**Notes for MCA-I (Semester- II**

**Course Name :- Software Project Management**

**(Course Code:- IT22)**

**Chapter 3]Agile Project Management Framework**

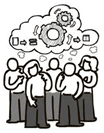
* **3.1 Agile :-**

Agile is a software development methodology to build a software incrementally using short iterations of 1 to 4 weeks so that the development is aligned with the changing business needs. An agile methodology is an iterative approach to software development. Each iteration of agile methodology takes a short time interval of 1 to 4 weeks. The agile development process is aligned to deliver the changing business requirement. It distributes the software with faster and fewer changes.

The single-phase software development takes 6 to 18 months. In single-phase development, all the requirement gathering and risks management factors are predicted initially.

The agile software development process frequently takes the feedback of workable product. The workable product is delivered within 1 to 4 weeks of iteration.

AGILE methodology is a practice that promotes **continuous iteration** of development and testing throughout the software development lifecycle of the project. In the Agile model, both development and testing activities are concurrent, unlike the Waterfall model.

[](https://www.guru99.com/images/11-2014/agile_Processesv1_1.png)

Agile Methodology

The **Agile software development** methodology is one of the simplest and effective processes to turn a vision for a business need into software solutions. Agile is a term used to describe software development approaches that employ continual planning, learning, improvement, team collaboration, evolutionary development, and early delivery. It encourages flexible responses to change.

**The agile software development emphasizes on four core values.**

1. Individual and team interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

**Agile in Brief :-**

1. Agile is an **iterative and incremental approach** to software development methodology.
2. In this approach, the **leadership** plays an important role.
3. Agile software development is highly suitable for the **medium or large project**.
4. **Flexibility** is the most significant advantage of agile as it quickly reacts to changes.
5. Agile involves **face-to-face communication** and collaboration between the members of various cross-functional teams.
6. Agile development needs **frequent delivery** to the end user for their feedback.
7. In this development, each step like requirements, analysis, design, are **continually monitored** during the lifecycle.
8. The **project leader** takes cares of all the tasks in the agile method.
9. End-user may give their **feedback during the development** process. So, the end product will be more useful.
10. **Delivery and update** of the software are taking place regularly.
11. Design and execution should be kept **simple**.
12. The priority of agile development is always to satisfy the customer by providing **continuous delivery** of valuable software.
13. Working software is the most **fundamental measure** of progress.
14. It is best to have **face-to-face communication** to get as close to the project goal as possible.

* **3.1.1 Agile Project Life Cycle :-**

Agile model believes that every project needs to be handled differently and the existing methods need to be tailored to best suit the project requirements. In Agile, the tasks are divided to time boxes (small time frames) to deliver specific features for a release.

Iterative approach is taken and working software build is delivered after each iteration. Each build is incremental in terms of features; the final build holds all the features required by the customer.

**Here is a graphical illustration of the Agile Model −**

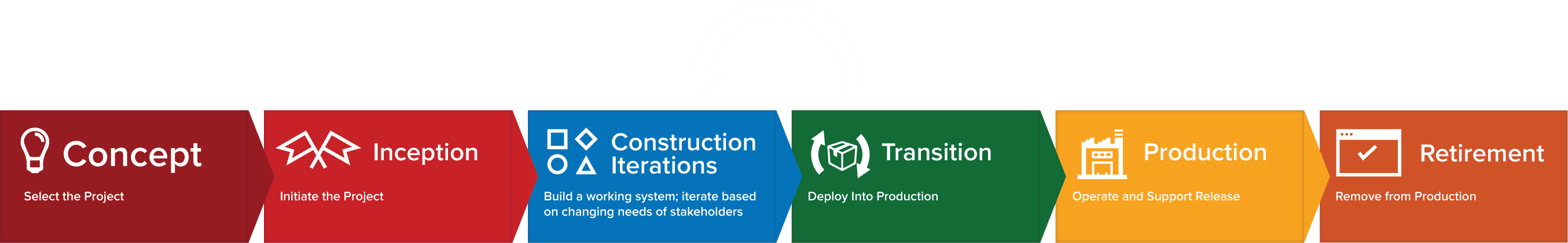
****

**The Agile Process Flow**

1. **Concept** - Projects are envisioned and prioritized
2. **Inception** - Team members are identified, funding is put in place, and initial environments and requirements are discussed
3. **Iteration/Construction** - The development team works to deliver working software based on iteration requirements and feedback
4. **Release** - QA (Quality Assurance) testing, internal and external training, documentation development, and final release of the iteration into production
5. **Production** - Ongoing support of the software
6. **Retirement** - End-of-life activities, including customer notification and migration

This view presents the full Agile lifecycle model within the enterprise. In any enterprise there may be projects operating simultaneously, multiple sprints/iterations being logged on different product lines, and a variety of customers, both external and internal, with a range of business needs.

**Agile Software Development Lifecycle**



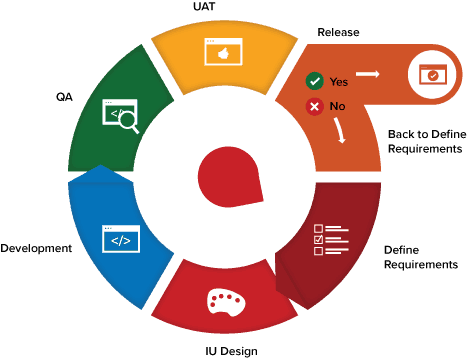
**The Agile Iteration Workflow**

The Agile software development lifecycle is dominated by the iterative process. Each iteration results in the next piece of the software development puzzle - working software and supporting elements, such as documentation, available for use by customers - until the final product is complete. Each iteration is usually two to four weeks in length and has a fixed completion time. Due to its time-bound nature, the iteration process is methodical and the scope of each iteration is only as broad as the allotted time allows.

Multiple iterations will take place during the Agile software development lifecycle and each follows its own workflow. During an iteration, it is important that the customers and business stakeholders provide feedback to ensure that the features meet their needs.

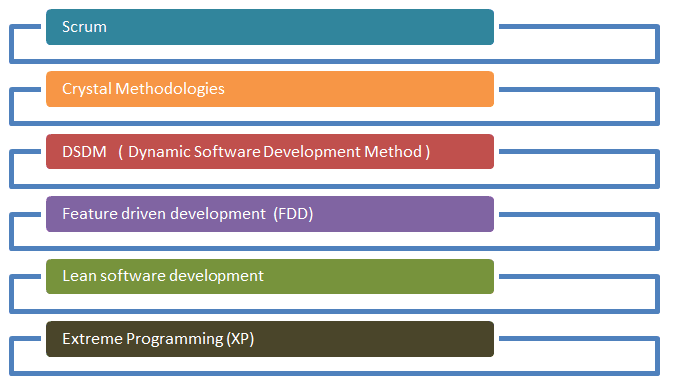
A typical iteration process flow can be visualized as follows:

* **Requirements** - Define the requirements for the iteration based on the product backlog, sprint backlog, customer and stakeholder feedback
* **Development** - Design and develop software based on defined requirements
* **Testing** - QA (Quality Assurance) testing, internal and external training, documentation development
* **Delivery** - Integrate and deliver the working iteration into production
* **Feedback** - Accept customer and stakeholder feedback and work it into the requirements of the next iteration



For the duration of the project, while additional features may be fed into the product backlog, the rest of the process is a matter of repeating the steps over and over until all of the items in the product backlog have been fulfilled. As a result, the process flow is more of a loop and not a linear process.

Below Agile process model to deliver successful systems quickly.

[](https://www.guru99.com/images/11-2014/agile_Processesv1_2.png)

* **3.2.1 History of Agile :-**

During the 1990s, a number of *lightweight* software development methods evolved in reaction to the prevailing *heavyweight* methods (often referred to collectively as [*waterfall*](https://en.wikipedia.org/wiki/Waterfall_model)) that critics described as overly regulated, planned, and [micromanaged](https://en.wikipedia.org/wiki/Micromanagement). These included: [Rapid Application Development](https://en.wikipedia.org/wiki/Rapid_application_development) (RAD), from 1991; the [Unified Process](https://en.wikipedia.org/wiki/Unified_Process) (UP) and [Dynamic Systems Development Method](https://en.wikipedia.org/wiki/Dynamic_systems_development_method) (DSDM), both from 1994; [Scrum](https://en.wikipedia.org/wiki/Scrum_(software_development)), from 1995; Crystal Clear and [eXtreme Programming](https://en.wikipedia.org/wiki/Extreme_programming) (XP), both from 1996; and [Feature-Driven Development](https://en.wikipedia.org/wiki/Feature-driven_development), from 1997. Although these all originated before the publication of the *Agile Manifesto*, they are now collectively referred to as agile software development methods. At the same time, similar changes were underway in manufacturing and management thinking.

However, people started working in an Agile fashion before that 2001 meeting. Starting in the mid-nineties, there were various practitioners, either people working inside organizations developing software products or consultants helping organizations build software who thought, *“You know what? The way we’ve been building software just isn’t working for us. We’ve got to come up with something different.”*

These software developers started mixing old and new ideas, and when they found a combination that worked, they created a methodology for their team to help them remember the combination of ideas that worked in a given situation.

These methodologies emphasized close collaboration between the development team and business stakeholders; frequent delivery of business value, tight, self-organizing teams; and smart ways to craft, confirm, and deliver code.

The people who created those methodologies figured that others might be interested in getting some of the same benefits they were experiencing, so they created frameworks to spread the ideas to other teams in other organizations and contexts. This is where frameworks such as Scrum, Extreme Programming, Feature-Driven Development (FDD), and Dynamic Systems Development Method (DSDM), among others, started to appear.

The spread of the ideas at this time was very organic, and all of those different approaches started to grow in a very grassroots manner. People borrowed the original frameworks and tweaked them with different practices to make them appropriate for their own contexts.

There wasn’t a consistent way of describing these different ways to develop software until a group of 17 people thought, “We’re all doing these different approaches to developing software. We ought to get together and see where there are commonalities in what we’re thinking about.” The result was a meeting at a ski resort in Snowbird, Utah in 2001.

When they got together, they did some skiing and also discussed where their approaches to software development had commonalities and differences.

They didn’t agree upon a lot of things, but there were a few things that they were able to agree upon, and that ended up becoming the Manifesto for Agile Software Development. The two main things the Agile Manifesto did was to provide a set of value statements that form the foundation for Agile software development and to coin the term Agile software development itself.

In the months afterward, the authors expanded on the ideas of the Agile Manifesto with the 12 Principles Behind the Agile Manifesto.

(*In 2001, these seventeen software developers met at a resort in*[*Snowbird*](https://en.wikipedia.org/wiki/Snowbird,_Utah)*,*[*Utah*](https://en.wikipedia.org/wiki/Utah)*to discuss these lightweight development methods:*[*Kent Beck*](https://en.wikipedia.org/wiki/Kent_Beck)*,*[*Ward Cunningham*](https://en.wikipedia.org/wiki/Ward_Cunningham)*,*[*Dave Thomas*](https://en.wikipedia.org/wiki/Dave_Thomas_(programmer))*,*[*Jeff Sutherland*](https://en.wikipedia.org/wiki/Jeff_Sutherland)*,*[*Ken Schwaber*](https://en.wikipedia.org/wiki/Ken_Schwaber)*,*[*Jim Highsmith*](https://en.wikipedia.org/wiki/Jim_Highsmith)*,* [*Alistair Cockburn*](https://en.wikipedia.org/wiki/Alistair_Cockburn)*,*[*Robert C. Martin*](https://en.wikipedia.org/wiki/Robert_C._Martin)*,*[*Mike Beedle*](https://en.wikipedia.org/wiki/Mike_Beedle)*,*[*Arie van Bennekum*](https://en.wikipedia.org/w/index.php?title=Arie_van_Bennekum&action=edit&redlink=1)*,*[*Martin Fowler*](https://en.wikipedia.org/wiki/Martin_Fowler_(software_engineer))*, James Grenning,*[*Andrew Hunt*](https://en.wikipedia.org/wiki/Andy_Hunt_(author))*,*[*Ron Jeffries*](https://en.wikipedia.org/wiki/Ron_Jeffries)*,*[*Jon Kern*](https://en.wikipedia.org/w/index.php?title=Jon_Kern&action=edit&redlink=1)*, Brian Marick, and*[*Steve Mellor*](https://en.wikipedia.org/wiki/Stephen_J._Mellor)*.* ***Together they published the Manifesto for Agile Software Development.***)

In 2005, a group headed by Cockburn and Highsmith wrote an addition of [project management](https://en.wikipedia.org/wiki/Project_management) principles, the PM Declaration of Interdependence, to guide software project management according to agile software development methods.

