed to expose defects in the interfaces and interaction between hardware and software components. See also *integration testing*.
hazard analysis: A technique used to characterize the elements of risk. The result of a hazard analysis will drive the methods used for development and testing of a system. See also *risk analysis.*
heuristic evaluation: A usability review technique that targets usability problems in the user interface or user interface design. With this technique, the reviewers examine the interface and judge its compliance with recognized usability principles (the "heuristics").
high level test case: A test case without concrete (implementation level) values for input data and expected results. Logical operators are used; instances of the actual values are not yet defined and/or available. See also *low level test case*.
horizontal traceability: The tracing of requirements for a test level through the layers of test
documentation (e.g. test plan, test design specification, test case specification and test procedure specification or test script).
hyperlink: A pointer within a web page that leads to other web pages.
hyperlink test tool: A tool used to check that no broken hyperlinks are present on a web site.

I

IDEAL: An organizational improvement model that serves as a roadmap for initiating, planning, and implementing improvement actions. The IDEAL model is named for the five phases it describes: initiating, diagnosing, establishing, acting, and learning.
Ideal Hours: Ideal hours is a concept often used when applying time-based estimates to development work items. Ideal time is the time it would take to complete a given task assuming zero interruptions or unplanned problems. Many time-based estimation methods utilize this time scale when planning and estimating. Considering the grossly optimistic assumptions this approach takes, the accuracy of ideal estimates are often inversely proportional to the duration of the estimate.
impact analysis: The assessment of change to the layers of development documentation, test documentation and components, in order to implement a given change to specified requirements.
incident: Any event occurring that requires investigation.
incident logging: Recording the details of any incident that occurred, e.g. during testing.
incident management: The process of recognizing, investigating, taking action and disposing of incidents. It involves logging incidents, classifying them and identifying the impact.
incident management tool: A tool that facilitates the recording and status tracking of incidents. They often have workflow-oriented facilities to track and control the allocation, correction and re-testing of incidents and provide reporting facilities. See also *defect management tool.*
incident report: A document reporting on any event that occurred, e.g. during the testing, which requires investigation.
incremental development model: A development lifecycle where a project is broken into a series of increments, each of which delivers a portion of the functionality in the overall project requirements. The requirements are prioritized and delivered in priority order in the appropriate increment. In some (but not all) versions of this lifecycle model, each subproject follows a 'mini Vmodel' with its own design, coding and testing phases.
incremental testing: Testing where components or systems are integrated and tested one or some at a time, until all the components or systems are integrated and tested.
independence of testing: Separation of responsibilities, which encourages the accomplishment of objective testing.
indicator: A measure that can be used to estimate or predict another measure.
infeasible path: A path that cannot be exercised by any set of possible input values.
informal review: A review not based on a formal (documented) procedure.
initiating (IDEAL): The phase within the IDEAL model where the groundwork is laid for a successful improvement effort. The initiating phase consists of the activities: set context, build sponsorship and charter infrastructure. See also *IDEAL*.
input: A variable (whether stored within a component or outside) that is read by a component.
input domain: The set from which valid input values can be selected. See also *domain*.
input value: An instance of an input. See also *input*.
insourced testing: Testing performed by people who are co-located with the project team but are not fellow employees.
inspection: A type of peer review that relies on visual examination of documents to detect ATM defects, e.g. violations of development standards and non-conformance to higher level documentation. The most formal review technique and therefore always based on a documented procedure. See also *peer review*.
inspection leader: See *moderator*.
inspector: See *reviewer*.
installability: The capability of the software product to be installed in a specified environment. See also *portability*.
installability testing: The process of testing the installability of a software product. See also *portability testing*.
installation guide: Supplied instructions on any suitable media, which guides the installer through the installation process. This may be a manual guide, step-by-step procedure, installation wizard, or any other similar process description.
installation wizard: Supplied software on any suitable media, which leads the installer through the installation process. It normally runs the installation process, provides feedback on installation results, and prompts for options.
instrumentation: The insertion of additional code into the program in order to collect information about program behavior during execution, e.g. for measuring code coverage.
instrumenter: A software tool used to carry out instrumentation.

intake test: A special instance of a smoke test to decide if the component or system is ready for detailed and further testing. An intake test is typically carried out at the start of the test execution phase. See also *smoke test.*
integration: The process of combining components or systems into larger assemblies.
integration testing: Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems. See also *component integration testing, system integration testing*.
integration testing in the large: See *system integration testing*.
integration testing in the small: See *component integration testing*.
interface testing: An integration test type that is concerned with testing the interfaces between components or systems.
interoperability: The capability of the software product to interact with one or more specified components or systems. See also *functionality*.
interoperability testing: The process of testing to determine the interoperability of a software product. See also *functionality testing*.
invalid testing: Testing using input values that should be rejected by the component or system. See also *error tolerance, negative testing*.

INVEST (acronym): Coined by Bill Wake in eXtreme Programing Explored, INVEST is an acronym that defines a simple set of rules used in creating well-formed User Stories.

- **Independent:** Stories should not be dependent on other stories.
- **Negotiable:** Too much explicit detail regarding particulars and solutions. Stories should capture the essence of the requirement and should not represent a contract on how to solve it.
- **Valuable:** Stories should clearly illustrate value to the customer.
- **Estimable:** Stories should provide just enough information so they can be estimated. It is not important to know the exact way that a particular problem will be solved, it must be understood enough to provide a high level estimate.
- **Small:** Stories should strive to be granular enough in scope that they may be completed in as little time as possible, from a few weeks to a few days.
- **Testable:** Stories need to be understood well enough so that a test can be defined for it. An effective way to ensure testability is to define user acceptance criteria for all user stories.

Iteration: Often also referred to as a Sprint, an iteration is a predefined, time-boxed and recurring period of time in which working software is created. The most commonly used iteration durations are 2, 4 and 6 week periods. The iteration level is also considered to be the fourth level in the five level Agile planning process.

> *Note: The terms Sprint and Iteration are synonyms and are effectively interchangeable. The term sprint is widely used by teams that identify their Agile approach as Scrum, whereas iteration is a more generic term used in the same manner.*

Iteration Backlog: Often also referred to as the Sprint Backlog, the iteration backlog is a subset of user stories from the product backlog that contains the planned scope of a specific iteration. Generally, the iteration backlog reflects the priority and order of the release plan and product roadmap.
Iteration Execution: Often also referred to as Sprint Execution, the recurring phase within the project lifecycle in which the Agile team executes against the iteration backlog. The goal of this phase is to complete all iteration commitments by delivering working software within the defined time constraints of the iteration. Typically, the iteration execution phase begins with the iteration kick-off and culminates with the iteration review.
Iteration Plan: Often also referred to as the Sprint Plan, the iteration plan is the detailed execution plan for a given (usually current) iteration. It defines the iteration goals and commitments by specifying the user stories, work tasks, priorities and team member work assignments required to complete the iteration. The iteration plan is normally produced by the entire development unit during the iteration planning session.
Iteration Review: Often also referred to as Sprint Review, the iteration review is an important communication forum that occurs at the end of an iteration. During the iteration review an Agile team will evaluate and agree on which stories have been completed and which stories need to be deferred or split. The iteration review is an event that generally signifies the closing of an iteration.
Ishikawa diagram: See *cause-effect diagram.*
isolation testing: Testing of individual components in isolation from surrounding components, with surrounding components being simulated by stubs and drivers, if needed.
item transmittal report: See *release note*.

iterative development model: A development lifecycle where a project is broken into a usually large number of iterations. An iteration is a complete development loop resulting in a release (internal or external) of an executable product, a subset of the final product under development, which grows from iteration to iteration to become the final product.

K

Kano Analysis: Developed by Professor Noriako Kano, it is a method used for classifying and categorizing requirements (user stories) based on their impact to customer satisfaction. The Kano Analysis model utilizes four categories into which each requirement can be classified. Those categories are:

- **Must Have/Must Be:** Baseline features, functional barriers to entry. Without these features customers won't use the product.
- **Satisfiers:** These are the features that a customer will generally expect and make the difference between a satisfying user experience versus one that is simply adequate. The more satisfiers the better.
- **Exciters and Delighters:** These are the features that 'delight' your customers, they love your product because of these features. Generally these are product differentiators.
- **Dissatisfiers:** These are features that customers do not want and should not be delivered. Dissatisfiers emerge as a backlog ages and better ideas are identified and delivered.

key performance indicator: See *performance indicator*.
keyword-driven testing: A scripting technique that uses data files to contain not only test data and expected results, but also keywords related to the application being tested. The keywords are interpreted by special supporting scripts that are called by the control script for the test. See also *data-driven testing*.

L

LCSAJ: A Linear Code Sequence And Jump, consists of the following three items (conventionally identified by line numbers in a source code listing): the start of the linear sequence of executable statements, the end of the linear sequence, and the target line to which control flow is transferred at the end of the linear sequence.
LCSAJ coverage: The percentage of LCSAJs of a component that have been exercised by a test suite. 100% LCSAJ coverage implies 100% decision coverage.
LCSAJ testing: A white box test design technique in which test cases are designed to execute LCSAJs.
lead assessor: The person who leads an assessment. In some cases, for instance CMMi and TMMi when formal assessments are conducted, the lead assessor must be accredited and formally trained.