In 2009, a group working with Martin wrote an extension of [software development](https://en.wikipedia.org/wiki/Software_development) principles, the [Software Craftsmanship Manifesto](https://en.wikipedia.org/wiki/Software_craftsmanship), to guide agile software development according to [professional](https://en.wikipedia.org/wiki/Professional) conduct and mastery.

In 2011, the Agile Alliance created the *Guide to Agile Practices* (renamed the *Agile Glossary* in 2016), an evolving open-source compendium of the working definitions of agile practices, terms, and elements, along with interpretations and experience guidelines from the worldwide community of agile practitioners.

* **3.2.2 :- The Manifesto for Agile Software Development**

Based on their combined experience of developing software and helping others do that, the seventeen signatories to the manifesto proclaimed that they value:[[5]](https://en.wikipedia.org/wiki/Agile_software_development#cite_note-AgileManifesto-5)

* ***Individuals and interactions****over processes and tools*
* ***Working software****over comprehensive documentation*
* ***Customer collaboration****over contract negotiation*
* ***Responding to change****over following a plan*
* **Another way to describe the Agile Manifesto:-**
* **Individuals and interactions**− In Agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.
* **Working software** − Demo working software is considered the best means of communication with the customers to understand their requirements, instead of just depending on documentation.
* **Customer collaboration** − As the requirements cannot be gathered completely in the beginning of the project due to various factors, continuous customer interaction is very important to get proper product requirements.
* **Responding to change** − Agile Development is focused on quick responses to change and continuous development.
* **3.2.3:- The 12 Principle for Agile software development**

The *Manifesto for Agile Software Development* is based on 12 principles:

1. Customer satisfaction by early and continuous delivery of valuable software.
2. Welcome changing requirements, even in late development.
3. Deliver working software frequently (weeks rather than months).
4. Close, daily cooperation between business people and developers.
5. Projects are built around motivated individuals, who should be trusted.
6. Face-to-face conversation is the best form of communication (co-location).
7. Working software is the primary measure of progress.
8. Sustainable development, able to maintain a constant pace.
9. Continuous attention to technical excellence and good design.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. Best [architectures](https://en.wikipedia.org/wiki/Agile_Architecture), requirements, and designs emerge from self-organizing teams
12. Regularly, the team reflects on how to become more effective, and adjusts accordingly

* **Another Way describe 12 Principle of Agile Manifesto:-**

1. **Customer Satisfaction:** Manifesto provides high priority to satisfy the costumer's requirements. This is done through early and continuous delivery of valuable software.
2. **Welcome Change:** Making changes during software development is common and inevitable. Every changing requirement should be welcome, even in the late development phase. Agile process works to increase the customers' competitive advantage.
3. **Deliver the Working Software:** Deliver the working software frequently, ranging from a few weeks to a few months with considering the shortest time period.
4. **Collaboration:** Business people (Scrum Master and Project Owner) and developers must work together during the entire life of a project development phase.
5. **Motivation:** Projects should be build around motivated team members. Provide such environment that supports individual team members and trust them. It makes them feel responsible for getting the job done thoroughly.
6. **Face-to-face Conversation:** Face-to-face conversation between Scrum Master and development team and between the Scrum Master and customers for the most efficient and effective method of conveying information to and within a development team.
7. **Measure the Progress as per the Working Software:** The working software is the key and primary measure of the progress.
8. **Maintain Constant Pace:** The aim of agile development is sustainable development. All the businesses and users should be able to maintain a constant pace with the project.
9. **Monitoring:** Pay regular attention to technical excellence and good design to maximize agility.
10. **Simplicity:** Keep things simple and use simple terms to measure the work that is not completed.
11. **Self-organized Teams:** The Agile team should be self-organized. They should not be depending heavily on other teams because the best architectures, requirements, and designs emerge from self-organized teams.
12. **Review the Work Regularly:** The work should be reviewed at regular intervals, so that the team can reflect on how to become more productive and adjust its behavior accordingly.

* **3.3 Key Agile Concept :-**
* **3.3.1 User Stories and Story Points:-**

*A* ***user story*** *is an informal, general explanation of a software feature written from the perspective of the end user. Its purpose is to articulate how a software feature will provide value to the customer.*

It's tempting to think that user stories are, simply put, software system requirements. But they're not.

**A key component of agile software development** is putting people first, and a user story puts end users at the center of the conversation. These stories use non-technical language to provide context for the development team and their efforts. After reading a user story, the team knows why they are building, what they're building, and what value it creates.

**User stories are one of the core components of an agile program**. They help provide a user-focused framework for daily work — which drives collaboration, creativity, and a better product overall.

**What are agile user stories?**

A user story is the smallest unit of work in an agile framework. It’s an end goal, not a feature, expressed from the software user’s perspective.

A user story is an informal, general explanation of a software feature written from the perspective of the end user or customer.

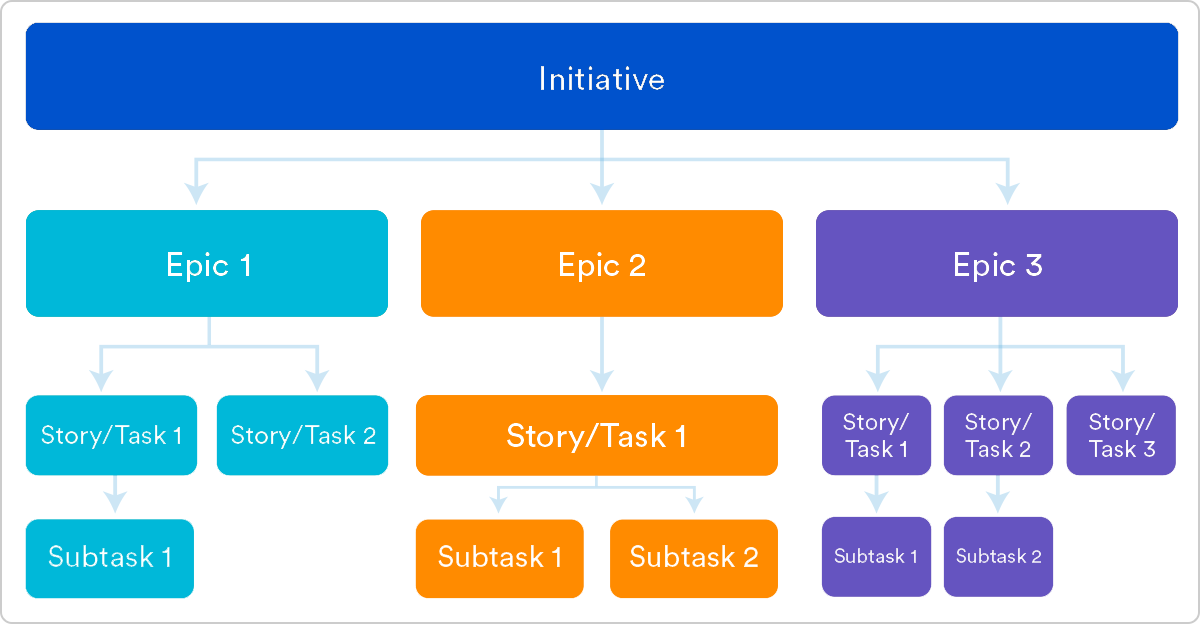
The purpose of a user story is to articulate how a piece of work will deliver a particular value back to the customer. Note that "customers" don't have to be external end users in the traditional sense, they can also be internal customers or colleagues within your organization who depend on your team.

User stories are a few sentences in simple language that outline the desired outcome. They don't go into detail. Requirements are added later, once agreed upon by the team.

Stories fit neatly into agile frameworks like scrum and kanban. In scrum, user stories are added to sprints and “burned down” over the duration of the sprint. Kanban teams pull user stories into their backlog and run them through their workflow. It’s this work on user stories that help scrum teams get better at [estimation](https://www.atlassian.com/agile/project-management/estimation) and sprint planning, leading to more accurate forecasting and greater agility. Thanks to stories, kanban teams learn how to manage work-in-progress (WIP) and can further refine their workflows.

User stories are also the building blocks of larger agile frameworks like epics and initiatives. Epics are large work items broken down into a set of stories, and multiple epics comprise an initiative. These larger structures ensure that the day to day work of the development team (on stores) contributes to the organizational goals built into epics and initiatives.

**Learn more about epics and initiatives :-**



**Why create user stories?**

For development teams new to agile, user stories sometimes seem like an added step. Why not just break the big project ([the epic](https://www.atlassian.com/agile/project-management/epics)) into a series of steps and get on with it? But stories give the team important context and associate tasks with the value those tasks bring.

**User stories serve a number of key benefits:**

* **Stories keep the focus on the user.** A To Do list keeps the team focused on tasks that need checked off, but a collection of stories keeps the team focused on solving problems for real users.
* **Stories enable collaboration.** With the end goal defined, the team can work together to decide how best to serve the user and meet that goal.
* **Stories drive creative solutions.** Stories encourage the team to think critically and creatively about how to best solve for an end goal.
* **Stories create momentum.** With each passing story the development team enjoys a small challenges and a small win, driving momentum.

**Working with user stories**

Once a story has been written, it’s time to integrate it into your workflow. Generally a story is written by the product owner, product manager, or program manager and submitted for review.

During a sprint or iteration planning meeting, the team decides what stories they’ll tackle that sprint. Teams now discuss the requirements and functionality that each user story requires. This is an opportunity to get technical and creative in the team’s implementation of the story. Once agreed upon, these requirements are added to the story.

Another common step in this meeting is to score the stories based on their complexity or time to completion. Teams use t-shirt sizes, the Fibonacci sequence, or planning poker to make proper estimations. A story should be sized to complete in one sprint, so as the team specs each story, they make sure to break up stories that will go over that completion horizon.

**How to write user stories**

Consider the following when writing user stories:

* **Definition of “Done”** — The story is generally “done” when the user can complete the outlined task, but make sure to define what that is.
* **Outline subtasks or tasks** — Decide which specific steps need to be completed and who is responsible for each of them.
* **User personas** — For Whom? If there are multiple end users, consider making multiple stories.
* **Ordered Steps** — Write a story for each step in a larger process.
* **Listen to feedback** — Talk to your users and capture the problem or need in their words. No need to guess at stories when you can source them from your customers.
* **Time** — Time is a touchy subject. Many development teams avoid discussions of time altogether, relying instead on their estimation frameworks. Since stories should be completable in one sprint, stories that might take weeks or months to complete should be broken up into smaller stories or should be considered their own epic.

Once the user stories are clearly defined, make sure they are visible for the entire team.

**User story template and examples**

User stories are often expressed in a simple sentence, structured as follows:

**“As a [persona], I [want to], [so that].”**

Breaking this down:

* **"As a [persona]"**: Who are we building this for? We’re not just after a job title, we’re after the persona of the person. Sachin. Our team should have a shared understanding of who Sachin is. We’ve hopefully interviewed plenty of Sachin’s. We understand how that person works, how they think and what they feel. We have empathy for Sachin.
* **“Wants to”:** Here we’re describing their intent — not the features they use. What is it they’re actually trying to achieve? This statement should be implementation free — if you’re describing any part of the UI and not what the user goal is you're missing the point.
* **“So that”**: how does their immediate desire to do something this fit into their bigger picture? What’s the overall benefit they’re trying to achieve? What is the big problem that needs solving?

For example, user stories might look like:

* As Sachin, I want to invite my friends, so we can enjoy this service together.
* As Organizer, I want to organize my work, so I can feel more in control.
* As a manager, I want to be able to understand my colleagues progress, so I can better report our success and failures.

This structure is not required, but it is helpful for defining done. When that persona can capture their desired value, then the story is complete. We encourage teams to define their own structure, and then to stick to it.

**Getting started with agile user stories**

User stories describe the why and the what behind the day-to-day work of development team members, often expressed as *persona + need + purpose*. Understanding their role as the source of truth for what your team is delivering, but also why, is key to a smooth process.

Start by evaluating the next, or most pressing, large project (e.g. an epic). Break it down into smaller user stories, and work with the development team for refinement. Once your stories are out in the wild where the whole team can see them, you’re ready to get to work.

**User stories** are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system. They typically follow a simple template:

**As a < type of user >, I want < some goal > so that < some reason >.**

* As a user, I can backup my entire hard drive.
* As a power user, I can specify files or folders to backup based on file size, date created and date modified.
* As a user, I can indicate folders not to backup so that my backup drive isn't filled up with things I don't need saved.

**There are quite a few benefits for adopting user story approach in agile development such as:**

* The simple and consistent format saves time when capturing and prioritizing requirements while remaining versatile enough to be used on large and small features alike.
* Keep yourself expressing business value by delivering a product that the client really needs
* Avoid introducing detail too early that would prevent design options and inappropriately lock developers into one solution.
* Avoid the appearance of false completeness and clarity
* Get to small enough chunks that invite negotiation and movement in the backlog
* Leave the technical functions to the architect, developers, testers, and so on

**Basic Concepts of User Story**

A user story is a lightweight method for quickly capturing the "who", "what" and "why" of a product requirement. In simple terms, user stories are stated ideas of requirements that express what users need. A user story is defined incrementally, in three stages:

* The brief description of the need
* The conversations that happen during backlog grooming and iteration planning to solidify the details
* The tests that confirm the story's satisfactory completion

And these, although, are known as the 3C's - Card, Conversation and Confirmation. We will talk more about this later on in this user story guide.

**User Stories - INVEST**

The acronym INVEST helps to remember a widely accepted set of criteria, or checklist, to assess the quality of a user story. If the story fails to meet one of these criteria, the team may want to reword it, or even consider a rewrite (which often translates into physically tearing up the old story card and writing a new one).

A good user story should be - **INVEST**:

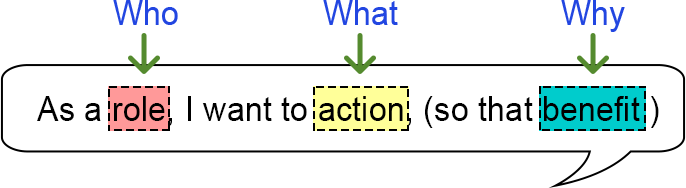
* I**ndependent**: Should be self-contained in a way that allows to be released without depending on one another.
* N**egotiable**: Only capture the essence of user's need, leaving room for conversation. User story should not be written like contract.
* V**aluable**: Delivers value to end user.
* E**stimable**: User stories have to able to be estimated so it can be properly prioritized and fit into sprints.
* S**mall**: A user story is a small chunk of work that allows it to be completed in about 3 to 4 days.
* T**estable**: A user story has to be confirmed via pre-written acceptance criteria.

**User Story Template**

User stories only capture the essential elements of a requirement:

* Who it is for?
* What it expects from the system?
* Why it is important (optional?)?

Here is a simple format of user story used by 70% of practitioners:



**Role** - The user should be an actual human who interacts with the system.

* Be as specific as possible
* The development team is NOT a user

**Action** - The behavior of the system should be written as an action.

* Usually unique for each User Story
* The "system" is implied and does not get written in the story
* Active voice, not passive voice ("I can be notified")

**Benefits** - The benefit should be a real-world result that is non-functional or external to the system.

* Many stories may share the same benefit statement.
* The benefit may be for other users or customers, not just for the user in the story.

A user story is a requirement which is formulated into few sentences. The user requirement is the everyday language of user. This user story should be completed within iteration. The user story is done when

* All the related code and documentation have been checked-in.
* The product passed all the processes of unit test.
* All the processes of the acceptance test case have been moved.
* The product owner must have accepted the story.
* The help text (documentation) is written.
* **3.3.1. Story Points :-**

A story point is a metric used in [agile project management and development](https://www.visual-paradigm.com/scrum/what-is-agile-software-development/) to estimate the difficulty of implementing a given user story, which is an abstract measure of effort required to implement it. In simple terms, a story point is a number that tells the team about the difficulty level of the story. Difficulty could be related to complexities, risks, and efforts involved.

Story points are a unit of measure for expressing an estimate of the overall effort that will be required to fully implement a product backlog item or any other piece of work.

**What Goes Into a Story Point?**

Because story points represent the effort to develop a story, a team’s estimate must include everything that can affect the effort. That could include:

* **The amount of work to do**
* **The complexity of the work**
* **Any risk or uncertainty in doing the work**

**The Amount of Work to Do**

Certainly, if there is more to do of something, the estimate of effort should be larger. Consider the case of developing two web pages. The first page has only one field and a label asking to enter a name. The second page has 100 fields to also simply be filled with a bit of text.

The second page is no more complex. There are no interactions among the fields and each is nothing more than a bit of text. There’s no additional risk on the second page. The only difference between these two pages is that there is more to do on the second page.

The second page should be given more story points. It probably doesn’t get 100 times more points even though there are 100 times as many fields. There are, after all, economies of scale and maybe making the second page is only 2 or 3 or 10 times as much effort as the first page.

**Risk and Uncertainty**

The amount of risk and uncertainty in a product backlog item should affect the story point estimate given to the item.

If a team is asked to estimate a product backlog item and the stakeholder asking for it is unclear about what will be needed, that uncertainty should be reflected in the estimate.

If implementing a feature involves changing a particular piece of old, brittle code that has no automated tests in place, that risk should be reflected in the estimate.

**Complexity**

Complexity should also be considered when providing a story point estimate. Think back to the earlier example of developing a web page with 100 trivial text fields with no interactions between them.

Now think about another web page also with 100 fields. But some are date fields with calendar widgets that pop up. Some are formatted text fields like phone numbers or Social Security numbers. Other fields do checksum validations as with credit card numbers.

This screen also requires interactions between fields. If the user enters a Visa card, a three-digit CVV field is shown. But if the user enters an American Express card, a four-digit CVV field is shown.

Even though there are still 100 fields on this screen, these fields are harder to implement. They’re more complex. They’ll take more time. There’s more chance the developer makes a mistake and has to back up and correct it

Story points are units of measure for expressing an estimate of the overall effort required to fully implement a product backlog item or any other piece of work. Teams assign story points relative to work complexity, the amount of work, and risk or uncertainty. Values are assigned to more effectively break down work into smaller pieces, so they can address uncertainty. Over time, this helps teams understand how much they can achieve in a period of time and builds consensus and commitment to the solution.  It may sound counter-intuitive, but this abstraction is actually helpful because it pushes the team to make tougher decisions around the difficulty of work. Here are few reasons to use story points:

* Dates don’t account for the non-project related work that inevitably creeps into our days: emails, meetings, and interviews that a team member may be involved in.
* Dates have an emotional attachment to them. Relative estimation removes the emotional attachment.
* Each team will estimate work on a slightly different scale, which means their velocity (measured in points) will naturally be different. This, in turn, makes it impossible to play politics using velocity as a weapon.
* Once you agree on the relative effort of each story point value, you can assign points quickly without much debate.
* Story points reward team members for solving problems based on difficulty, not time spent. This keeps team members focused on shipping value, not spending time.

Unfortunately, story points are often misused. Story points go wrong when they’re used to judge people, assign detailed timelines and resources, and when they’re mistaken for a measure of productivity. Instead, teams should use story points to understand the size of the work and the prioritization of the work.

* **3.3.2 Product Backlog :-**

A product backlog is a list of items to be done. Items are ranked with feature descriptions. In an ideal scenario, items should be broken down into user stories.

The agile product backlog in [Scrum](https://www.javatpoint.com/agile-scrum) is a list of prioritized features. It contains a short description of all the functionalities desired in the product. In usual scenario, items should be broken down into user stories. Commonly, a Scrum team and its product owner write everything that they can think for agile backlog prioritization.

This is a repository where requirements are tracked with details on the no of requirements(user stories) to be completed for each release. It should be maintained and prioritized by Product Owner, and it should be distributed to the scrum team. Team can also request for a new requirement addition or modification or deletion

**Why Product Backlog is Important?**

* It is prepared so that estimates can be given to each and every feature.
* It helps in planning the roadmap for the product.
* It helps in re-ranking the features so that more value can be added to the product.
* It helps in determining what to prioritize first. Team ranks the item and then builds value. The team member works first on the higher prioritize product.

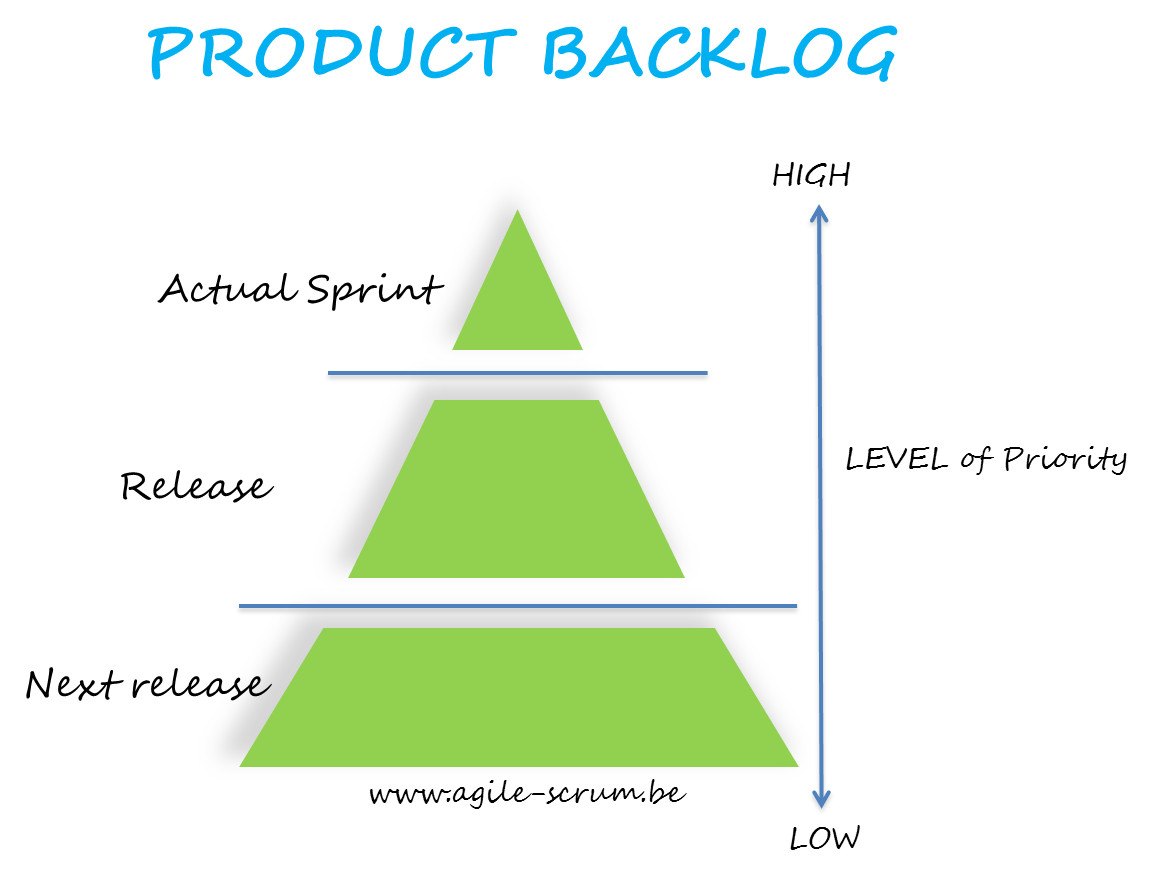
**Characteristics of Product Backlog**

* Each product should have one product backlog which can have a set of large to very large features.
* Multiple teams can work on a single product backlog.
* Ranking of features is done based on business value, technical value, risk management or strategic fitness.
* Highest ranking items are decomposed into smaller stories during release planning so that they can be completed in future iterations.