Lean Software Development: Lean Software Development, which is rooted in the Lean Manufacturing techniques developed by Toyota, is another popular example of a lightweight Agile approach to product development. Much like other Agile methods, Lean attempts to address the shortcomings of traditional software project management methods by focusing on people and effective communication. Lean is further defined by the following seven key principles:

- **Eliminate Waste:** Understand your customers' needs and seek to deliver solutions that address only those needs as simply as possible.
- **Create Knowledge:** Create a team-based environment in which all individual participate in the design and problem-solving process. Create a culture that encourages constant improvement through regular inspection and adaptation.
- **Build Quality In:** Embrace re-factoring and test automation and test driven development. Understand your test requirements and plans before you begin coding.
- **Defer Commitment:** Avoid dependencies by embracing loose coupling. Maximize flexibility by narrowly defining requirements and schedule irreversible decisions to the last possible moment.
- **Optimize the Whole:** Focus on the entire value stream. Understand that a completed product is the sum of the all the various contributors and the result of effective collaboration.
- **Deliver Fast:** Reduce risk and volatility by limiting scope. Commit only to work for which you have the capacity to complete. Decompose work to discrete actionable tasks and release work often and iteratively.
- **Respect People:** Trust that your people know best how to do their jobs and empower them to make the decisions needed to complete their commitments.

learnability: The capability of the software product to enable the user to learn its application. See also *usability*.

learning (IDEAL): The phase within the IDEAL model where one learns from experiences and improves one's ability to adopt new processes and technologies in the future. The learning phase consists of the activities: analyze and validate, and propose future actions. See also *IDEAL*.
level of intrusion: The level to which a test object is modified by adjusting it for testability.
ATM **level test plan**: A test plan that typically addresses one test level. See also *test plan.*
lifecycle model: A partitioning of the life of a product or project into phases. See also *software lifecycle.*
linear scripting: A simple scripting technique without any control structure in the test scripts.
link testing: See *component integration testing*.
load profile: A specification of the activity which a component or system being tested may experience in production. A load profile consists of a designated number of virtual users who process a defined set of transactions in a specified time period and according to a predefined operational profile. See also *operational profile.*
load testing: A type of performance testing conducted to evaluate the behavior of a component or system with increasing load, e.g. numbers of parallel users and/or numbers of transactions, to determine what load can be handled by the component or system. See also *performance testing, stress testing*.
load testing tool: A tool to support load testing whereby it can simulate increasing load, e.g.,
numbers of concurrent users and/or transactions within a specified time-period. See also *performance testing tool.*
logic-coverage testing: See *white box testing*.
logic-driven testing: See *white box testing*.
logical test case: See *high level test case*.
low level test case: A test case with concrete (implementation level) values for input data and expected results. Logical operators from high level test cases are replaced by actual values that correspond to the objectives of the logical operators. See also *high level test case*.

M
maintainability: The ease with which a software product can be modified to correct defects, modified to meet new requirements, modified to make future maintenance easier, or adapted to a changed environment.
maintainability testing: The process of testing to determine the maintainability of a software product.
maintenance: Modification of a software product after delivery to correct defects, to improve
performance or other attributes, or to adapt the product to a modified environment.
maintenance testing: Testing the changes to an operational system or the impact of a changed environment to an operational system.
man in the middle attack: The interception, mimicking and/or altering and subsequent relaying of communications (e.g., credit card transactions) by a third party such that a user remains unaware of that third party's presence.
management review: A systematic evaluation of software acquisition, supply, development,
operation, or maintenance process, performed by or on behalf of management that monitors
progress, determines the status of plans and schedules, confirms requirements and their system allocation, or evaluates the effectiveness of management approaches to achieve fitness for purpose.
manufacturing-based quality: A view of quality, whereby quality is measured by the degree to which a product or service conforms to its intended design and requirements. Quality arises from the process(es) used. See also *product-based quality, transcendent-based quality, userbased quality, value-based quality.*
master test plan: A test plan that typically addresses multiple test levels. See also *test plan*.
maturity: (1) The capability of an organization with respect to the effectiveness and efficiency of its processes and work practices. See also *Capability Maturity Model Integration*, *Test Maturity Model integration*. (2) The capability of the software product to avoid failure as a result of defects in the software. See also *reliability*.

maturity level: Degree of process improvement across a predefined set of process areas in which all goals in the set are attained.
maturity model: A structured collection of elements that describe certain aspects of maturity in an organization, and aid in the definition and understanding of an organization's processes. A maturity model often provides a common language, shared vision and framework for prioritizing improvement actions.
MCDC: See *modified condition decision coverage*.
Mean Time Between Failures: The arithmetic mean (average) time between failures of a system. The MTBF is typically part of a reliability growth model that assumes the failed system is immediately repaired, as a part of a defect fixing process. See also *reliability growth model.*
Mean Time To Repair: The arithmetic mean (average) time a system will take to recover from any failure. This typically includes testing to insure that the defect has been resolved.
measure: The number or category assigned to an attribute of an entity by making a measurement.
measurement: The process of assigning a number or category to an entity to describe an attribute of that entity.
measurement scale: A scale that constrains the type of data analysis that can be performed on it.
Meta-Scrum: The Meta-Scrum is a communication forum that is often used in larger projects that scale across multiple Agile teams, for the purpose of coordinating resources and dependencies. Generally, this planning forum will involve the product owners, project managers and scrum masters.
memory leak: A memory access failure due to a defect in a program's dynamic store allocation logic that causes it to fail to release memory after it has finished using it, eventually causing the program and/or other concurrent processes to fail due to lack of memory.
methodical testing: Testing based on a standard set of tests, e.g., a checklist, a quality standard, or a set of generalized test cases.
metric: A measurement scale and the method used for measurement.
migration testing: See *conversion testing*.
milestone: A point in time in a project at which defined (intermediate) deliverables and results should be ready.
mind map: A diagram used to represent words, ideas, tasks, or other items linked to and arranged around a central keyword or idea. Mind maps are used to generate, visualize, structure, and classify ideas, and as an aid in study, organization, problem solving, decision making, and writing.
mistake: See *error*.
model-based testing: Testing based on a model of the component or system under test, e.g., reliability growth models, usage models such as operational profiles or behavioural models such as decision table or state transition diagram.
modelling tool: A tool that supports the creation, amendment and verification of models of the
software or system.
moderator: The leader and main person responsible for an inspection or other review process.
modified condition decision coverage: The percentage of all single condition outcomes that independently affect a decision outcome that have been exercised by a test case suite. 100% modified condition decision coverage implies 100% decision condition coverage.
modified condition decision testing: A white box test design technique in which test cases are designed to execute single condition outcomes that independently affect a decision outcome.
modified multiple condition coverage: See *modified condition decision coverage*.
modified multiple condition testing: See *modified condition decision testing*.
module: See *component*.
module testing: See *component testing*.
monitor: A software tool or hardware device that runs concurrently with the component or system under test and supervises, records and/or analyses the behavior of the component or system.
monitoring tool: See *monitor.*
monkey testing: Testing by means of a random selection from a large range of inputs and by randomly pushing buttons, ignorant of how the product is being used.

MoSCoW: MoSCoW is a feature classification/categorization method, rooted in rapid application development, that is commonly utilized in Agile projects. The method is intended for short, time-boxed development iterations where focus should remain on those items that are deemed most critical for delivery within the time-boxed period. MoSCoW itself is a modified acronym, which represents 4 different levels of priority classification.

- **Must Have:** These are time critical project requirements that must be delivered in order for the project not to be considered an outright failure. These are generally baseline, or critical path features.
- **Should Have:** These are also critical project level requirements, however they are not as time critical as Must Have requirements.
- **Could Have:** These are considered to be the Nice to Have requirements. Features that are not necessarily required for the success of the iteration or project, but features that would increase end-user/customer satisfaction in the completed product.
- **Won't Have:** These are lowest priority requirements that will not be scheduled or planned within the delivery time box.

MTBF: See *Mean Time Between Failures.*
MTTR: See *Mean Time To Repair.*
multiple condition: See *compound condition*.
multiple condition coverage: The percentage of combinations of all single condition outcomes within one statement that have been exercised by a test suite. 100% multiple condition coverage implies 100% modified condition decision coverage.
multiple condition testing: A white box test design technique in which test cases are designed to execute combinations of single condition outcomes (within one statement).
mutation analysis: A method to determine test suite thoroughness by measuring the extent to which a test suite can discriminate the program from slight variants (mutants) of the program.
mutation testing: See *back-to-back testing*.
Myers-Briggs Type Indicator (MBTI): An indicator of psychological preference representing the different personalities and communication styles of people.

N
N-switch coverage: The percentage of sequences of N+1 transitions that have been exercised by a test suite.
N-switch testing: A form of state transition testing in which test cases are designed to execute all valid sequences of N+1 transitions. See also *state transition testing*.
n-wise testing: A black box test design technique in which test cases are designed to execute all possible discrete combinations of any set of n input parameters. See also *combinatorial testing, orthogonal array testing, pairwise testing.*
negative testing: Tests aimed at showing that a component or system does not work. Negative testing is related to the testers' attitude rather than a specific test approach or test design technique, e.g. testing with invalid input values or exceptions.
neighborhood integration testing: A form of integration testing where all of the nodes that connect to a given node are the basis for the integration testing.
non-conformity: Non fulfillment of a specified requirement.
non-functional requirement: A requirement that does not relate to functionality, but to attributes such as reliability, efficiency, usability, maintainability and portability.
non-functional test design technique: Procedure to derive and/or select test cases for non-functional testing based on an analysis of the specification of a component or system without
reference to its internal structure. See also *black box test design technique*.
non-functional testing: Testing the attributes of a component or system that do not relate to
functionality, e.g. reliability, efficiency, usability, maintainability and portability.

O
off-the-shelf software: A software product that is developed for the general market, i.e. for a large number of customers, and that is delivered to many customers in identical format.
open source tool: A software tool that is available to all potential users in source code form, usually via the internet; its users are permitted, usually under license, to study, change, improve and, at times, to distribute the software.
operability: The capability of the software product to enable the user to operate and control it. See also *usability*.

operational acceptance testing: Operational testing in the acceptance test phase, typically performed in a (simulated) operational environment by operations and/or systems administration staff focusing on operational aspects, e.g. recoverability, resource-behavior, installability and technical compliance. See also *operational testing.*
operational environment: Hardware and software products installed at users' or customers' sites where the component or system under test will be used. The software may include operating systems, database management systems, and other applications.
operational profile: The representation of a distinct set of tasks performed by the component
or system, possibly based on user behavior when interacting with the component or system, and their probabilities of occurrence. A task is logical rather that physical and can be executed over several machines or be executed in non-contiguous time segments.
operational profile testing: Statistical testing using a model of system operations (short duration tasks) and their probability of typical use.
operational profiling: The process of developing and implementing an operational profile. See also *operational profile.*
operational testing: Testing conducted to evaluate a component or system in its operational
environment.
oracle: See *test oracle*.
orthogonal array: A 2-dimensional array constructed with special mathematical properties, such that choosing any two columns in the array provides every pair combination of each number in the array.
orthogonal array testing: A systematic way of testing all-pair combinations of variables using orthogonal arrays. It significantly reduces the number of all combinations of variables to test all pair combinations. See also *combinatorial testing, n-wise testing, pairwise testing.*
outcome: See *result*.
output: A variable (whether stored within a component or outside) that is written by a component.
output domain: The set from which valid output values can be selected. See also *domain*.
output value: An instance of an output. See also *output*.
outsourced testing: Testing performed by people who are not co-located with the project team and are not fellow employees.

P
pair programming: A software development approach whereby lines of code (production and/or test) of a component are written by two programmers sitting at a single computer. This implicitly means ongoing real-time code reviews are performed. A programming technique where two developers work together at a single workstation on the development of a single feature. Paired programming allows for better review of the code as it is created and also allows one developer to consider the strategic direction of the feature while the other produces the code. Paired programming often yields higher quality code and can allow for cross-training opportunities among team members of differing skill levels.
pair testing: Two persons, e.g. two testers, a developer and a tester, or an end-user and a tester, working together to find defects. Typically, they share one computer and trade control of it while testing.
pairwise integration testing: A form of integration testing that targets pairs of components that work together, as shown in a call graph.
pairwise testing: A black box test design technique in which test cases are designed to execute all possible discrete combinations of each pair of input parameters. See also *combinatorial testing, nwise testing*, *orthogonal array testing.*
Pareto analysis: A statistical technique in decision making that is used for selection of a limited number of factors that produce significant overall effect. In terms of quality improvement, a large majority of problems (80%) are produced by a few key causes (20%).
partition testing: See *equivalence partitioning*.
pass: A test is deemed to pass if its actual result matches its expected result.
pass/fail criteria: Decision rules used to determine whether a test item (function) or feature has passed or failed a test.
path: A sequence of events, e.g. executable statements, of a component or system from an entry point to an exit point.

path coverage: The percentage of paths that have been exercised by a test suite. 100% path
coverage implies 100% LCSAJ coverage.
path sensitizing: Choosing a set of input values to force the execution of a given path.
path testing: A white box test design technique in which test cases are designed to execute paths.
peer review: A review of a software work product by colleagues of the producer of the product for the purpose of identifying defects and improvements. Examples are inspection, technical review and walkthrough.
performance: The degree to which a system or component accomplishes its designated functions within given constraints regarding processing time and throughput rate. See also *efficiency*.
performance indicator: A high level metric of effectiveness and/or efficiency used to guide and control progressive development, e.g. lead-time slip for software development.
performance profiling: The task of analysing, e.g., identifying performance bottlenecks based on generated metrics, and tuning the performance of a software component or system using tools.
performance testing: The process of testing to determine the performance of a software product. See also *efficiency testing*.
performance testing tool: A tool to support performance testing that usually has two main facilities: load generation and test transaction measurement. Load generation can simulate either multiple users or high volumes of input data. During execution, response time measurements are taken from selected transactions and these are logged. Performance testing tools normally provide reports based on test logs and graphs of load against response times.
Persona: A fictional character that is created to represent the attributes of a group of the product's users. Personas are helpful tools to use as a guide when deciding on a product's features, functionality, or visual design. Personas allow a team to easily identify with a fictional version of the product's end users.
phase containment: The percentage of defects that are removed in the same phase of the software lifecycle in which they were introduced.
phase test plan: A test plan that typically addresses one test phase. See also *test plan.*
Planning Game: A planning meeting with the goal of selecting user stories for a release or iteration. The user stories selected for inclusion in the iteration or release should be is based on which user stories will deliver the highest value to the business given current development estimates.
planning poker: A consensus-based estimation technique, mostly used to estimate effort or relative size of user stories in agile software development. It is a variation of the Wide Band Delphi method using a deck of cards with values representing the units in which the team estimates. See also *agile software development, Wide Band Delphi.*
Planning poker is a team based exercise that is commonly used for assigning relative estimate values to user stories/requirements to express the effort required to deliver specific features or functionality. The game utilizes playing cards, printed with numbers based on a modified Fibonacci sequence (0, 1/2, 1, 2, 3, 5, 8, 13, 20, 40, 100). Equipped with playing cards, all members of the development unit team and the product owner meet together to discuss product backlog requirements for the purpose of reaching a consensus based estimate. The rules of the game are as follows:

- The team and the product owner select from the backlog a requirement that all agree is small enough in size and complexity to be considered a 2. This becomes the baseline requirement.
- With the baseline requirements selected, the product owner selects a requirement from the backlog and describes it greater detail, allowing the team to ask any clarifying questions. Discussion regarding the requirement should be limited in time to 5–8 minutes.
- Once everyone on the team is satisfied with the details, each player pulls a numbered card from their deck that they feel best represents the current requirements size and complexity relative to the baseline requirement or other known, estimated requirements. This card is placed face down.
- Once all the players have made their selection, all cards are turned face up and revealed to the group.
- Those players with high and/or low estimates are asked to justify their selection by discussing with the team.
- After the outliers have justified their selections, the team repeats the estimation process until group consensus achieved.
- This process is repeated as much as needed in order to score all requirements from the backlog.

pointer: A data item that specifies the location of another data item; for example, a data item that specifies the address of the next employee record to be processed.
portability: The ease with which the software product can be transferred from one hardware or software environment to another.
portability testing: The process of testing to determine the portability of a software product.
post condition: Environmental and state conditions that must be fulfilled after the execution of a test or test procedure.
post-execution comparison: Comparison of actual and expected results, performed after the software has finished running.
post-project meeting: See *retrospective meeting.*
precondition: Environmental and state conditions that must be fulfilled before the component or system can be executed with a particular test or test procedure.
predicate: A statement that can evaluate to true or false and may be used to determine the control flow of subsequent decision logic. *See also* decision.
predicted outcome: See *expected result*.
pretest: See *intake test*.
priority: The level of (business) importance assigned to an item, e.g. defect.
PRISMA (Product RISk MAnagement): A systematic approach to risk-based testing that employs product risk identification and analysis to create a product risk matrix based on likelihood and impact.
probe effect: The effect on the component or system by the measurement instrument when the component or system is being measured, e.g. by a performance testing tool or monitor. For example performance may be slightly worse when performance testing tools are being used.
problem: See *defect*.
problem management: See *defect management*.
problem report: See *defect report*.
procedure testing: Testing aimed at ensuring that the component or system can operate in conjunction with new or existing users' business procedures or operational procedures.
process: A set of interrelated activities, which transform inputs into outputs.
process assessment: A disciplined evaluation of an organization's software processes against a reference model.
process-compliant testing: Testing that follows a set of defined processes, e.g., defined by an external party such as a standards committee. See also *standard-compliant testing*.
process cycle test: A black box test design technique in which test cases are designed to execute business procedures and processes. See also *procedure testing.*
process-driven testing: A scripting technique where scripts are structured into scenarios which represent use cases of the software under test. The scripts can be parameterized with test data.
process improvement: A program of activities designed to improve the performance and maturity of the organization's processes, and the result of such a program.
process model: A framework wherein processes of the same nature are classified into a overall model, e.g. a test improvement model.
process reference model: A process model providing a generic body of best practices and how to improve a process in a prescribed step-by-step manner.
Product Backlog: The product backlog is a prioritized and estimated list of all outstanding product/project requirements, features, defects and other work items. The product backlog is typically owned and managed by the product owner who reviews it on a regular cadence to ensure that the development unit is focusing on the completion of those items that represent the highest impact on the overall product value.
product-based quality: A view of quality, wherein quality is based on a well-defined set of quality attributes. These attributes must be measured in an objective and quantitative way. Differences in the quality of products of the same type can be traced back to the way the specific quality attributes
have been implemented. See also *manufacturing-based quality, quality attribute, transcendent-based quality, user-based quality, value-based quality.*
Product Owner: Often referred to as the "Voice of Customer" on Agile projects or the "Product Manager" on traditional projects, the product owner is the person responsible for communicating the customer requirements. While considered part of the customer unit, the product owner role is critical to the success of an Agile product development effort. In addition to communicating requirements, the product owner is responsible for defining user acceptance criteria for requirements, prioritizing and managing the product backlog,

as well as collaborating with the development unit throughout the iterative development cycle.
product risk: A risk directly related to the test object. See also *risk*.
Product RISk MAnagement: See *PRISMA*.
production acceptance testing: See *operational acceptance testing.*
program instrumenter: See *instrumenter*.
program testing: See *component testing*.
project: A project is a unique set of coordinated and controlled activities with start and finish dates undertaken to achieve an objective conforming to specific requirements, including the constraints of time, cost and resources.
project retrospective: A structured way to capture lessons learned and to create specific action plans for improving on the next project or next project phase.
project risk: A risk related to management and control of the (test) project, e.g. lack of staffing, strict deadlines, changing requirements, etc. See also *risk*.
project test plan: See *master test plan*.
pseudo-random: A series which appears to be random but is in fact generated according to some prearranged sequence.