**The Product Backlog comprises the following different types of items:**

* Features
* Bugs
* Technical work
* Knowledge acquisition

In Agile development, a product backlog is a prioritized list of deliverables (such as new features) that should be implemented as part of a project or product development.



It's a decision-making artifact that helps you estimate, refine, and prioritize everything you *might sometime in the future*want to complete.

It helps ensure the team is working on the most important and valuable features, fixing the most important bugs, or doing other important work critical to product development.

The backlog, therefore, is tremendously useful in situations when you are unable to do everything being asked (so, most situations), or in contexts when even a small amount of planning will help a lot (so, in most contexts).

*Many think of this backlog as a to-do list, and define it in exactly this way, as a list of things you must do to deliver your product to market.*

*In truth, it is not necessarily a to-do list. Think of it as a wish list.*

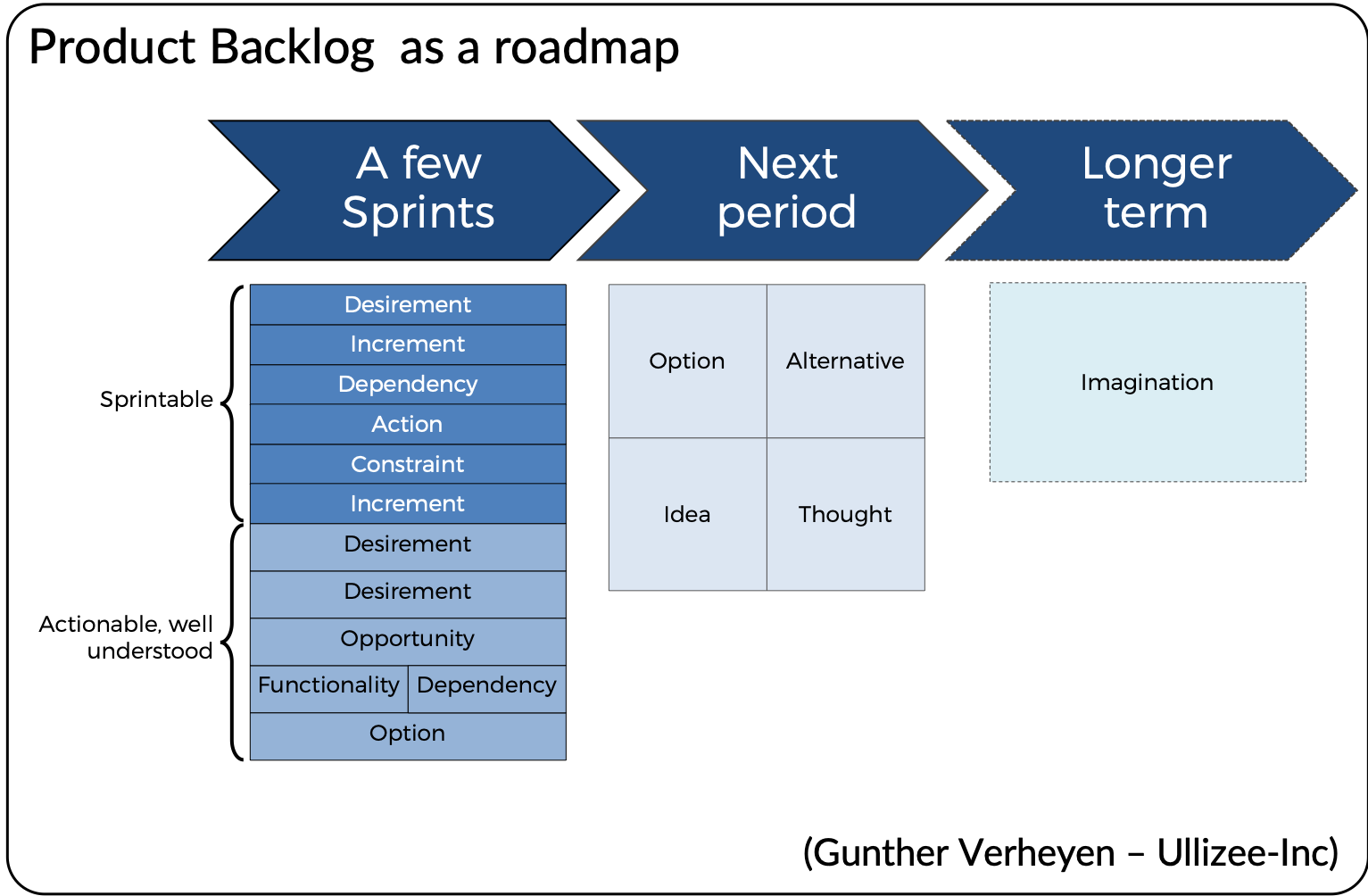
*Think of a product backlog as a* ***wish list — not a to-do list.***

**Why?**

When we think about the backlog as a wish list, we invite flexibility and change. In doing so, we enable true agility and give the organization a power that’s necessary to win in the marketplace today: the power to change its mind.

In this context, the purpose of the backlog can be reduced to three simple goals.

* Develop a common ground to align stakeholders and teams, so that teams implement the most valuable user stories.
* Provide flexibility to adapt to new needs and realities.
* Improve the accuracy of product release forecasts by creating a common denominator across many teams collaborating on one product.



* **3.3.3 Sprint Backlog :-**

The **Sprint** is a fixed period of time (1 to 4 Weeks) in which a defined set of activities take place and at the end of which a product increment is created.

The **Sprint Backlog** is a list of tasks identified by the Scrum team to be completed during the [Scrum](https://www.mountaingoatsoftware.com/agile/scrum) sprint. During the sprint planning meeting, the team selects some number of product backlog items, usually in the form of user stories, and identifies the tasks necessary to complete each user story. Most teams also estimate how many hours each task will take someone on the team to complete.

It's critical that the team selects the items and size of the sprint backlog. Because they are the people committing to completing the tasks, they must be the people to choose what they are committing to during the Scrum sprint.

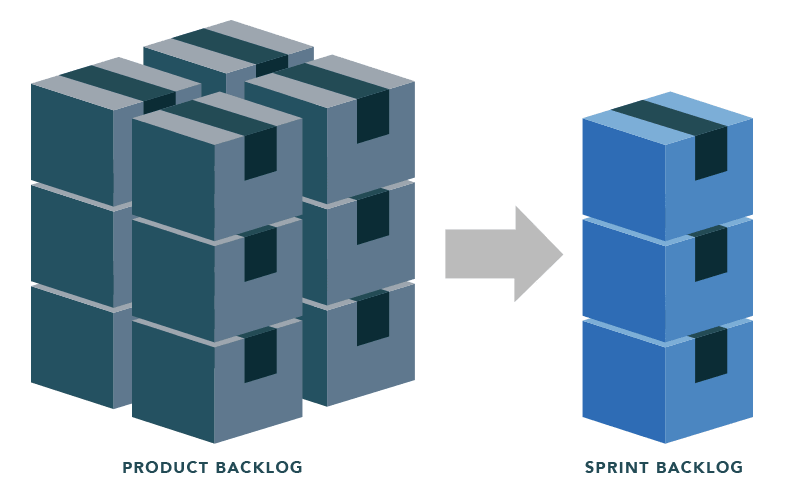
The sprint backlog is commonly maintained as a spreadsheet, but it is also possible to use your defect tracking system or any of a number of software products designed specifically for Scrum or agile

*The Sprint Backlog is composed of the Sprint Goal (why), the set of Product Backlog items selected for the Sprint (what), as well as an actionable plan for delivering the Increment (how).*

The Sprint Backlog is a plan by and for the Developers. It is a highly visible, real-time picture of the work that the Developers plan to accomplish during the Sprint in order to achieve the Sprint Goal. Consequently, the Sprint Backlog is updated throughout the Sprint as more is learned. It should have enough detail that they can inspect their progress in the Daily Scrum.

**Agile Product Backlog vs. Sprint Backlog**

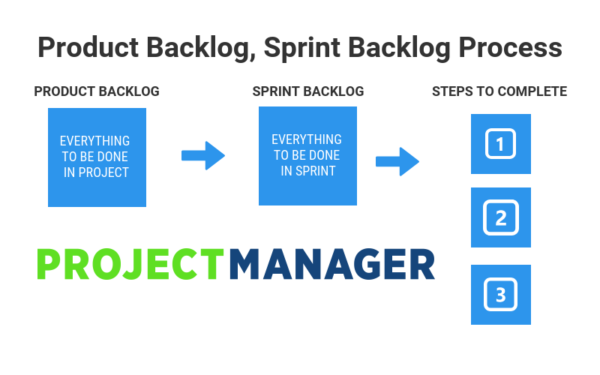
Content, granularity, and immediacy are three key differences between the product backlogs and sprint backlogs.

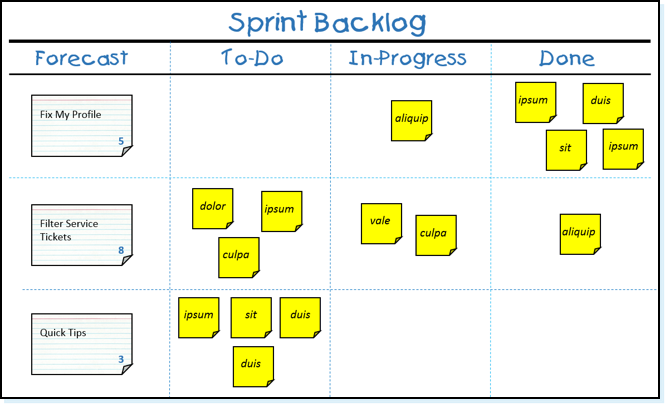


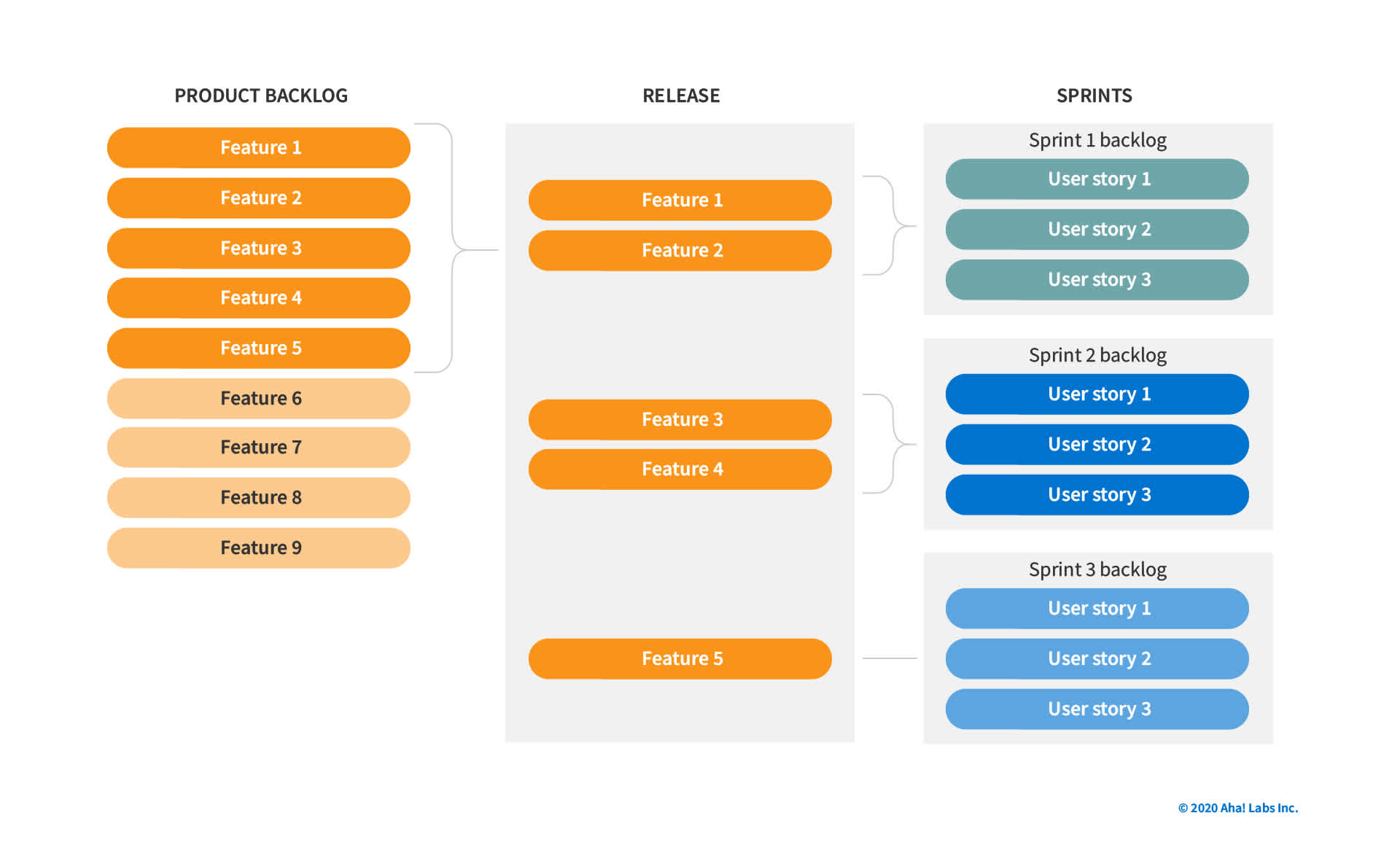
In short, the sprint backlog is the short-term plan for the team’s sprint. The product backlog is the long-term plan for the product, where the vision is itemized into concrete deliverable items that make the product more valuable. Many think of the sprint backlog as a subset of the product one. Ideally, this is true; the sprint backlog consists solely of items from the product backlog. In practice, however, a sprint backlog will include other work the team has committed to.