Q
QFD: See *quality function deployment.*
qualification: The process of demonstrating the ability to fulfil specified requirements. Note the term 'qualified' is used to designate the corresponding status.
quality: The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations.
quality assurance: Part of quality management focused on providing confidence that quality
requirements will be fulfilled.
quality attribute: A feature or characteristic that affects an item's quality.
quality characteristic: See *quality attribute*.
quality control: The operational techniques and activities, part of quality management, that are focused on fulfilling quality requirements.
quality function deployment: A method to transform user demands into design quality, to deploy the functions forming quality, and to deploy methods for achieving the design quality into subsystems and component parts, and ultimately to specific elements of the manufacturing process.
quality gate: A special milestone in a project. Quality gates are located between those phases of a project strongly depending on the outcome of a previous phase. A quality gate includes a formal check of the documents of the previous phase.
quality management: Coordinated activities to direct and control an organization with regard to quality. Direction and control with regard to quality generally includes the establishment of the quality policy and quality objectives, quality planning, quality control, quality assurance and quality improvement.
quality risk: A product risk related to a quality attribute. See also *quality attribute, product risk*.

R
Rapid Application Development: Rapid application development, often abbreviated simply as RAD, is a software development methodology that favors rapid and iterative prototyping in lieu of detailed and comprehensive plans, and is often considered an example of Agile development methods. RAD promotes a highly collaborative, team based approach to developing software, by evolving requirements through frequent and iterative delivery of working prototypes.
RACI matrix: A matrix describing the participation by various roles in completing tasks or deliverables for a project or process. It is especially useful in clarifying roles and responsibilities. RACI is an acronym derived from the four key responsibilities most typically used: Responsible, Accountable, Consulted, and Informed.
random testing: A black box test design technique where test cases are selected, possibly using a pseudo-random generation algorithm, to match an operational profile. This technique can be used for testing non-functional attributes such as reliability and performance.
Rational Unified Process: A proprietary adaptable iterative software development process framework consisting of four project lifecycle phases: inception, elaboration, construction and transition.

Created by Rational Software (which was later acquired by IBM), the Rational Unified Process (RUP) is an iterative development process that seeks to increase development agility by providing a flexible, best practice based life cycle management framework. RUP prescribes the utilization of 9 key disciplines extended across 4 main project phases. Those phases are:

1. **Inception Phase**
2. **Elaboration Phase**
3. **Construction Phase**
4. **Transition Phase**

The 9 key disciplines are as follows:

1. **Model:** The goal of this discipline is to understand the business of the organization, the problem domain being addressed by the project, and to identify a viable solution to address the problem domain.
2. **Requirements:** The goal of this discipline is to elicit stakeholder feature/function requirements in order to define the scope of the project.
3. **Analysis and Design:** The goal of this discipline is the define the requirements into actionable and executable designs and models.
4. **Implementation** The goal of this discipline is to transform your model(s) into executable code and to perform a basic level of testing, in particular unit testing.
5. **Test** The goal of this discipline is to perform an objective evaluation to ensure quality. This includes finding defects, validating that the system works as designed and verifying that the requirements are met.
6. **Deployment** The goal of this discipline is to plan for the delivery of the system and to execute the plan to make the system available to end users.
7. **Configuration Management** The goal of this discipline is to manage access to your project artifacts. This includes not only tracking artifact versions over time but also controlling and managing changes to them.
8. **Project Management** The goal of this discipline is to direct the activities that takes place on the project. This includes managing risks, directing people (assigning tasks, tracking progress, etc.) and coordinating with people and systems outside the scope of the project to be sure that it is delivered on time and within budget.
9. **Environment** The goal of this discipline is to support the rest of the effort by ensuring that the proper process, guidance (standards and guidelines), and tools (hardware, software, etc.) are available for the team as needed.

reactive testing: Testing that dynamically responds to the actual system under test and test results being obtained. Typically reactive testing has a reduced planning cycle and the design and implementation test phases are not carried out until the test object is received.
recorder: See *scribe*.
record/playback tool: See *capture/playback tool*.
recoverability: The capability of the software product to re-establish a specified level of performance and recover the data directly affected in case of failure. See also *reliability*.
recoverability testing: The process of testing to determine the recoverability of a software product. See also *reliability testing*.
recovery testing: See *recoverability testing*.
Refactoring: Refactoring refers to the process of modifying and revising development code in order to improve performance, efficiency, readability, or simplicity without affecting functionality. As Agile methods advocate simplicity in design, solving only the problems that exist today and technical excellence through continuous improvement, code refactoring is something that should be embraced by teams and made part of the normal development process. Refactoring is often seen by management as an investment in the value of longevity and adaptability of the product over time.
regression-averse testing: Testing using various techniques to manage the risk of regression, e.g., by designing re-usable testware and by extensive automation of testing at one or more test levels.
regression testing: Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a ETAE result of the changes made. It is performed when the software or its environment is changed. A test completed to validate previously completed and tested code. The regression test is performed in an effort to ensure that subsequent deliveries of

code segments have not corrupted previously completed code. These tests are also often performed after defects are remediated to ensure that the fixes have not corrupted any other portion of the software.
regulation testing: See *compliance testing.*
Relative Estimation: Relative estimation is a software estimation technique that attempts to size development requirements and work items not in terms of time or duration, but rather in terms of size and complexity relative to the size and complexity of other known requirements and work items. Relative estimation is commonly used in Agile development methods, and forms the basis of the planning poker estimation game.
release note: A document identifying test items, their configuration, current status and other delivery information delivered by development to testing, and possibly other stakeholders, at the start of a test execution phase.
Release Plan: A release plan is a document that further distills the roadmap by describing all of the anticipated activities, resources, and responsibilities related to a particular release, including the estimated duration of that release. Unlike in traditional waterfall managed projects, Agile methods seek to ensure the highest degree of plan accuracy by encouraging regular and iterative re-planning based on actual iteration results. Additionally, release planning is considered to be the third level in the five- level Agile planning process.
reliability: The ability of the software product to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations.
reliability growth model: A model that shows the growth in reliability over time during continuous testing of a component or system as a result of the removal of defects that result in reliability failures.
reliability testing: The process of testing to determine the reliability of a software product.
replaceability: The capability of the software product to be used in place of another specified software product for the same purpose in the same environment. See also *portability*.
requirement: A condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
requirements-based testing: An approach to testing in which test cases are designed based on test objectives and test conditions derived from requirements, e.g. tests that exercise specific functions or probe non-functional attributes such as reliability or usability.
requirements management tool: A tool that supports the recording of requirements, requirements attributes (e.g. priority, knowledge responsible) and annotation, and facilitates traceability through layers of requirements and requirements change management. Some requirements management tools also provide facilities for static analysis, such as consistency checking and violations to predefined requirements rules.
requirements phase: The period of time in the software lifecycle during which the requirements for a software product are defined and documented.
resource utilization: The capability of the software product to use appropriate amounts and types of resources, for example the amounts of main and secondary memory used by the program and the sizes of required temporary or overflow files, when the software performs its function under stated conditions. See also *efficiency*.
resource utilization testing: The process of testing to determine the resource-utilization of a software product. See also *efficiency testing*.
result: The consequence/outcome of the execution of a test. It includes outputs to screens, changes to data, reports, and communication messages sent out. See also *actual result, expected result.*
resumption criteria: The criteria used to restart all or a portion of the testing activities that were suspended previously.
resumption requirements: The defined set of testing activities that must be repeated when testing is re-started after a suspension.
re-testing: See *confirmation testing.*
Retrospective: By dictionary definition, retrospective refers to process of looking back on, and/or contemplating the past. In Agile methods, a retrospective is a communication forum in which Agile teams come together to celebrate team successes and to reflect on what can be improved. The goal of the meeting is to develop a plan that the team will use to apply lessons learned going forward. Unlike in traditionally managed projects where these meetings (often called "post-mortems" or "lessons learned" meetings) are typically held at the conclusion of a project, Agile methods advocate a more regular and iterative

approach, and encourage scheduling these meetings at the conclusion of each and every iteration. Retrospectives are an immensely powerful tool and are extremely useful in fostering an environment of continuous improvement.
retrospective meeting: A meeting at the end of a project during which the project team members evaluate the project and learn lessons that can be applied to the next project.
review: An evaluation of a product or project status to ascertain discrepancies from planned results and to recommend improvements. Examples include management review, informal review, technical review, inspection, and walkthrough.
review plan: A document describing the approach, resources and schedule of intended review activities. It identifies, amongst others: documents and code to be reviewed, review types to be used, participants, as well as entry and exit criteria to be applied in case of formal reviews, and the rationale for their choice. It is a record of the review planning process.
review tool: A tool that provides support to the review process. Typical features include review planning and tracking support, communication support, collaborative reviews and a repository for collecting and reporting of metrics.
reviewer: The person involved in the review that identifies and describes anomalies in the product or project under review. Reviewers can be chosen to represent different viewpoints and roles in the review process.
risk: A factor that could result in future negative consequences; usually expressed as impact and likelihood.
risk analysis: The process of assessing identified project or product risks to determine their level of risk, typically by estimating their impact and probability of occurrence (likelihood).
risk assessment: The process of identifying and subsequently analysing the identified project or product risk to determine its level of risk, typically by assigning likelihood and impact ratings. *See also* product risk, project risk, risk, risk impact, risk level, risk likelihood.
risk-based testing: An approach to testing to reduce the level of product risks and inform stakeholders of their status, starting in the initial stages of a project. It involves the identification of product risks and the use of risk levels to guide the test process.
risk category: See *risk type*.
risk control: The process through which decisions are reached and protective measures are
implemented for reducing risks to, or maintaining risks within, specified levels.
risk identification: The process of identifying risks using techniques such as brainstorming,
checklists and failure history. ATT
risk impact: The damage that will be caused if the risk become an actual outcome or event.
risk level: The importance of a risk as defined by its characteristics impact and likelihood. The level of risk can be used to determine the intensity of testing to be performed. A risk level can be expressed either qualitatively (e.g. high, medium, low) or quantitatively.
risk likelihood: The estimated probability that a risk will become an actual outcome or event.
risk management: Systematic application of procedures and practices to the tasks of identifying, analyzing, prioritizing, and controlling risk.
risk mitigation: See *risk control*.
risk type: A set of risks grouped by one or more common factors such as a quality attribute, cause, location, or potential effect of risk;. A specific set of product risk types is related to the type of testing that can mitigate (control) that risk type. For example the risk of user-interactions being misunderstood can be mitigated by usability testing.
Roadmap: The roadmap (or product roadmap) is a document that further distills the product vision as defined in the product charter into a high level plan. Commonly, the roadmap will attempt to outline project work that spans one or more releases, by grouping requirements into prioritized themes and estimating the execution schedule against said themes. Additionally, roadmap is considered to be the second level in the five-level Agile planning process.
robustness: The degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions. See also *error-tolerance, fault tolerance.*
robustness testing: Testing to determine the robustness of the software product.

root cause: A source of a defect such that if it is removed, the occurrence of the defect type is decreased or removed.
root cause analysis: An analysis technique aimed at identifying the root causes of defects. By directing corrective measures at root causes, it is hoped that the likelihood of defect recurrence will be minimized.
RUP: See *Rational Unified Process*.

S
safety: The capability of the software product to achieve acceptable levels of risk of harm to people, business, software, property or the environment in a specified context of use.
safety critical system: A system whose failure or malfunction may result in death or serious injury to people, or loss or severe damage to equipment, or environmental harm.
safety testing: Testing to determine the safety of a software product.
sanity test: See *smoke test*.
scalability: The capability of the software product to be upgraded to accommodate increased loads.
scalability testing: Testing to determine the scalability of the software product.
scenario testing: See *use case testing*.
Schedule Based Planning: Schedule based planning is an approach used in release planning, where date and schedule take priority over features and scope. Release plans created utilizing the approach are created by estimating the amount of scope that can be completed within defined release time box.
scorecard: A representation of summarized performance measurements representing progress towards the implementation of long-term goals. A scorecard provides static measurements of performance over or at the end of a defined interval. See also *balanced scorecard, dashboard.*
scribe: The person who records each defect mentioned and any suggestions for process improvement during a review meeting, on a logging form. The scribe should ensure that the logging form is readable and understandable.
scripted testing: Test execution carried out by following a previously documented sequence of tests.
scripting language: A programming language in which executable test scripts are written, used by a test execution tool (e.g. a capture/playback tool).
SCRUM: Scrum is a incremental and iterative software development framework, and is arguably one of the most commonly used Agile methods. Unlike other Agile methods, Scrum is not an acronym. The method was coined as such by Hirotaka Takeuchi and Ikujiro Nonaka in a HBR article titled 'The New Product Development Game' written in 1986, in reference to the sport of rugby. The process involved was developed by Ken Schwaber and Dr. Jeff Sutherland.
Scrum outlines a process framework in which Product Owners, Scrum Masters and Team Members, all work together collaboratively to define product and sprint backlogs that are executed in short, time-boxed iterations that are called sprints. At the end of each sprint, a working increment of the software is delivered/demonstrated to the product owner and the entire process repeats itself.
ScrumMaster: A key role in the Scrum product development framework, the ScrumMaster is the person who is primarily responsible for facilitating all Scrum meetings, removing team impediments, protecting teams from external distractions, keeping the team honest and on-track to ensure that the team is best able to deliver against the sprint goals. As Scrum teams are self-organizing and self-managed, it is important to differentiate between a ScrumMaster and a traditional manager. Rather than attempt to manage the scrum team, effective ScrumMasters work for the team, and are often best described as servant-leaders. Although many ScrumMasters were once traditional project managers, ScrumMasters focus on achieving the best performance from the product team and holding the team accountable to their commitments.

Scrum of Scrums: Similar in intent to the Daily Scrum (or Daily Stand Up), the Scrum of Scrums is a daily communication forum commonly used in larger projects utilizing multiple scrum teams. As more teams are introduced, the likelihood of intra-team impediments due to overlapping work and dependencies increases. The Scrum of Scrums is an effective way of managing these impediments. Typically, this meeting occurs after all of the individual team Scrum meetings have been completed. An individual from each Scrum team (usually the Scrum Master) is tasked with representing the Scrum team in this meeting.

The agenda of this meeting is virtually identical to that of the daily scrum, other than the three questions are modified slightly to better reflect the focus on teams.

1. What did my team accomplish yesterday?
2. What will my team commit to, or complete, today?
3. What impediments or obstacles are preventing my team from meeting its commitments?

Single Done Definition: Single Done Definition is a concept that attempts to address the phenomena where the number of different interpretations of the word "done" is equally proportional to the number of individuals and roles found in an Agile team. Understanding that progress is measured in terms of delivered, working software, and that team credit for iteration commitments is only realized when said commitments are fully completed, it becomes clear that a common team understanding of "completed" is required. By explicitly defining and documenting all of the various elements that must be completed before a work item can be considered "done", the likelihood of delivering working software is improved as teams ensure a consistent and common understanding of "done" as it pertains to team iteration commitments.
security: Attributes of software products that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs and data. See also *functionality.*
security testing: Testing to determine the security of the software product. See also *functionality testing.*
security testing tool: A tool that provides support for testing security characteristics and vulnerabilities.
security tool: A tool that supports operational security.
serviceability testing: See *maintainability testing*.
session-based test management: A method for measuring and managing session-based testing, e.g. exploratory testing.
session-based testing: An approach to testing in which test activities are planned as uninterrupted sessions of test design and execution, often used in conjunction with exploratory testing.
severity: The degree of impact that a defect has on the development or operation of a component or system.
Shewhart chart: See *control chart.*
short-circuiting: A programming language/interpreter technique for evaluating compound conditions in which a condition on one side of a logical operator may not be evaluated if the condition on the other side is sufficient to determine the final outcome.
simulation: The representation of selected behavioral characteristics of one physical or abstract system by another system.
simulator: A device, computer program or system used during testing, which behaves or operates like a given system when provided with a set of controlled inputs. See also *emulator*.
site acceptance testing: Acceptance testing by users/customers at their site, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes, normally including hardware as well as software.
S.M.A.R.T. goal methodology: A methodology whereby objectives are defined very specifically rather than generically. SMART is an acronym derived from the attributes of the objective to be defined: Specific, Measurable, Attainable, Relevant and Timely.
smoke test: A subset of all defined/planned test cases that cover the main functionality of a component or system, to ascertaining that the most crucial functions of a program work, but not bothering with finer details. See also *build, verification test, intake test*.
software: Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.
software attack: *See attack.*
Software Failure Mode and Effect Analysis (SFMEA): See *Failure Mode and Effect Analysis (FMEA)*.
Software Failure Mode, Effects, and Criticality Analysis (SFMECA): See *Failure Mode, Effects, and Criticality Analysis (FMECA)*.
Software Fault Tree Analysis (SFTA): See *Fault Tree Analysis (FTA).*
software feature: See *feature*.
software integrity level: The degree to which software complies or must comply with a set of stakeholder-selected software and/or software-based system characteristics (e.g., software complexity, risk assessment, safety level, security level, desired performance, reliability, or cost) which are defined to reflect the importance of the software to its stakeholders.
software lifecycle: The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software lifecycle typically

includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. Note these phases may overlap or be performed iteratively.
Software Process Improvement: A program of activities designed to improve the performance and maturity of the organization's software processes and the results of such a program.
software product characteristic: See *quality attribute.*
software quality: The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs. See also *quality*.
software quality characteristic: See *quality attribute*.
software test incident: See *incident*.
software test incident report: See *incident report*.
Software Usability Measurement Inventory (SUMI): A questionnaire-based usability test technique for measuring software quality from the end user's point of view.
source statement: See *statement*.
specification: A document that specifies, ideally in a complete, precise and verifiable manner, the requirements, design, behavior, or other characteristics of a component or system, and, often, the procedures for determining whether these provisions have been satisfied.
specification-based technique: See *black box test design technique.*
specification-based testing: See *black box testing*.
specification-based test design technique: See *black box test design technique*.
specified input: An input for which the specification predicts a result.
SPI: See *Software Process Improvement.*

Sprint: Often also referred to as an Iteration, a sprint is a predefined, time-boxed and recurring period of time in which working software is created. The most commonly used sprint durations are 2, 4 and 6 week periods. The sprint level is also considered to be the fourth level in the five-level Agile planning process.