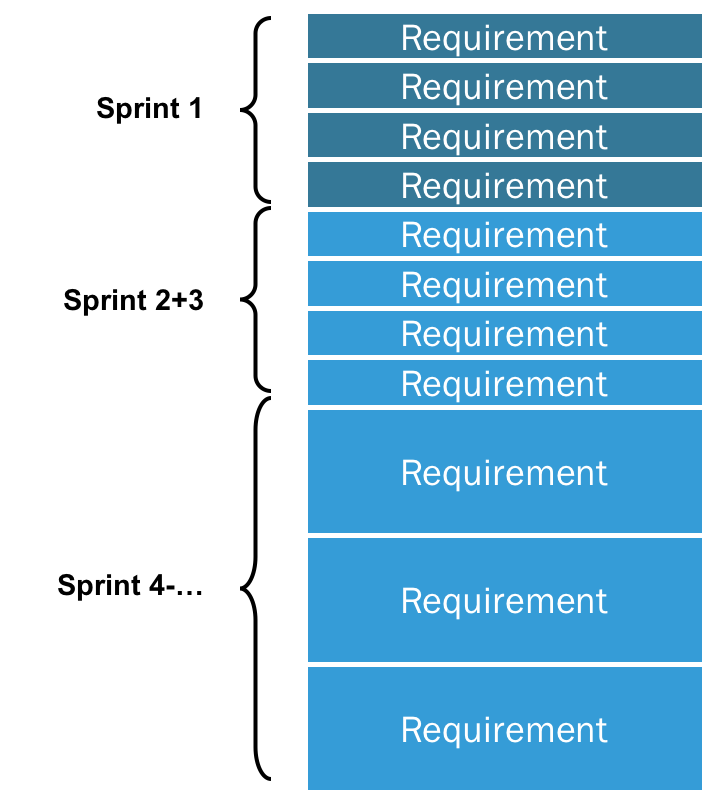
**The product backlog** is a container for work you think you will do in the future to keep your product competitive. It is the output of the product owner in collaboration with stakeholders (customers, the team, analysts). It will change frequently, with items being added or taken out on a regular basis. It will be larger than the sprint backlog, generally. It will also have items with a mix of granularity; with fewer items broken down below the user story level. It is overseen by the product owner.

**The sprint backlog**is a container for work the team is committed to doing, either right now as a part of the sprint (typically a one- to four-week period). It is an output of a sprint planning meeting attended by the team. The sprint backlog, ideally, doesn’t change at all for duration of the sprint. It consists of user stories that the team has committed to delivering within the next sprint’s timeframe. But it can also include bugs, refactoring work, and so forth. It is usually more granular, and broken down into tasks, focusing on the technical implementation of a user story. It is the purview of the scrum master and team.









* **3.3.4 Sprint Velocity :-**

By looking at the amount of work your team completed in previous sprints, you should be able to estimate how much work they can do in future sprints. In Agile development, this estimate is known as sprint velocity.

At the end of each iteration, the team adds up effort [estimates](http://guide.agilealliance.org/guide/nuts.html) associated with [user stories](http://guide.agilealliance.org/guide/stories.html) that were [completed](http://guide.agilealliance.org/guide/sashimi.html) during that iteration. This total is called velocity.

Velocity is a measure of the amount of work a Team can tackle during a single Sprint and is the key metric in Scrum. Velocity is calculated at the end of the Sprint by totaling the Points for all fully completed [User Stories](https://www.scruminc.com/independent-user-stories/).

Knowing velocity, the team can compute (or revise) an estimate of how long the project will take to complete, based on the estimates associated with remaining user stories and assuming that velocity over the remaining iterations will remain approximately the same. This is generally an accurate prediction, even though rarely a precise one.

In the following example, we will use story points to measure the amount of work completed in each sprint. A story point is a measurement used by Agile development teams to estimate how much effort and time it will take to complete a user story.

**Step 1: Count how many user story points are completed in each sprint**

At the end of a sprint, add up how many story points the team completed.

For example, assume that **in sprint 1**:

* The team committed to completing five user stories.
* Each user story had eight story points for a total of 40 story points.
* The team completed three of the five user stories.

**In sprint 2**:

* The team committed to seven user stories (including the two that were not completed in sprint 1).
* Each user story had eight story points for a total of 56 story points.
* The team completed four of the seven user stories.

**In sprint 3:**

* The team committed to nine user stories.
* Each user story had eight story points for a total of 72 story points.
* The team completed five of the nine user stories.

**Step 2: Calculate the average of completed story points**

Simply add up the total of story points completed from each sprint, then divide by the number of sprints.

Sprint 1: 3 user stories x 8 story points = 24

Sprint 2: 4 user stories x 8 story points = 32

Sprint 3: 5 user stories x 8 story points = 40

Total = 96

So, your average sprint velocity is 96 ÷ 3 = 32.

You can now base the amount of work to be done in future sprints on the average of 32 story points. If you have 160 story points remaining to be completed in the project, you can assume that your team will need another five sprints to complete the project.

However, this is just an estimate and is accurate only if variables such as team size and project complexity and scope remain constant. Your teams will experience fluctuations from sprint to sprint. But the sprint velocity estimation is a good starting point to help you determine how much work your team can do.

If your team is new to Agile development, you won’t have any previous sprints to look at. As part of your sprint velocity estimation to-do list, you’ll have to complete a couple of sprints while tracking how many story points are completed in each. Then you will have some useful data that will help calculate an average.

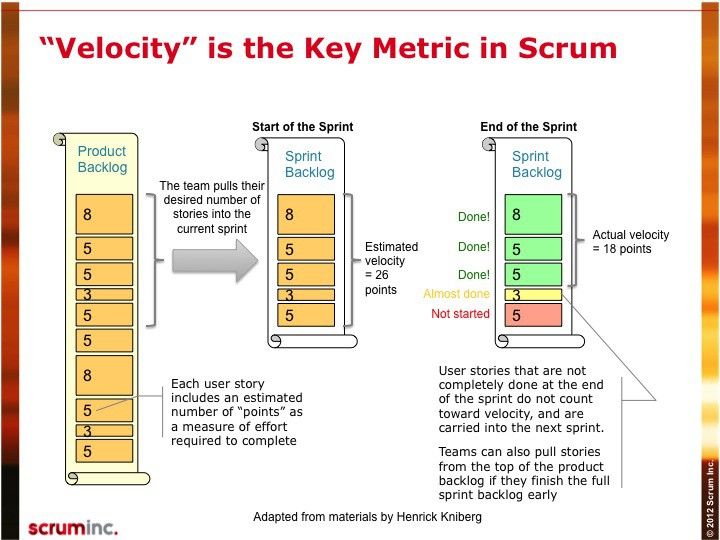
Expected Benefits

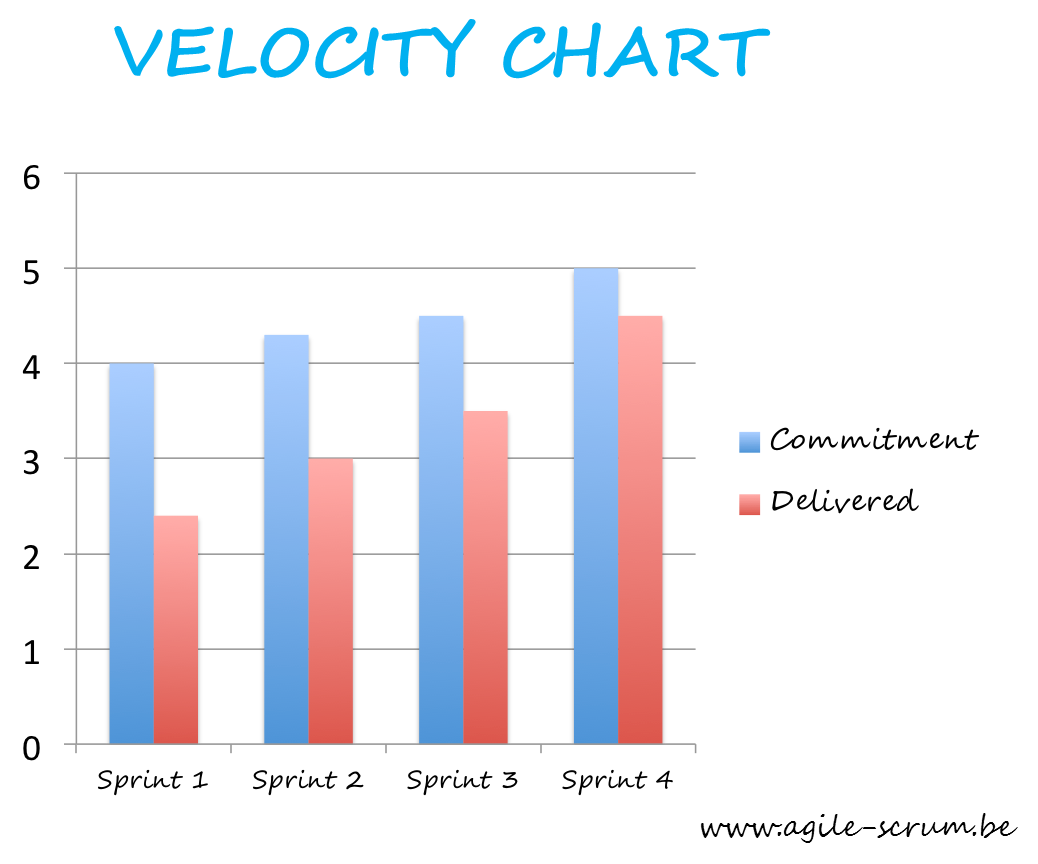
“Worked example:” an agile team has started work on an iteration, planning to complete stories A and B, estimated at 2 points each, and story C, estimated at 3 points. At the end of the iteration, stories A and B are 100% complete but C is only 80% complete.

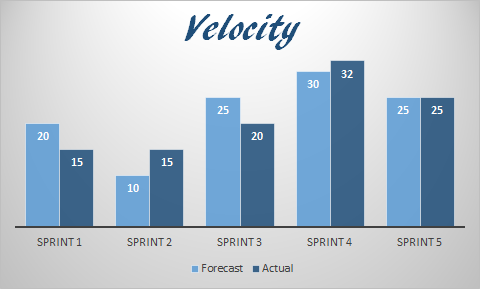
Agile teams generally acknowledge only two levels of completion, 0% done or 100% done. Therefore, C is not counted toward velocity, and velocity as of that iteration is 4 points.

Suppose the user stories remaining represent a total of 40 points; the team’s forecast of the remaining effort for the project is then 10 iterations.

Velocity is also used to limit the amount of work taken on in further iterations. In our example, the team would be well advised to plan for only 4 points’ worth of stories in the next iteration. This doesn’t necessarily mean it will complete only that much; in fact, completing story C in the next iteration might mean that the team’s velocity will, on the contrary, be much higher.



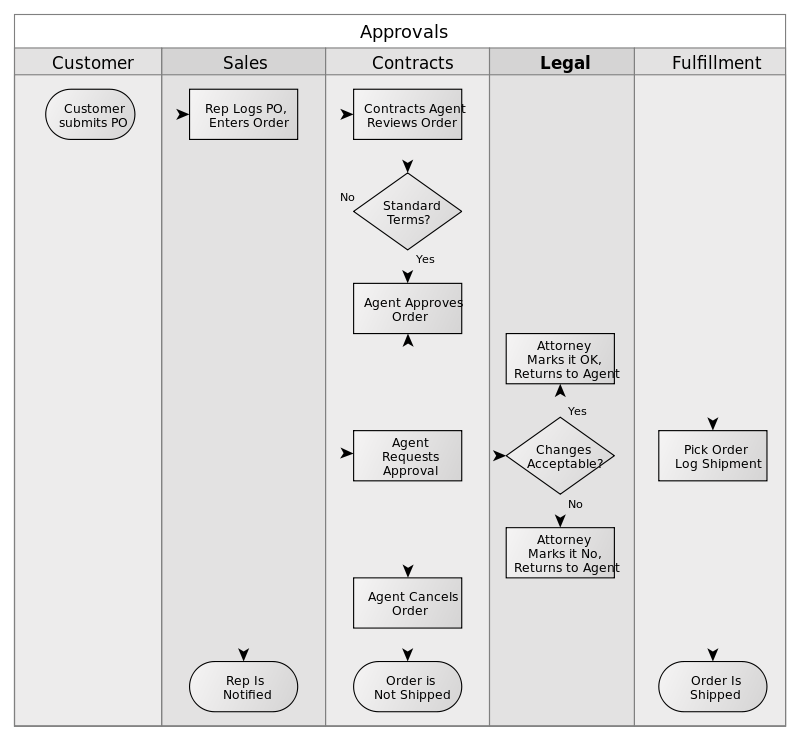




* **3.3.5 Swim lanes :-**

A swimlane is a horizontal categorization of issues in the Active sprints of a Scrum board, You can use swimlanes to help you distinguish tasks of different categories, such as workstreams, users, application areas, etc.

A swimlane is used in [process flow diagrams](https://en.wikipedia.org/wiki/Flowchart), or flowcharts, that visually distinguishes [job sharing](https://en.wikipedia.org/wiki/Job_sharing) and responsibilities for sub-processes of a [business process](https://en.wikipedia.org/wiki/Business_process). Swimlanes may be arranged either horizontally or vertically



The swimlane flowchart differs from other flowcharts in that processes and decisions are grouped visually by placing them in *lanes*. Parallel lines divide the chart into lanes, with one lane for each person, group or sub process. Lanes are labelled to show how the chart is organized.

In the accompanying example, the vertical direction represents the sequence of events in the overall process, while the horizontal divisions depict what sub-process is performing that step. Arrows between the lanes represent how information or material is passed between the sub processes.

Alternately, the flow can be rotated so that the sequence reads horizontally from left to right, with the roles involved being shown at the left edge. This can be easier to read and design, since computer screens are typically wider than they are tall, which gives an improved view of the flow.

Use of standard symbols enables clear linkage to be shown between related flow charts when charting flows with complex relationships.

Swimlane allows easy screening of data by classifying it in different lanes. You can choose to view items by different categories (Responsible, Release, Severity etc). Swimlanes also help you to group data and offer customized views for efficient analysis of your data. Swimlanes are available on different boards including Scrum/Sprint, Kanban, Release, Issue Tracker (list view) and Backlog board.

**Types of Swimlanes**

* Default Swimlane (i.e., no horizontal grouping of data)
* Responsible
* Priority
* Issue Severity

In addition to above, some categories are board specific e.g., Sprint, Release and Epic User Story.

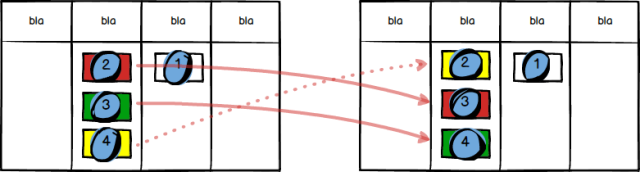
**Tracking with Swimlane**

Swimlane basically groups data in different rows in order to enhance Items visibility on different boards. Each user can use swimlanes individually, according to own needs.

For various Agile teams, grouping issues together based on parent task or other matching criteria is very important. For example, grouping all technical tasks belonging to a story is quite common. But each team has its unique issue data, workflow conditions, and prioritization behavior.

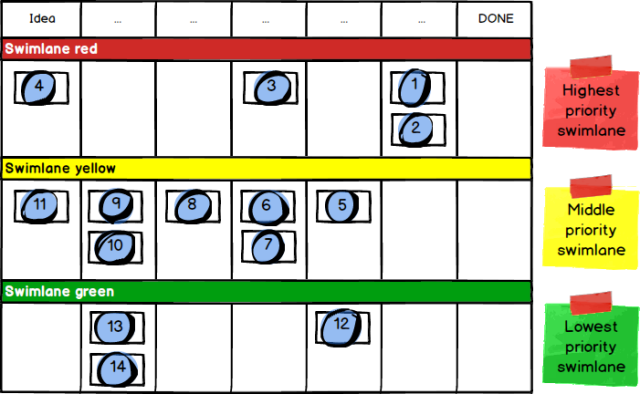
While working on the work/task board, have grouped or categorized the issues together in a vertical way based on issue status. Each column groups issues having the same status or equal to statuses defined for the column.

You can move a ticket *up* to show an increase in priority, but other tickets have to move *down* to make room – your board is both showing and enforcing a trade-off decision:



This example shows tickets moving within a single swimlane. But what if you have multiple swimlanes, stacked vertically one on top of the other? In this case you must stick by the rule that *vertical position is always and only used to show priority*, and that means we must set up our boards such that *higher swimlanes have a greater priority than lower ones*. This restricts the meaning we can give to swimlanes on a board.

Let’s consider the hypothetical board below, which has 14 tickets on it, distributed across three swimlanes. We have used the [*triangulation method*](https://agilefixer.com/2017/02/20/how-to-read-an-agile-board/)to show the order of tickets. Whatever meanings we give to the red, yellow and green swimlanes, it *must* be the case that ticket 4 is more important than ticket 5, and ticket 11 is more important than 12.



Very often you will find Agile Boards are set up with swimlanes corresponding to functional areas, and this may conflict with the principle that *vertical position shows priority*. If it *does* conflict, you should **not** use your swimlanes to show functional areas

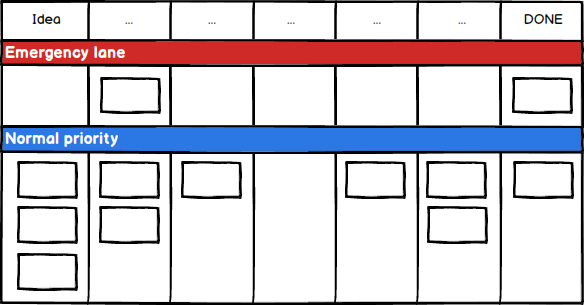
A team building an enterprise level identity system, and three of our functional areas were 1) registration, 2) sign-out and 3) user uplift. As it turns out, there was *not* a clean and strict mapping of priorities onto these functional areas; some registration work was relatively important, and some was not, and the same applied to the other two functional areas, therefore did *not* set up a swimlane for each functional area, as the board would have been falsely indicating the priority of some tickets.

So what *should* you map your swimlanes to, so that you don’t break the *vertical position shows priority* principle? There are no doubt many possible answers, but will consider two swimlane patterns below that between them cover a large percentage of cases.

First let’s consider a team that is in control of its own delivery cadence, and either is already doing, or aspires to move in the direction of, [Continuous Delivery](https://en.wikipedia.org/wiki/Continuous_delivery); in other words, they release work to Live in frequent small batches. What swimlanes might be chosen by such a team?

This team’s board needs to cope with both normal, planned work, and the all-too-common situation that something super-urgent crops up, requiring normal planned work be dropped immediately until the crisis is over. Such emergency situations are typically due to the discovery of a nasty bug in Production.

Since Emergencies are the highest of all possible priorities, they go in the highest lane up the board, with ‘normal work’ sitting below it, as here:



How do these swimlanes get used in practice? We of course want visual representation of work and preferred team behaviour to dovetail neatly and reinforce each other, and this swimlane layout can certainly do so.

Team rules around emergencies are usually something like: 1) verify any claim that new work really deserves ‘emergency’ status, 2) if it does, *everybody who can help with the emergency does so immediately and only returns to other tasks when they can do no more on the emergency work.*

A top-of-the-board Emergency Lane is admirably suited to this. Any ticket in that lane is above the fold on a computer screen, and is the very first thing any user’s eyes fall on. A new ticket arriving in this lane cannot be missed, and is a clear signal for the appropriate group of people to verify if the ticket is a genuine emergency or not, in which case they move the ticket out of the top lane. For tickets that stay in the Emergency Lane, that is again a strong visual signal that work exists that needs full and immediate attention, with team members resuming other work lower down the board only when the emergency allows.

The disruption caused by emergency tickets is considerable, even worse than normal context switching, since a) there is usually no option to get the prior work to a nice, neat point before dropping it, and b) working on an emergency often leaves you in a frazzled mental and emotional state, not easy to switch back out of. This makes it critically important to ensure the Emergency Lane is treated with proper respect, ie for genuine emergencies only. Such disruption is only merited in extreme cases.

There can be a temptation for people to try to slip tickets into the Emergency Lane just to get stuff done faster, and the team need to be on their guard against any freeloading on this crisis-management process. The very term “Emergency Lane” is helpful; less impactful alternative names such as “Expedite” don’t provoke such an emotional response. Saying you want to “expedite” a ticket is really quite different from looking a colleague in the eye and declaring that your work is a genuine emergency!

This two lane pattern of Emergency Lane and Normal Prio Lane works, without much modification, for the majority of modern software development teams. It also applies to most teams outside the world of software

* **3.3.6 Minimum Viable Product (MVP) :-**

***A minimum viable product (MVP) is a version of a product with just enough features to be usable by early customers who can then provide feedback for future***[***product development***](https://en.wikipedia.org/wiki/New_product_development)***.***

A focus on releasing an MVP means that developers potentially avoid lengthy and (ultimately) unnecessary work. Instead, they iterate on working versions and respond to feedback, challenging and validating assumptions about a product's requirements.

 The term was coined and defined in 2001 by Frank Robinson and then popularized by [Steve Blank](https://en.wikipedia.org/wiki/Steve_Blank) and [Eric Ries](https://en.wikipedia.org/wiki/Eric_Ries). It may also involve carrying out market analysis beforehand. The MVP is analogous to experimentation in the scientific method applied in the context of validating business hypotheses, it is utilized so that prospective entrepreneurs would know whether a given business idea would actually be viable and profitable by testing the assumptions behind a product or business idea. The concept can be used to validate a market need for a product and for incremental developments of an existing product. As it tests a potential business model to customers to see how the market would react, it is especially useful for new/startup companies who are more concerned with finding out where potential business opportunities exist rather than executing a prefabricated, isolated business model.