> *Note: The terms* Sprint *and* Iteration *are synonyms and are effectively interchangeable. The term sprint is widely used by teams that identify their Agile approach as Scrum, whereas iteration is a more generic term used in the same manner.*

Sprint Backlog: Often also referred to as the Iteration Backlog, the sprint backlog is a subset of user stories from the product backlog that contains the planned scope of a specific iteration. Generally, the iteration backlog reflects the priority and order of the release plan and product roadmap.
Sprint Plan: Often also referred to as the Iteration Plan, the sprint plan is the detailed execution plan for a given (usually current) iteration. It defines the iteration goals and commitments by specifying the user stories, work tasks, priorities and team member work assignments required to complete the iteration. The Sprint Plan is normally produced by the entire development unit during the sprint planning session.
Sprint Review: Often also referred to as Iteration Review, the sprint review is an important communication forum that occurs at the end of a sprint. During the sprint review an Agile team will evaluate and agree on which stories have been completed and which stories need to be deferred or split. The sprint review is an event that generally signifies the closing of a sprint. Often times, teams will also use the sprint review as a forum to demonstrate the work that was completed within the sprint.
stability: The capability of the software product to avoid unexpected effects from modifications in the software. See also *maintainability*.
staged representation: A model structure wherein attaining the goals of a set of process areas establishes a maturity level; each level builds a foundation for subsequent levels.
standard: Formal, possibly mandatory, set of requirements developed and used to prescribe
consistent approaches to the way of working or to provide guidelines (e.g., ISO/IEC standards, IEEE standards, and organizational standards).
standard-compliant testing: Testing that complies to a set of requirements defined by a standard, e.g., an industry testing standard or a standard for testing safety-critical systems. See also *process-compliant testing*.
standard software: See *off-the-shelf software*.
standards testing: See *compliance testing*.
state diagram: A diagram that depicts the states that a component or system can assume, and shows the events or circumstances that cause and/or result from a change from one state to another.

state table: A grid showing the resulting transitions for each state combined with each possible event, showing both valid and invalid transitions.
state transition: A transition between two states of a component or system.
state transition testing: A black box test design technique in which test cases are designed to execute valid and invalid state transitions. See also *N-switch testing*.
statement: An entity in a programming language, which is typically the smallest indivisible unit of execution.
statement coverage: The percentage of executable statements that have been exercised by a test suite.
statement testing: A white box test design technique in which test cases are designed to execute statements.
static analysis: Analysis of software development artifacts, e.g. requirements or code, carried out without execution of these software development artifacts. Static analysis is usually carried out by means of a supporting tool.
static analysis tool: See *static analyzer*.
static analyzer: A tool that carries out static analysis.
static code analysis: Analysis of source code carried out without execution of that software.
static code analyzer: A tool that carries out static code analysis. The tool checks source code, for certain properties such as conformance to coding standards, quality metrics or data flow anomalies.
static testing: Testing of a software development artifact, e.g., requirements, design or code, without execution of these artifacts, e.g., reviews or static analysis.
statistical testing: A test design technique in which a model of the statistical distribution of the input is used to construct representative test cases. See also *operational profile testing*.
status accounting: An element of configuration management, consisting of the recording and reporting of information needed to manage a configuration effectively. This information includes a listing of the approved configuration identification, the status of proposed changes to the configuration, and the implementation status of the approved changes.
STEP: See *Systematic Test and Evaluation Process.*
storage: See *resource utilization.*
storage testing: See *resource utilization testing*.
Story Points: Story points are unit-less measures of relative size assigned to requirements for functionality. Story points are assigned by the entire team utilizing the planning poker exercise. Story points allow the team to focus on the pure size and complexity of delivering a specific piece of functionality rather than trying to perfectly estimate a duration of time required for the completion of the functionality.
Story Review: The story review forum, is a brief meeting that occurs prior to the start of new iteration and is attended by the product owner and development team. The intent of this meeting is to give teams a better understanding regarding upcoming stories/requirements so that they may be better prepared for the iteration planning meeting. Generally, story review meetings are between an hour or two in length and are scheduled about one week prior to the upcoming iteration planning/kick off meeting.
stress testing: A type of performance testing conducted to evaluate a system or component at or beyond the limits of its anticipated or specified workloads, or with reduced availability of resources such as access to memory or servers. See also *performance testing, load testing*.
stress testing tool: A tool that supports stress testing.
structural coverage: Coverage measures based on the internal structure of a component or system.
structural test design technique: See *white-box test design technique*.
structural testing: See *white-box testing*.
structure-based test design technique: See *white-box test design technique*.
structure-based technique: See *white-box test design technique.*
structure-based testing: See *white-box testing.*
structured scripting: A scripting technique that builds and utilizes a library of reusable (parts of) scripts.
structured walkthrough: See *walkthrough*.
stub: A skeletal or special-purpose implementation of a software component, used to develop or test a component that calls or is otherwise dependent on it. It replaces a called component.

subpath: A sequence of executable statements within a component.
suitability: The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives. See also *functionality*.
suitability testing: The process of testing to determine the suitability of a software product
SUMI: See *Software Usability Measurement Inventory*.
suspension criteria: The criteria used to (temporarily) stop all or a portion of the testing activities on the test items.
Sustainable Pace: The concept that developers should not work an excessive number of hours due to the possibility of "developer burnout." This approach reflects studies that have determined the team productivity greatly decreases when teams work in excess of 40 hours per week. Teams working at a sustainable pace are more effective in accurately predicting their capacity over time while also maintaining the quality of their deliverables.
SUT: Acronym for system under test.
syntax testing: A black box test design technique in which test cases are designed based upon the definition of the input domain and/or output domain.
system: A collection of components organized to accomplish a specific function or set of functions.
system integration testing: Testing the integration of systems and packages; testing interfaces to external organizations (e.g. Electronic Data Interchange, Internet).
system of systems: Multiple heterogeneous, distributed systems that are embedded in networks at multiple levels and in multiple interconnected domains, addressing large-scale inter-disciplinary common problems and purposes, usually without a common management structure.
system under test: See *test object*.
system testing: The process of testing an integrated system to verify that it meets specified requirements.
Systematic Test and Evaluation Process: A structured testing methodology, also used as a content-based model for improving the testing process. Systematic Test and Evaluation Process (STEP) does not require that improvements occur in a specific order. See also *content-based model.*

T
Task Boards: Task boards are visual communication and planning tools that are extremely useful for teams working in co-located environments. Typically, task boards take the form of a simple matrix, utilizing different work states (examples: not started, in progress, QA ready, completed) as column headers, and User Story Cards as the row headers. Within the matrix are the discrete tasks that describe the work required to complete a story. As the iteration progresses, tasks should move from one end of the task board to the other, through all of the various states. Due to their intuitive and simple nature, tasks boards provide a powerful means of measuring and communicating iteration health and progress.
TDD: *See* test-driven development.
Team Member: Simply put, a team member is anyone on a scrum team who is not a Product Owner or Scrum Master, who is responsible for specific iteration deliverables and is accountable for contributing to team iteration commitments and goals. A team member may be a developer, a technical writer, an architect, quality analyst or any other role essential to the production and delivery of working software.
technical review: A peer group discussion activity that focuses on achieving consensus on the technical approach to be taken. See also *peer review*.
test: A set of one or more test cases.
test adaption layer: The layer in a generic test automation architecture which provides the necessary code to adapt the automated tests for the various components, configuration or interfaces of the SUT.
test analysis: The process of analysing the test basis and defining test objectives.
test approach: The implementation of the test strategy for a specific project. It typically includes the decisions made that follow based on the (test) project's goal and the risk assessment carried out, starting points regarding the test process, the test design techniques to be applied, exit criteria and test types to be performed
test architect : (1) A person who provides guidance and strategic direction for a test organization and for its relationship with other disciplines. (2) A person who defines the way testing is structured for a given system, including topics such as
test tools and test data management.

test automation: The use of software to perform or support test activities, e.g. test management, test design, test execution and results checking. In other words, the practice of using software to automate the testing process. Testing automation requires up-front planning and configuration of the testing software to ensure that the execution of the test meets the expectations of the customer. Test automation allows for more frequent regression testing without increasing the resource requirements to execute the tests.
test automation architecture: An instantiation of the generic test automation architecture to define the architecture of a test automation solution, i.e., its layers, components, services and interfaces.
test automation engineer: A person who is responsible for the design, implementation and
maintenance of a test automation architecture as well as the technical evolution of the resulting test automation solution.
test automation framework: A tool that provides an environment for test automation. It usually includes a test harness and test libraries.
test automation manager: A person who is responsible for the planning and supervision of the development and evolution of a test automation solution.
test automation solution: A realization/implementation of a test automation architecture, i.e., a combination of components implementing a specific test automation assignment. The components may include off-the-shelf test tools, test automation frameworks, as well as test hardware.
test automation strategy: A high-level plan to achieve long-term objectives of test automation under given boundary conditions.
test basis: All documents from which the requirements of a component or system can be inferred; the documentation on which the test cases are based. If a document can be amended only by way of formal amendment procedure, then the test basis is called a frozen test basis.
test bed: See *test environment*.
test case: A set of input values, execution preconditions, expected results and execution post conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.
test case design technique: See *test design technique*.
test case result: The final verdict on the execution of a test and its outcomes, like pass, fail, or error. The result of error is used for situations where it is not clear whether the problem is in the test object.
test case specification: A document specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item. See also *test specification.*
test case suite: See *test suite*.
test charter: A statement of test objectives, and possibly test ideas about how to test. Test
charters are used in exploratory testing. See also *exploratory testing*.
test closure: During the test closure phase of a test process data is collected from completed
activities to consolidate experience, testware, facts and numbers. The test closure phase consists of finalizing and archiving the testware and evaluating the test process, including preparation of a test evaluation report. See also *test process.*
test comparator: A test tool to perform automated test comparison of actual results with expected results.
test comparison: The process of identifying differences between the actual results produced by the component or system under test and the expected results for a test. Test comparison can be performed during test execution (dynamic comparison) or after test execution.
test completion criteria: See *exit criteria*.
test condition: An item or event of a component or system that could be verified by one or
more test cases, e.g. a function, transaction, feature, quality attribute, or structural element.
test control: A test management task that deals with developing and applying a set of corrective actions to get a test project on track when monitoring shows a deviation from what was planned. See also *test management*.
test coverage: See *coverage*.