A minimum viable product (MVP) is a concept from Lean Startup that stresses the impact of learning in new product development. Eric Ries, defined an MVP as [that version of a new product which allows a team to collect the maximum amount of validated learning about customers with the least effort](http://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html). This validated learning comes in the form of whether your customers will actually purchase your product.

A key premise behind the idea of MVP is that you produce an actual product (which may be no more than a landing page, or a service with an appearance of automation, but which is fully manual behind the scenes) that you can offer to customers and observe their actual behavior with the product or service. Seeing what people actually do with respect to a product is much more reliable than asking people what they would do.

## Expected Benefits

The primary benefit of an MVP is you can gain understanding about your customers’ interest in your product without fully developing the product. The sooner you can find out whether your product will appeal to customers, the less effort and expense you spend on a product that will not succeed in the market.

## Common Pitfalls

Teams use the term MVP, but don’t fully understand its intended use or meaning. Often this lack of understanding manifests in believing that an MVP is the smallest amount of functionality they can deliver, without the additional criteria of being sufficient to learn about the business viability of the product.

Teams may also confuse an MVP–which has a focus on learning–for a [Minimum Marketable Feature (MMF)](https://www.agilealliance.org/glossary/mmf/) or Minimum Marketable Product (MMP)–which has a focus on earning. There’s not too much harm in this unless the team becomes too focused on delivering something without considering whether it is the **right** something that satisfies customer’s needs.

A minimum viable product has just enough core features to effectively deploy the product, and no more. Developers typically deploy the product to a subset of possible customers—such as [early adopters](https://en.wikipedia.org/wiki/Early_adopter) thought to be more forgiving, more likely to give feedback, and able to grasp a product vision from an early prototype or marketing information.

This strategy targets avoiding building products that customers do not want and seek to maximize information about the customer with the least money spent.

**An example** : in 2015, specialists from the University of Sydney devised the Rippa robot to automate farm and weed (Wild Plant ) management. Before it was released, the underlying hypothesis that the robot's systems can detect weeds from farm plants was tested before the robot project can undergo further development. The application of the MVP method here is that the business hypothesis (that the robot can detect weeds) is tested on a farm, and only if it proves successful will the robot proceed to further development.

"***The minimum viable product is that version of a new product a team uses to collect the maximum amount of validated learning about customers with the least effort.***" The definition's use of the words maximum and minimum means it is not formulaic. It requires judgment to figure out, for any given context, what MVP makes sense.

**Purposes**

* Be able to test a product hypothesis with minimal resources
* Accelerate learning
* Reduce wasted engineering hours
* Get the product to early customers as soon as possible
* Base for other products
* To establish a builder's abilities in crafting the product required
* Brand building very quickly

**Testing**

Testing is the essence of minimum viable products. As described above, an MVP seeks to test out whether an idea works in market environments while using the least possible expenditure. This would be beneficial as it reduces the risk of innovating (so that enormous amounts of capital would not have to be sacrificed before proving that the concept does not actually work), and allowing for gradual, market-tested expansion models such as the real options model.

 A simple method of testing the financial viability of an idea would be discovery-driven planning, which first tests the financial viability of new ventures by carefully examining the assumptions behind the idea by a reverse income statement (first, begin with the income you want to obtain, then the costs the new invention would take, and see if the required amount of revenue that must be gained for the project to work). Results from a minimum viable product test aim to indicate if the product should be built, to begin with. Testing evaluates if the initial problem or goal is solved in a manner that makes it reasonable to move forward.

**Notable quote**

* Steve Blank: "*You’re selling the vision and delivering the minimum feature set to visionaries, not everyone.*"

**Marketing**

Releasing and assessing the impact of a minimum viable product is a market testing strategy that is used to screen product ideas soon after their generation. In software development, the release is facilitated by [rapid application development](https://en.wikipedia.org/wiki/Rapid_application_development) tools and languages common to [web application](https://en.wikipedia.org/wiki/Web_application) development.

The MVP differs from the conventional market testing strategy of investing time and money early to implement a product before testing it in the market.

The MVP is intended to ensure that the market *wants the product* before large time and monetary investments are made.

The MVP differs from the open-source software methodology of [*release early, release often*](https://en.wikipedia.org/wiki/Release_early,_release_often) that listens to users, letting them define the features and future of the product.

The MVP starts with a product vision, which is maintained throughout the product life cycle, although it is adapted based on the explicit and implicit (indirect measures) feedback from potential future customers of the product.

**Customers**

Customers on the Business Model Canvas denote to whom a value proposition is considered for. Utilizing the minimum viable concept here would be useful to determine whether the selected customer segment actually wants that product, either from questionnaires or experimental launches. Whichever method is chosen, the key in using the MVP is to spend as little as possible while learning as much as possible, thus in this case validating the market with the least possible cost.

**Minimum viable brand (MVB)**

Using a minimum viable brand (MVB) concept can ensure brand hypotheses are grounded in strategic intent and market insights.

**Minimum viable co-founder**

Finding other people to create a minimum viable product is a common challenge for new companies and startups. The concept of minimum viable co-founder is based on looking for a co-founder with the following attributes:

* Trust
* Exceptional at building or selling
* Company commitment
* Personally likable
* Productivity
* Reasonable
* Rational
* Realistic

**Minimum viable team**

Founders with an early-stage company are faced with the challenge of building a team with minimal people and cost. The process starts by listing out basic functions of a particular company (e.g., engineer, operations, finance) and then stripping down to the abstract job activities and skills that the company must have to operate.

* **3.3.7 Version & Release :-**

Versioning is the creation and management of multiple releases of a product, all of which have the same general function but are improved, upgraded or customized. ***Version control*** is the practice of ensuring collaborative data sharing and editing among users of systems that employ different versions of a product. The terms "versioning" and "version control" are sometimes used interchangeably even though their technical meanings are different.

In software versioning, subsequent releases of a specific product receive numerical identifiers consisting of two or three numbers separated by periods. The ***first number, called the major number***, is increased when there are significant improvements or changes in functionality. The ***second number, called the minor number, is incremented when there are minor feature changes or significant fixes***. The ***third number, if it exists, is called the revision number.*** It is added or increased when minor bugs are eliminated.

Version control combines procedures and tools to manage different versions of configuration objects that are created during software product development.

To control versions, you can use Version Control Register. In Version Control Register, you enter the details of components, such as component identification numbers, their versions, and dates of validity. It is advisable to release a baseline after a version is released. Baseline is a specification or a product that is formally reviewed and agreed upon. This serves as the basis for further development. Baseline can be changed only through formal change control procedures. A baseline consists of a set of SCIs that are logically related to each other. Baselines are established when subsequent changes to the SCIs need to be controlled. Version control is essential so that everybody uses only the latest version. Any kind of version mismatch might result in rework.

SCM differentiates between baselines and interim versions. A baseline is a tested and certified version of a system. Baselines can be assigned version numbers such as 1.0, 2.0, 3.0, and so on. A baseline usually undergoes intensive testing. Interim versions, on the other hand, have version numbers, such as 1.1 or 1.2. The interim version is a temporary version. Interim versions have a short life and survive only during bug fixing, testing, or debugging.

Version control is used to manage multiple versions of computer files and programs. A version control system, or VCS (Version Control System ), provides two primary data management capabilities. It allows users to 1) lock files so they can only be edited by one person at a time, and 2) track changes to files.

If you are the only person editing a document, there is no need to lock a file for editing. However, if a team of developers is working on a project, it is important that no two people are editing the same file at the same time. When this happens, it is possible for one person to accidentally overwrite the changes made by someone else. For this reason, version control allows users to "check out" files for editing. When a file has been checked out from a shared file server, it cannot be edited by other users. When the person finishes editing the file, he can save the changes and "check in" the file so that other users can edit the file.

Version control also allows users to track changes to files. This type of version control is often used in software development and is also known as "source control" or "revision control." Popular version control systems like Subversion and CVS allow developers to save incremental versions of programs and source code files during the development process. This provides the capability to rollback to an earlier version of the program if necessary. For example, if bugs are found in a new version of a software program, the developer can review the previous version when debugging the code.

Version control software requires that all files are saved in a central location. This location is called the repository and contains all previous and current versions of files managed by the VCS. Whenever a new file is created or a current file is updated, the changes are "committed" to the repository, so the latest version is available to all users.

A good VCS (Version Control System) does the following:

* **Backup and Restore**. Files are saved as they are edited, and you can jump to any moment in time.
* **Synchronization.** Lets people share files and stay up-to-date with the latest version.
* **Track Changes**. As files are updated, you can leave messages explaining why the change happened (stored in the VCS, not the file). This makes it easy to see how a file is evolving over time, and why.
* **Track Ownership.** A VCS tags every change with the name of the person who made it. Helpful for blame storming giving credit.

**Release management** is the process of managing, planning, scheduling and controlling a software build through different stages and environments; it includes testing and deploying [software releases](https://en.wikipedia.org/wiki/Software_release).

Release management is a relatively new but rapidly growing discipline within [software engineering](https://en.wikipedia.org/wiki/Software_engineering). As [software](https://en.wikipedia.org/wiki/Software) systems, [software development processes](https://en.wikipedia.org/wiki/Software_development_process), and resources become more distributed, they invariably become more specialized and complex. Furthermore, software products (especially [web applications](https://en.wikipedia.org/wiki/Web_application)) are typically in an ongoing cycle of development, [testing](https://en.wikipedia.org/wiki/Software_testing), and [release](https://en.wikipedia.org/wiki/Software_release), often running on evolving platforms with growing complexity. Such systems require dedicated resources to oversee the integration and flow of development, testing, deployment, and support.

Organizations that have adopted [agile software development](https://en.wikipedia.org/wiki/Agile_software_development) are seeing much higher quantities of releases. With the increasing popularity of agile development a new approach to software releases known as [Continuous delivery](https://en.wikipedia.org/wiki/Continuous_delivery) is starting to influence how software transitions from development to a release. One goal of Continuous Delivery. The movement of the application from a “build” through different environments to production as a “release” is part of the Continuous Delivery pipeline. Release managers are beginning to utilize tools such as [application release automation](https://en.wikipedia.org/wiki/Application_release_automation) and [continuous integration](https://en.wikipedia.org/wiki/Continuous_integration) tools to help advance the process of Continuous Delivery and incorporate a culture of DevOps by automating a task so that it can be done more quickly, reliably, and is repeatable. More software releases have led to increased reliance on release management and automation tools to execute these complex application release processes.

**What is Agile release planning?**

Agile release planning is a product management method where you plan incremental releases of a product. It differs from traditional software planning where you focus on major releases.

In Agile release planning, you prepare for staged releases and then break those down into several different sprints or iterations.

Depending on your team structure and the size of the [project](https://monday.com/blog/project-management/so-what-is-a-project-everything-you-should-know/), you may even have several sprints running at the same time.

A sprint ends with a new product increment, but that may not mean a product release happens.

Think about writing a book — each sprint should end with a new draft. But, just because you have a finished draft for your client (editor, publisher, etc.) to review doesn’t mean you release it to the public.

Release planning helps you plan which product increments (versions) get released to the market and when.

This approach helps your team adapt to the unpredictable nature of software development.

And it’s an integral part of the [Agile SDLC](https://monday.com/blog/project-management/agile-sdlc/) (Software Development Life Cycle).

You might create versions 1.1, 1.12, 1.14, 1.2, and so on, each one incorporating the new features and improvements of the last. But you’ll only release the big ones (1.1 and 1.2, etc.)

Under [15%](https://assets.kpmg/content/dam/kpmg/be/pdf/2019/11/agile-transformation.pdf) of leaders think their employees are ready to work in an Agile environment and self-organize. So, release planning is a vital structural tool, especially for teams new to Agile.

It gives the higher-ups peace of mind that there’s a structure and plan beyond just the next sprint, and helps the individual Agile teams stay on track.

The process involved in version and release management are concerned with identifying and keeping track of the versions of a system. Versions managers devise procedures to ensure that versions of a system may be retrieved when required and are not accidentally changed by the development team. For products, version managers work with marketing staff and for custom systems with customers, to plan when new releases of a system should be created not distributed for deployment.