test cycle: Execution of the test process against a single identifiable release of the test object.
test data: Data that exists (for example, in a database) before a test is executed, and that affects or is affected by the component or system under test.
test data management: The process of analyzing test data requirements, designing test data structures, creating and maintaining test data.
test data preparation tool: A type of test tool that enables data to be selected from existing databases or created, generated, manipulated and edited for use in testing.
test definition layer: The layer in a generic test automation architecture which supports test implementation by supporting the definition of test suites and/or test cases, e.g., by offering templates or guidelines.
test deliverable: Any test (work) product that must be delivered to someone other than the test (work) product's author. See also *deliverable.*
test design: (1) See *test design specification*. (2) The process of transforming general test objectives into tangible test conditions and test cases.
test design specification: A document specifying the test conditions (coverage items) for a test item, the detailed test approach and identifying the associated high level test cases. See also *test specification.*
test design technique: Procedure used to derive and/or select test cases.
test design tool: A tool that supports the test design activity by generating test inputs from a specification that may be held in a CASE tool repository, e.g. requirements management tool, from specified test conditions held in the tool itself, or from code.
test director: A senior manager who manages test managers. *See also* test manager.
test-driven development: A way of developing software where the test cases are developed, and often automated, before the software is developed to run those test cases. More a technique than an actual development methodology, test driven development, often abbreviated as TDD, is a practice that is commonly utilized in Agile methods. TDD advocates the discipline of building working code, by first designing and building tests that exercise how the completed code should work, then creating the code so as to make all the pre-defined tests pass. The idea being that is if a developer first understands the tests required to validate how the code should work and when it is complete, that developer will be much more thoughtful and successful in designing and developing working, quality software. While TDD promotes the utilization of automated testing tools combined with version and source code control tools, automation should not be considered a strict requirement as there is still considerable value in simply following the basic, quality driven principles of TDD.
test driver: See *driver*.
test environment: An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test.
test estimation: The calculated approximation of a result related to various aspects of testing
(e.g. effort spent, completion date, costs involved, number of test cases, etc.) which is usable even if input data may be incomplete, uncertain, or noisy.
test evaluation report: A document produced at the end of the test process summarizing all testing activities and results. It also contains an evaluation of the test process and lessons learned.
test execution: The process of running a test on the component or system under test, producing actual result(s).
test execution automation: The use of software, e.g. capture/playback tools, to control the
execution of tests, the comparison of actual results to expected results, the setting up of test preconditions, and other test control and reporting functions.
test execution layer: The layer in a generic test automation architecture which supports the execution of test suites and/or test cases.
test execution phase: The period of time in a software development lifecycle during which the components of a software product are executed, and the software product is evaluated to
determine whether or not requirements have been satisfied.
test execution schedule: A scheme for the execution of test procedures. Note: The test procedures are included in the test execution schedule in their context and in the order in which they are to be executed.
test execution technique: The method used to perform the actual test execution, either manual or automated.

test execution tool: A type of test tool that is able to execute other software using an automated test script, e.g. capture/playback.
test fail: See *fail*.
test generation layer: The layer in a generic test automation architecture which supports manual or automated design of test suites and/or test cases.
test generator: See *test data preparation tool*.
test harness: A test environment comprised of stubs and drivers needed to execute a test.
test hook: A customized software interface that enables automated testing of a test object.
test implementation: The process of developing and prioritizing test procedures, creating test data and, optionally, preparing test harnesses and writing automated test scripts.
test improvement plan: A plan for achieving organizational test process improvement objectives based on a thorough understanding of the current strengths and weaknesses of the organization's test processes and test process assets.
test incident: See *incident*.
test incident report: See *incident report*.
test infrastructure: The organizational artifacts needed to perform testing, consisting of test environments, test tools, office environment and procedures.
test input: The data received from an external source by the test object during test execution. The external source can be hardware, software or human.
test item: The individual element to be tested. There usually is one test object and many test items. See also *test object*.
test item transmittal report: See *release note*.
test leader: See *test manager.*
test level: A group of test activities that are organized and managed together. A test level is linked to the responsibilities in a project. Examples of test levels are component test, integration test, system test and acceptance test.
test log: A chronological record of relevant details about the execution of tests.
test logging: The process of recording information about tests executed into a test log.
test management: The planning, estimating, monitoring and control of test activities, typically carried out by a test manager.
test management tool: A tool that provides support to the test management and control part of a test process. It often has several capabilities, such as testware management, scheduling of tests, the logging of results, progress tracking, incident management and test reporting.
test manager: The person responsible for project management of testing activities and resources, and evaluation of a test object. The individual who directs, controls, administers, plans and regulates the evaluation of a test object.
Test Maturity Model integration: A five level staged framework for test process improvement, related to the Capability Maturity Model Integration (CMMI), that describes the key elements of an effective test process.
test mission: The purpose of testing for an organization, often documented as part of the test policy. *See also* test policy.
test monitoring: A test management task that deals with the activities related to periodically checking the status of a test project. Reports are prepared that compare the actuals to that which was planned. See also *test management*.
test object: The component or system to be tested. See also *test item*.
test objective: A reason or purpose for designing and executing a test.
test oracle: A source to determine expected results to compare with the actual result of the software under test. An oracle may be the existing system (for a benchmark), other software, a user manual, or an individual's specialized knowledge, but should not be the code.
test outcome: See *result*.
test pass: See *pass*.
test performance indicator: A high level metric of effectiveness and/or efficiency used to guide and control progressive test development, e.g. Defect Detection Percentage (DDP).
test phase: A distinct set of test activities collected into a manageable phase of a project, e.g. the execution activities of a test level.
test plan: A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test

environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process.
test planning: The activity of establishing or updating a test plan.
Test Point Analysis (TPA): A formula based test estimation method based on function point analysis.
test policy: A high level document describing the principles, approach and major objectives the organization regarding testing.
test procedure: See *test procedure specification*.
test procedure specification: A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script. See also *test specification.*
test process: The fundamental test process comprises test planning and control, test analysis and design, test implementation and execution, evaluating exit criteria and reporting, and test closure activities.
Test Process Group: A collection of (test) specialists who facilitate the definition, maintenance, and improvement of the test processes used by an organization.
test process improvement: A program of activities designed to improve the performance and maturity of the organization's test processes and the results of such a program.
test process improvement manifesto: A statement that echoes the agile manifesto, and defines values for improving the testing process. The values are:

- flexibility over detailed processes
- best practices over templates
- deployment orientation over process orientation
- peer reviews over quality assurance (departments)
- business driven over model driven.

test process improver: A person implementing improvements in the test process based on a test improvement plan.
test progress report: A document summarizing testing activities and results, produced at regular intervals, to report progress of testing activities against a baseline (such as the original test plan) and to communicate risks and alternatives requiring a decision to management.
test record: See *test log*.
test recording: See *test logging*.
test report: See *test summary report* and *test progress report*.
test reporting: Collecting and analyzing data from testing activities and subsequently consolidating the data in a report to inform stakeholders. See also test process.
test reproducibility: An attribute of a test indicating whether the same results are produced each time the test is executed.
test requirement: See *test condition*.
test result: See *result*.
test rig: See *test environment.*
test run: Execution of a test on a specific version of the test object.
test run log: See *test log*.
test scenario: See *test procedure specification.*
test schedule: A list of activities, tasks or events of the test process, identifying their intended start and finish dates and/or times, and interdependencies.
test script: Commonly used to refer to a test procedure specification, especially an automated one.
test session: An uninterrupted period of time spent in executing tests. In exploratory testing, each test session is focused on a charter, but testers can also explore new opportunities or issues during a session. The tester creates and executes on the fly and records their progress. See also *exploratory testing*.
test set: See *test suite*.
test situation: See *test condition.*
test specification: A document that consists of a test design specification, test case specification and/or test procedure specification.
test specification technique: See *test design technique.*
test stage: See *test level*.
test strategy: A high-level description of the test levels to be performed and the testing within those levels for an organization or programme (one or more projects).