A system instance is an instance of a system which can be different from other instances in some way. There is a chance in which versions of the system may have different functionality, enhanced performance or repaired software faults. Some versions may be functionally equivalent but designed for different hardware or software configuration. Versions with only small differences are sometimes called **variants.**

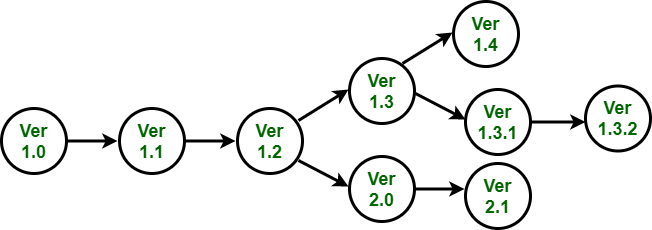
A system release may be a version that’s distributed to customers. Each system release should either include new functionality or should be intended for a special hardware platform. There are normally many more versions of a system than release. Versions are created with an organization for internal development or testing and are not intended for release to customers.

**Version Identification :**

To create a specific version of a system, you’ve got to specify the versions of the system components that ought to be included in it. In a large software system, there are hundreds to software components, each of which may exist in several different versions.

* **Version Numbering :**

In version numbering scheme, a version number is added to the components or system name. If the first version is called 1.0, subsequent versions are 1.1, 1.2 and so on. At some stage, a new release is created (release 2.0) and process start again at version 2.1.The scheme is linear, based on the assumption that system versions are created in sequence. Most version management tools such as RCS and CVS support this approach to version identification.



* **Attribute Based Identification :**

If each version is identified by a unique set of attributes, it is easy to add new versions, that are derived from any of existing versions. These are identified using unique set of attribute values. They share most of those values with their parent version so relationships between versions are maintained. You can retrieve specific versions by specifying attribute values required. Functions on attributes support queries like ‘the last created version’ or ‘the version created between given dates’. Attribute-based identification may be implemented directly by version management system, with component attributes maintained in a system database. Alternatively, attribute identification system may be built as a layer on top of a hidden version-numbering scheme.

* **Change Oriented Identification :**

Each component is known as as in attribute-based identification but is additionally related to one or more change requests. That is, it is assumed that each version of component has been created in response to one or more change requests. Component version is identified by set of change requests that apply to components.

There must therefore be an unambiguous way to identify each component version to ensure that the right components are included in the system. Three basic techniques are used for components version identification

**What is the purpose of Agile release planning?**

The purpose of release planning within the [Agile methodology](https://monday.com/blog/project-management/agile-project-management/) is to ensure the product is always moving in the right direction and that logical releases are frequently happening.

A release plan outlines immediate future releases but doesn’t try to plan for years to come.

It goes into more detail than a product roadmap (high-level scope and timeline).

But an Agile release plan doesn’t outline the work in each release. Instead, it batches iterations or sprints together into releases.

Old-fashioned executives fear going Agile means each version is a random collection of features. A release plan ensures that you create a coherent version of your product every time.

It’s a great tool for combining changes that will have a significant impact on the user experience.

Overall, implementing Agile can help you cut your time to market for a new product by up to [70%](https://www.mckinsey.com/business-functions/organization/our-insights/enterprise-agility-buzz-or-business-impact).

**When to do release planning in Scrum?**

Release planning comes after you’ve outlined your product vision and roadmap.

Planning and combining sprints into larger releases is often a good idea. Especially if you have a lot of major items in your product backlog.

Most people don’t like change. Users take time to adapt to a new interface. So batching changes to the UX is a must.

Scrum release planning isn’t part of the original processes or meetings in the [Scrum Guide](https://monday.com/blog/project-management/scrum-guide/).

Since the focus of Scrum is on shorter sprints, some teams work without release planning at all.

Instead, they just release the product increment. That keeps the focus on speed and adapting to the stakeholder needs at any moment.

**Who is responsible for release planning in Scrum?**

In Scrum, the release planning process isn’t reserved for product managers. Your product owner and Scrum team must also participate.

The team can also be solely responsible for the plan depending on your company structure.

The Scrum team is often better connected to the current state of the software and all the different stakeholders, so they tend to lead the decision-making.

The release cycle should always be driven by those closest to the project. If not, you risk creating a roadblock instead of a roadmap. And nobody wants that.

**How to implement Agile release planning**

The steps below assume that you’re already familiar with [Agile principles](https://monday.com/blog/project-management/agile-principles/), and have implemented an [Agile workflow](https://monday.com/blog/project-management/agile-workflow/) in your company.

If that’s the case, getting started with Agile release planning is quite simple.

**1) Evaluate your product vision and roadmap and single out outcomes**

Your overall product vision and the product roadmap should guide the whole process. Focus on outcomes.

Which outcomes are the most important in the short to medium term?

Discuss this question with your main [stakeholders](https://monday.com/blog/project-management/stakeholder-analysis-ultimate-guide/) and customers.

**2) Expand and prioritize your product backlog based on these outcomes**

If you use Scrum, meet with your Agile team for a product backlog refinement meeting.

Break down your desired outcomes into specific user stories and add them to your backlog.

Prioritize the backlog, based on the desired outcomes, starting with an MVP (minimum viable product).

You don’t want to work on irrelevant things, and you can’t release something that’s not viable (can function on it’s own).

For Agile frameworks that don’t use a product backlog, like Feature-Driven Development, review the equivalent item. In the case of FDD, it’s the features list.

**3) Set a clear release goal that combines logical product backlog items**

Create a release goal that combines different user stories into a meaningful change in user experience.

For example, the ability of a user to access different analytics reports on the mobile app.

If you’re creating multiple reports, it makes sense to release them together, rather than piece-meal.

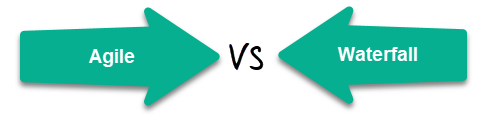
**4) Break your releases down into multiple sprints or iterations**

Once you’ve got your release goal and targeted user stories, it’s time to start planning your actual sprints.

* **3.7 Agile Project Management Vs. Traditional Project Management :-**

**Waterfall Model methodology** which is also known as Liner Sequential Life Cycle Model. Waterfall Model followed in the sequential order, and so project development team only moves to next phase of development or testing if the previous step completed successfully.

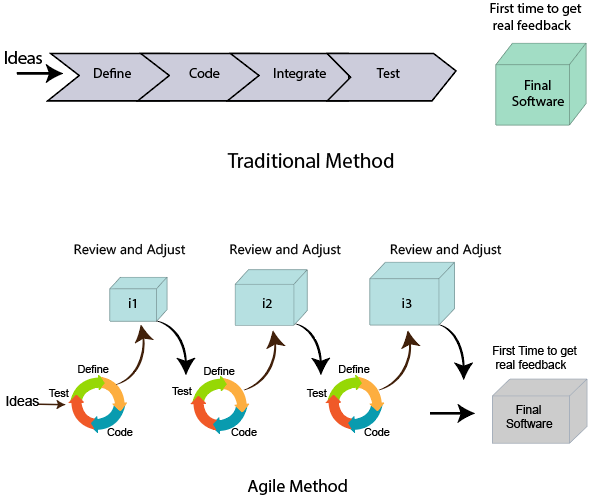
**Agile methodology** is a practice that helps continuous iteration of development and testing in the software development process. In this model, development and testing activities are concurrent, unlike the Waterfall model. This process allows more communication between customers, developers, managers, and testers.

[](https://www.guru99.com/images/1/111517_1049_WaterfallVs1.png)

**Waterfall vs Agile Key Difference**

* Waterfall is a Liner Sequential Life Cycle Model whereas Agile is a continuous iteration of development and testing in the software development process.
* In Agile vs Waterfall difference, the Agile methodology is known for its flexibility whereas Waterfall is a structured software development methodology.
* Comparing the Waterfall methodology vs Agile which follows an incremental approach whereas the Waterfall is a sequential design process.
* Agile performs testing concurrently with software development whereas in Waterfall methodology testing comes after the “Build” phase.
* Agile allows changes in project development requirement whereas Waterfall has no scope of changing the requirements once the project development starts.
* **Difference between Agile and Waterfall Model:**

|  |  |
| --- | --- |
| **Agile** | **Waterfall** |
| It separates the project development lifecycle into sprints. | Software development process is divided into distinct phases. |
| It follows an incremental approach | Waterfall methodology is a sequential design process. |
| Agile methodology is known for its flexibility. | Waterfall is a structured software development methodology so most times it can be quite rigid. |
| Agile can be considered as a collection of many different projects. | Software development will be completed as one single project. |
| Agile is quite a flexible method which allows changes to be made in the project development requirements even if the initial planning has been completed. | There is no scope of changing the requirements once the project development starts. |
| Agile methodology, follow an iterative development approach because of this planning, development, prototyping and other software development phases may appear more than once. | All the project development phases like designing, development, testing, etc. are completed once in the Waterfall model. |
| Test plan is reviewed after each sprint | The test plan is rarely discussed during the test phase. |
| Agile development is a process in which the requirements are expected to change and evolve. | The method is ideal for projects which have definite requirements and changes not at all expected. |
| In Agile methodology, testing is performed concurrently with software development. | In this methodology, the "Testing" phase comes after the "Build" phase |
| Agile introduces a product mindset where the software product satisfies needs of its end customers and changes itself as per the customer's demands. | This model shows a project mindset and places its focus completely on accomplishing the project. |
| Agile methodology works exceptionally well with Time & Materials or non-fixed funding. It may increase stress in fixed-price scenarios. | Reduces risk in the firm fixed price contracts by getting risk agreement at the beginning of the process. |
| Prefers small but dedicated teams with a high degree of coordination and synchronization. | Team coordination/synchronization is very limited. |
| Products owner with team prepares requirements just about every day during a project. | Business analysis prepares requirements before the beginning of the project. |
| Test team can take part in the requirements change without problems. | It is difficult for the test to initiate any change in requirements. |
| Description of project details can be altered anytime during the SDLC process. | Detail description needs to implement waterfall software development approach. |
| The Agile Team members are interchangeable, as a result, they work faster. There is also no need for project managers because the projects are managed by the entire team | In the waterfall method, the process is always straightforward so, project manager plays an essential role during every stage of SDLC. |



Compression between the Agile methodology and Waterfall model:

|  |  |
| --- | --- |
| **Agile methodology** | **Waterfall model** |
| It follows the incremental approach. | It is a sequential design process. |
| It divides the project development lifecycle into a sprint. | The software development process is divided into distinct phases. |
| Agile methodology is a flexible methodology. | The Waterfall is a structured software development methodology. |
| Agile is the collection of many different projects. | It is completed as one single project. |
| The test plan is reviewed after each sprint | Test plan is reviewed after complete development. |
| Testing team can take part in the requirements change phase without problems. | It is difficult for the test to initiate any change in needs. |

**Advantages of Waterfall Model:**

* It is one the easiest model to manage. Because of its nature, each phase has specific deliverables and a review process.
* It works well for smaller size projects where requirements are easily understandable.
* Faster delivery of the project
* Process and results are well documented.
* Easily adaptable method for shifting teams
* This project management methodology is beneficial to manage dependencies.

**Advantages of the Agile Model:**

* It is focused client process. So, it makes sure that the client is continuously involved during every stage.
* Agile teams are extremely motivated and self-organized so it likely to provide a better result from the development projects.
* Agile software development method assures that quality of the development is maintained
* The process is completely based on the incremental progress. Therefore, the client and team know exactly what is complete and what is not. This reduces risk in the development process.

**Limitations of Waterfall Model:**

* It is not an ideal model for a large size project
* If the requirement is not clear at the beginning, it is a less effective method.
* Very difficult to move back to makes changes in the previous phases.
* The testing process starts once development is over. Hence, it has high chances of bugs to be found later in development where they are expensive to fix.

**Limitations of Agile Model**

* It is not useful method for small development projects.
* It requires an expert to take important decisions in the meeting.
* Cost of implementing an agile method is little more compared to other development methodologies.
* The project can easily go off track if the project manager is not clear what outcome he/she wants.