test suite: A set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one.
test summary report: A document summarizing testing activities and results. It also contains
an evaluation of the corresponding test items against exit criteria.
test target: A set of exit criteria.
test technique: See *test design technique.*
test tool: A software product that supports one or more test activities, such as planning and control, specification, building initial files and data, test execution and test analysis. See also *CAST*.
test type: A group of test activities aimed at testing a component or system focused on a specific test objective, i.e. functional test, usability test, regression test etc. A test type may take place on one or more test levels or test phases.
testability: The capability of the software product to enable modified software to be tested.
See also *maintainability*.
testability review: A detailed check of the test basis to determine whether the test basis is at an adequate quality level to act as an input document for the test process.
testable requirement: A requirements that is stated in terms that permit establishment of test designs (and subsequently test cases) and execution of tests to determine whether the requirement has been met.
tester: A skilled professional who is involved in the testing of a component or system.
testing: The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.
testware: Artifacts produced during the test process required to plan, design, and execute tests, such as documentation, scripts, inputs, expected results, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing.
thread testing: An approach to component integration testing where the progressive integration of components follows the implementation of subsets of the requirements, as opposed to the integration of components by levels of a hierarchy.
three point estimation: A test estimation method using estimated values for the "best case", "worst case", and "most likely case" of the matter being estimated, to define the degree of certainty associated with the resultant estimate.
time behavior: See *performance*.
TMMi: See *Test Maturity Model integration.*
top-down testing: An incremental approach to integration testing where the component at the top of the component hierarchy is tested first, with lower level components being simulated by stubs. Tested components are then used to test lower level components. The process is repeated until the lowest level components have been tested. See also *integration testing.*
Total Quality Management: An organization-wide management approach centered on quality, based on the participation of all members of the organization and aiming at long-term success through customer satisfaction, and benefits to all members of the organization and to society. Total Quality Management consists of planning, organizing, directing, control, and assurance.
TPG: See *Test Process Group.*
TPI Next: A continuous business-driven framework for test process improvement that describes the key elements of an effective and efficient test process.
TQM: See *Total Quality Management.*
traceability: The ability to identify related items in documentation and software, such as requirements with associated tests. See also *horizontal traceability*, *vertical traceability*.
traceability matrix: A two-dimensional table, which correlates two entities (e.g., requirements and test cases). The table allows tracing back and forth the links of one entity to the other, thus enabling the determination of coverage achieved and the assessment of impact of proposed changes.
transactional analysis: The analysis of transactions between people and within people's minds; a transaction is defined as a stimulus plus a response. Transactions take place between people and between the ego states (personality segments) within one person's mind.

transcendent-based quality: A view of quality, wherein quality cannot be precisely defined, but we know it when we see it, or are aware of its absence when it is missing. Quality depends on the perception and affective feelings of an individual or group of individuals towards a product. See also *manufacturing-based quality, product-based quality, user-based quality, value based quality.*

U

understandability: The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use. See also *usability*.

unit: See *component*.

Unit Test: A test performed by the developer to verify and validate the code that the developer completed is fit for use. The Unit Test is often the first level of testing that is completed as a part of a comprehensive test approach for software development.

unit test framework: A tool that provides an environment for unit or component testing in which a component can be tested in isolation or with suitable stubs and drivers. It also provides other support for the developer, such as debugging capabilities.

unit testing: See *component testing*.

unreachable code: Code that cannot be reached and therefore is impossible to execute.

usability: The capability of the software to be understood, learned, used and attractive to the user when used under specified conditions.

usability testing: Testing to determine the extent to which the software product is understood, easy to learn, easy to operate and attractive to the users under specified conditions.

use case: A sequence of transactions in a dialogue between an actor and a component or system with a tangible result, where an actor can be a user or anything that can exchange information with the system.

A use case is a document that attempts to describes system behavior from an end-user's perspective, by outlining the flow of data, system behavioral interchanges and corresponding end-user interactions in a sequential, step-by-step manner. While there is no single standard or format that they must follow, and use cases should vary in detail based on the needs of the requirements, uses cases often make use of the following artifacts to describe how a system should work.

- **Goal:** What is the end goal and desired effect of the functionality that the use case is attempting to describe.
- **Summary:** A brief, high-level and easily understood description of the use case.
- **Actors:** The consumers of the system that will be interacting with the system within the scope of the use cases. Actors can be people or other systems or services.
- **Preconditions:** System conditions and assumptions that must be true for the use case to be valid.
- **Triggers:** The specific events required to initiate the use case.
- **Body Text:** The description of each of the steps involved/required to complete the use case. Generally, body text will focus only the main (happy) path.
- **Alternative Path:** Steps that deviate from the main path due to exceptions, alternative logic or other conditional events.
- **Post Conditions:** The changes in the system and data as a result of executing steps outlined in the use case.

use case testing: A black box test design technique in which test cases are designed to execute scenarios of use cases.

user acceptance testing: User acceptance tests describe the tests that must be successfully executed in order to validate that specific piece of functionality meets the needs of the user as outlined in the customer requirements. As many Agile methods advocate the use of narrowly defined requirements that speak from a specific user's perspective (i.e. user stories), it is recommended that user acceptance criteria follow similar form and define validation steps from the same user perspective. User acceptance tests are an essential component of user requirements, as without well-defined acceptance criteria, it becomes difficult to clearly define the scope of any given requirement. See *acceptance testing*.

user-based quality: A view of quality, wherein quality is the capacity to satisfy needs, wants and desires of the user(s). A product or service that does not fulfill user needs is unlikely to find any users. This is a context dependent, contingent approach to quality since different business characteristics require different qualities of a product. See also *manufacturing-based quality, product-based quality, transcendent-based quality, value-based quality.*

User Roles: A key ingredient in defining Agile requirements, user roles are used to describe the unique perspectives of the different consumers that will interact with the working software. Much like actors in a use case, user roles should not just be limited to the human consumers of the software and should also include any other relevant external software or service.
user scenario testing: See *use case testing*
user story: A high-level user or business requirement commonly used in agile software development, typically consisting of one or more sentences in the everyday or business language capturing what functionality a user needs, any non-functional criteria, and also includes acceptance criteria.

User stories are simple, brief and concise statements, used to describe customer software requirements, from a particular user's perspective. User stories are often used in Agile methods for capturing and communicating customer requirements, as their short format allow for quick and easy requirements gathering, and their high-level structure encourages design evolution through team collaboration.

Commonly captured on 3x5 index cards so as to force brevity, user stories should seek to describe requirements as granularly as possible while still retaining customer value. User stories are not intended to describe complex system requirements on their own, but rather only narrow portions of requirements singularly, and through the whole, the full system requirements emerge.

A recommended format for users' stories is as follows:

*As a **(user role)**, I would like **(statement of need)**, so that I can **(desired benefit)**.*

Or for example

As an Agile student, I would like an Agile glossary, so that I can understand the meanings of words that I hear in training.

See also *agile software development, requirement*.
user story testing: A black box test design technique in which test cases are designed based on user stories to verify their correct implementation. See also *user story*.
user test: A test whereby real-life users are involved to evaluate the usability of a component or system.

V
V-model: A framework to describe the software development lifecycle activities from requirements specification to maintenance. The V-model illustrates how testing activities can be integrated into each phase of the software development lifecycle.
validation: Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled.
value-based quality: A view of quality, wherein quality is defined by price. A quality product or service is one that provides desired performance at an acceptable cost. Quality is determined by means of a decision process with stakeholders on trade-offs between time, effort and cost aspects. See also *manufacturing-based quality, product-based quality, transcendent-based quality, user-based quality.*
variable: An element of storage in a computer that is accessible by a software program by referring to it by a name.
Velocity: Used in combination with relative (story point) estimation, static teams, and fixed time-boxed iterations, velocity is a predictive metric that is useful for long/mid term planning and estimation. Simply put, velocity attempts to measure the number of story points a single team can complete within a single, time-boxed iteration. Assuming that all of the requirements in a product backlog have all been estimated using a similar relative estimation point scale, it becomes possible to estimate the number of time-boxed iterations required to complete said backlog by simply dividing the team velocity value by the sum of the backlog's complexity point total.
verification: Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled.
version control: See *configuration control.*
vertical traceability: The tracing of requirements through the layers of development documentation to components.
volume testing: Testing where the system is subjected to large volumes of data. See also *resource utilization testing*.

Vision: Vision is the first and highest level in the Agile planning process. Activities that are often associated with this level of planning are the creation of project charters, feasibility studies, funding decisions, definition of project success factors/metrics, and the formation of individual teams. Typically, vision planning is very strategic in nature and only occurs on an annual basis.

W

walkthrough: A step-by-step presentation by the author of a document in order to gather information and to establish a common understanding of its content. See also *peer review*.
WAMMI: See *Website Analysis and MeasureMent Inventory.*
WBS: See *Work Breakdown Structure.*
Website Analysis and MeasureMent Inventory (WAMMI): A questionnaire-based usability test technique for measuring web site software quality from the end user's point of view.
white-box technique: See *white-box test design technique.*
white-box test design technique: Procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system.
white-box testing: Testing based on an analysis of the internal structure of the component or system.
Wide Band Delphi: An expert based test estimation technique that aims at making an accurate estimation using the collective wisdom of the team members.
wild pointer: A pointer that references a location that is out of scope for that pointer or that does not exist. See also *pointer.*
Work Breakdown Structure: An arrangement of work elements and their relationship to each other and to the end product.

Working Software: The term used to describe the level of completeness that features developed during an iteration should achieve by the conclusion of the iteration. Working software implies that the features demonstrated to the customer at the end of an iteration should be functionally complete and will not require redevelopment in order to prepare the functionality for a production environment.

> *Used in one of the 12 Agile Principles*
>
> *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*

Additional reads:

Tilo Linz, "Testing in Scrum: A Guide for Software Quality Assurance in the Agile World", Rocky Nook, 2014.

Lisa Crispin and Janet Gregory, "Agile Testing: A Practical Guide for Testers and Agile Teams", Addison-Wesley Professional, 2008.

Dorothy Graham, Erik van Veenendaal, Isabel Evans and Rex Black: "FOUNDATIONS OF SOFTWARE TESTING".