SCRUM

Scrum is a **management framework** for **incremental product development** using one or more **cross-functional, self-organizing teams** of about **seven (+/-2)** people each
Scrum uses **fixed-length iterations**, called **Sprints**, which are typically **one to four weeks** long. Scrum teams attempt to build a **potentially shippable (properly tested) product increment** every iteration

- *A Potentially Shippable Product is the sum of the Product Backlog Items delivered each Sprint*
- *A potentially shippable product is one that has been designed, developed and tested and is therefore ready for distribution to anyone in the company for review or even to any external stakeholder*

Whether the product is truly shippable will depend on the current Definition of Done. If there is a perfect definition of done, then the product can be shipped each Sprint. If the Definition of Done is weak, then there will still be work to be done, leading to less transparency, less flexibility and increased development risk.
The greatest potential benefit of Scrum is for complex work involving knowledge creation and collaboration, such as new product development
Scrum's relentless reality checks expose dysfunctional constraints in individuals, teams, and organizations
A key principle of Scrum is its recognition that during product development, the customers can change their minds about what they want and need (often called requirements volatility, and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an evidence-based empirical approach—accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly, to respond to emerging requirements and to adapt to evolving technologies and changes in market conditions.

PRINCIPLES OF AGILE (ICWR):

- Individuals and Interactions over process and tools
- Working Software over a comprehensive documentation
- Customer Collaboration over Contract Negotiations
- Responding to change over following a plan

3 PILLARS OF SCRUM:

- Transparency
- Adaptation
- Inspection

SCRUM ROLES:

Product Owner:

- Single person responsible for **maximizing ROI/ business value** on the development effort
- Responsible for the **product vision**
- **Constantly re-prioritizing** the **Product Backlog**, adjusting any long term expectations such as **Release Plans**
- **Final arbiter** of requirements questions
- Accepts or rejects each product increment
- Decides **whether to ship**
- Decides whether to **continue development**
- Considers **stakeholders' interests**
- May contribute as a team member
- Has a **leadership role**

Scrum Development Team:

- Cross-functional
- Self-organizing/ self-managing, without externally assigned roles
- The team negotiates commitment with the PO, one Sprint at a time
- Has autonomy regarding how to reach commitments
- Highly collaborative
- Most successful when located in one team room, particularly for the first few Sprints
- Most successful with long-term, full-time membership. Scrum moves work to a flexible learning team & avoids moving people or splitting them between teams
- 7 +/- 2 members, but recently there have been teams of 3 members that have been successful. Thus the number is 3 to 9 members
- Has a leadership role

Scrum Master:

- Facilitates Scrum process
- Helps resolve impediments
  - An impediment is anything that slows down or diminishes the pace of the Team
  - A blocker is anything that stops the delivery of the product
- Shields the team from external interference & distractions to keep it in group flow (aka the zone)
- Creates an environment conducive to team self-organization
- Captures empirical data to adjust forecasts
- Enforces timeboxes
  - A timebox is a previously agreed period of time during which a person or a team work steadily towards completion of some goal. Rather than allow work to continue until the goal is reached, and evaluating the time taken, the time-box approach consists of stopping work when the time limit is reached and evaluating what was accomplished
- Keeps Scrum artifacts visible

- Promotes improved engineering practices
- Has no management authority over the team
- Has a leadership role

Scrum Guidance Body:

The Scrum Guidance Body (SGB) is an optional role. It generally consists of a group of documents and/or a group of experts who are typically involved with defining objectives related to quality, government regulations, security, and other key organizational parameters. These objectives guide the work carried out by the Product Owner, Scrum Master, and Scrum Team. The Scrum Guidance Body also helps capture the best practices that should be used across all Scrum projects in the organization.

The Scrum Guidance Body does not make decisions related to the project. Instead, it acts as a consulting or guidance structure for all the hierarchy levels in the project organization—the portfolio, program, and project. Scrum Teams have the option of asking the Scrum Guidance Body for advice as required.

SCRUM CEREMONIES/ MEETINGS:

All meetings are facilitated by the Scrum Master who has no authority at these meetings

Sprint Planning Meeting:

- Held by Product Owner & team at the **beginning of every sprint**
- Max time for SPM is **4 hours for a 2-week Sprint** or 8 hours for a 4-week Sprint
- Which Product Backlog Items (PBIs) will they convert to working product--> **negotiation**
  - In Scrum, a **product backlog item ("PBI", "backlog item", or "item")** is a unit of work small enough to be completed by a team in one Sprint iteration. Backlog items are decomposed into one or more tasks
  - If a **PBI** would take more than a quarter of a two-week Sprint, it's probably too big
  - In Scrum, a **sprint task (or task)** is a unit of work generally between four and sixteen hours. Team members volunteer for tasks. They update the estimated number of hours remaining on a daily basis, influencing the Sprint Burndown Chart. Tasks are contained by backlog items. Scrum literature encourages splitting a task into several if the estimate exceeds twelve hours
- **PO declares which items are most important** for the business
- **The team responsible for selecting the amount of work** they can do during the sprint, without leaving any *technical debt.* Team "pulls" work from the Product Backlog into the Sprint Backlog
  - **Technical Debt** includes those internal things that you choose not to do now, but which will impede future development if left undone. This includes deferred refactoring
  - **Technical Debt** is the difference between what was promised and what was actually delivered
    - A common reason for bringing technical debt into a code base comes from the business stakeholders. Assuming they have a reasonable understanding of the consequences, the business might consider getting something released sooner is of more value than avoiding technical debt. They should understand the "interest" payment that will be incurred if they insist on this path! In many cases, businesses stakeholders simply don't understand the ramifications of what they are asking for, nor do they fully grasp the concept of. They make decisions solely on immediate business pressures rather than taking a more long-term view.
    - Technical debt also comes in the form of poorly constructed, inflexible software. This may come about when functions or interfaces are hard-coded, and as such, are difficult to change.
    - Lack of documentation is another reason for technical debt, both in the code itself and in the external documentation. When documentation is poor, new team members who want to modify the code in the future have a hard time coming up to speed on the code which that slows development.
  - Besides the use of practices such as test-driven development and continuous integration, it would be preferable to set apart a certain amount of time as "slack" time during each iteration, to service technical debt
    - Test Driven Development is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfil that test and refactoring
    - Continuous integration (CI) is a software engineering practice in which isolated changes are immediately tested and reported on when they are added to a larger code base. The goal of CI is to provide rapid feedback so that if a defect is introduced into the code base, it can be identified and corrected as soon as possible
  - Adding members to the team for some time to payback the technical debt
  - Managing Technical Debt
    - Technical Backlog
      - The technical backlog is an established best practice to define purely technical work packages. Tasks for this purpose are created in a task tracker or requirements

management tool. Each task has a brief description of the technical change to be made, why this technical change is important for the project and in which part of the code the technical change has to be performed. As with any other task, we need an estimate of how long it takes to develop a good enough solution
    - Include cost for Technical Debt in requirements estimation
    - Buffer-tasks for refactoring
        - The team creates one buffer-task per release with e.g. 10% of the available time. Team members can record the time on that task for not yet scheduled refactoring. So it is used for yet unknown problems that might appear in the future
    - Cleanup-releases
        - Some teams do a purely technical release to improve the codebase from time to time. This approach is only useful if a list with the really necessary refactorings already exists. Otherwise, the team risks wasting time on unimportant refactorings. This approach must also be supported by the business side because it might delay new features. Of course, this requires that business people understand Technical Debt

- Towards the end of the SPM, the **team breaks the selected items into an initial list of Sprint Tasks**

Daily Scrum Meeting & Sprint Execution:
- **Every day; same time same place**
- **15 min to report** to each other
    - What did I do yesterday?
    - What am I going to do today?
    - Any impediments?
- Standing up helps keep it short
- The team usually maintains:
    - Sprint Task List
    - Sprint Burndown Chart
    - Impediment List
        - Impediments caused by issues beyond the team's control are *organizational impediments*
- **Common to discover additional tasks** necessary to achieve the Sprint goals
- Useful for Product Owner to attend the Daily Scrum
- Intended to **disrupt old habits of working separately**

Sprint Review Meeting:
- Sprint Review Meeting is held **after a Sprint** to **demonstrate a working product increment** to the **PO** and **everyone else** who is interested
- Should feature a **live demonstration**, not a report
- After the demo, PO reviews the commitments made at the SPM & declares **which items are *done***
    - To be "Done" at a minimum, it should include:
        - Complete Design and Development
        - Feature Tested and Defects addressed
        - Meets Acceptance Criteria and Product Owner has signed off
    - Ideally, it would also include:
        - Integration Tested and Defects addressed
        - Regression Tested and Defects addressed
- Generally **timeboxed for 2-4 hours** for **2-4 week Sprints**
- ScrumMaster helps the PO & stakeholders convert their feedback to new Product Backlog Items for prioritization by PO
- Appropriate meeting for the external stakeholders (even end users) to attend

- Opportunity to inspect & adapt the product as it emerges & iteratively refine everyone's understanding of the requirements

Sprint Retrospective Meeting:

- Each Sprint ends with a Sprint Retrospective Meeting
- The sprint review looks at what the team is building, whereas the retrospective looks at how they are building it
- Team reflects on its own process, inspect their behaviour, and takes action to adapt to future sprints
- Scrum Masters should use a variety of *techniques to facilitate retrospectives*, including *Silent Writing, Safety Check, and Happi*ness *histograms*

- An in-depth retrospective requires an environment of psychological *safety* not found in most organizations
- 3 types of impediments are:
  - Scrum Master Owned Impediments:
    - "The team has been trying to have a meeting with the tech lead from another team for 4 days now. He is hard to catch up with and his calendar is nearly full." As a scrum master, I would coordinate/facilitate a face to face meeting so that the team is not consumed with administration
  - Team Owned Impediments:
    - "We need to test this feature but we don't have enough testers." As a Scrum master, I would guide the team to solving this themselves by "swarming" so that everyone becomes a tester so that we could get items to done
  - Organizational Impediments:
    - An example of an organizational impediment would be something like, "I have a person on my team who is constantly being asked by other managers to take on other duties. As a result, he is never able to properly commit to and focus on the work for this project " So I would work with managers to remove this impediment either by:
      - Having the person replaced with someone who had fewer distractions
      - Having the other managers take those distracting items off his list of things to do.

- Often expose organizational impediments
- Once a team has resolved the impediments within its immediate influence, the Scrum Master should work to expand that influence, chipping away at the organizational impediments
- Key elements of the Sprint Retrospective:
  - Process improvements are made at the end of every sprint. This ensures that the project team is always improving the way it works
  - The retrospective is a collaborative process among all members, including the team, the product owner, and the Scrum Master
  - All team members identify what went well and what could be improved
  - The team members discuss the process that they are following and give any suggestions for improvement
  - The team members discuss any other ideas that could improve their productivity
  - The Scrum Master prioritizes actions and lessons learned based on team direction
  - The retrospective supports team formation and bonding, particularly as any areas of conflict can be identified and dealt with
  - The retrospective helps build the team's sense of ownership and its self-management

Backlog Refinement Meeting:

- Most Product Backlog Items (PBIs) initially need refinement as they are too large or poorly understood
- In BRM, team estimates the amount of effort they would need to complete items in the Product Backlog & provide other technical information to help the PO prioritize them

- Large value items are split & clarified, considering both business & technical concerns
- BRM is also called as "Backlog Grooming", "Backlog Maintenance" or "Story Time"

- Sometimes, a part of the team, along with the PO and other stakeholders compose & split Product Backlog Items before involving the entire team in estimation
- A skilled Scrum Master will help the team identify thin vertical slices of work that still have business value while promoting a rigorous definition of "done" that includes proper testing and refactoring
- Common to write Product Backlog Items in User Story form
- Oversized PBIs are called Epics
- Since most customers don't use most features of most products, it's wise to split epics to deliver the most valuable stories first
- Best done a couple of work days before the next Sprint Planning Meeting
- Usually, no commitments are made in this meeting. That discussion shelved for the Sprint Planning Meeting
- This meeting can go ahead without team members who are really busy, so it does not affect their work

SCRUM ARTIFACTS:

Product Backlog:

- A Product Backlog, in its simplest form, is **merely a list of items** to work on. Having well-established rules about how work is added, removed and ordered helps the whole team make better decisions about how to change the product.
- *The Product Owner prioritizes* which *Product Backlog Items* are most needed. The team then chooses which items they can complete in the coming Sprint. On the Scrum board, the *team moves items from the Product Backlog to the Sprint Backlog*, which is the list of items they will build. Conceptually, it is ideal for the team to only select what they think they can accomplish from the top of the list, but it is not unusual to see in practice that teams are able to take lower-priority items from the list along with the top ones selected. This normally happens because there is time left within the Sprint to accommodate more work. Items at the top of the backlog, the items to work on first, should be broken down into stories that are suitable for the Development Team to work on. *The further down the backlog goes, the less refined the items should be*. As Schwaber and Beedle put it "The lower the priority, the less detail until you can barely make out the backlog item.

    - *Forced ranked list* of desired functionality
    - Visible to all stakeholders
    - *Any stakeholder can add* items to the list with the consent of the product owner.
    - Constantly re-prioritized by the PO
    - Items at top are more granular than items at bottom
    - Maintained during the Backlog Refinement Meeting

Product Backlog Item:

- Specifies *"what"* more than a "how" of a customer-centric feature
- Often written in *User Story form*
- Has a *product-wise definition of done* to prevent *technical debt*
- May have item-specific acceptance criteria
- Effort is estimated by the team, ideally in relative units (eg. Story points)

- Effort is roughly 2-3 people, 2-3 days, or smaller for advanced teams

Sprint Backlog:

- Consists of *committed PBIs* negotiated between the team and the Product Owner during the Sprint Planning Meeting
- Scope commitment is fixed during Sprint Execution
- Initial tasks are identified by the team during Sprint Planning Meeting

- *Team will discover additional tasks needed to meet the fixed scope commitment during Sprint execution*
- Visible to the team
- *Referenced during the Daily Scrum Meeting*

The *Sprint Backlog* is the list of work the Development Team must address during the next Sprint. The list is derived by the Scrum Team progressively selecting Product Backlog Items in priority order from the top of the Product Backlog until they feel they have enough work to fill the Sprint. The Development Team should keep in mind it's past performance assessing its capacity for the new Sprint, and use this as a guideline of how much 'effort' they can complete.
The Product Backlog Items may be broken down into tasks by the Development Team. Tasks on the Sprint Backlog are never assigned; rather, tasks are signed up for by the team members as needed according to the set priority and the Development Team member skills. This promotes self-organization of the Development Team and developer buy-in.
The Sprint Backlog is the property of the Development Team, and all included estimates are provided by the Development Team. Often an accompanying *task board* is used to see and change the state of the tasks of the current Sprint, like 'to do', 'in progress' and 'done'.
Once a Sprint Backlog is committed, no additional work can be added to the Sprint Backlog except by the team. Once a Sprint has been delivered, the Product Backlog is analysed and reprioritized if necessary, and the next set of functionality is selected for the next Sprint.

Sprint Task:

- Specifies "how" to achieve the PBI's rather than "what"
- Requires one day or less of work
- Remaining effort is re-estimated daily, typically in hours
- During Sprint Execution, a point person may volunteer to be primarily responsible for a task
- Owned by the entire team; collaboration is expected

Sprint Burndown Chart:

- Indicates *total remaining team task hours* within one Sprint
- *Re-estimated daily* thus may go up before going down
- Intended to facilitate team self-organization
- Fancy variations, such as itemizing by point person or adding trend lines, tend to reduce effectiveness at encouraging collaboration
- Seemed like a good idea in the early days of Scrum, but in practice has often been misused as a management report, inviting intervention. The Scrum Master should discontinue use of this chart if it becomes an impediment to team self-organization

The Sprint burn-down chart is a public displayed chart showing remaining work in the Sprint Backlog. Updated every day, it gives a simple view of the Sprint progress. It also provides quick visualizations for reference. The horizontal axis of the Sprint burndown chart shows the days in a Sprint, while the vertical axis shows the amount of work remaining each day (typically representing an estimate of hours of work remaining).
During Sprint Planning the ideal burndown chart is plotted. Then, during the Sprint, each member picks up tasks from the Sprint Backlog and works on them. At the end of the day, they update the remaining hours for tasks to be completed. In such a way, the actual burndown chart is updated day by day.

Product/ Release Burndown Chart:

- Tracks the remaining Product Backlog effort from one Sprint to the next

- May use relative units such as Story Points for Y axis
- Depicts historical trends to adjust forecasts

Release Burn Up Chart:

- The release burn-up chart is a way for the team to provide visibility and track progress toward a release. *Updated at the end of each Sprint, it shows progress toward delivering a forecast scope*. The horizontal axis of the release burn-up chart shows the Sprints in a release, while the vertical axis shows the amount of work completed at the end of each Sprint (typically representing cumulative story points of work completed). Progress is plotted as a line that grows up to meet a horizontal line that represents the forecast scope; often shown with a forecast, based on progress to date, that indicates how much scope might be completed by a given release date or how many sprints it will take to complete the given scope.
- The release burn-up chart makes it easy to see how much work has been completed, how much work has been added or removed (if the horizontal scope line moves), and how much work is left to be done.

Sprint Goal:

A sprint goal is a short, one- or two-sentence, description of what the team plans to achieve during the sprint. It is written collaboratively by the team and the product owner. The following are example sprint goals on an eCommerce application:

- Implement basic shopping cart functionality including add, remove, and update quantities.
- Develop the checkout process: pay for an order, pick shipping, order gift wrapping, etc.

The sprint goal can be used for quick reporting to those outside the sprint. There are always stakeholders who want to know what the team is working on, but who do not need to hear about each product backlog item (user story) in detail.

SCRUM ADVANTAGES, DISADVANTAGES and LIFECYCLE

Advantages of SCRUM:

- Helps company save time & money
- Enables project where the business requirements documentation is hard to quantify to be successfully developed
- Fast moving, cutting-edge developments can be quickly coded & tested using this method, as a mistake can be easily rectified
- The tightly controlled method which insists on frequent updating of progress in work through regular meetings. Thus, there is clear visibility of project development
- Like any other agile methodology, this is also iterative in nature. It requires continuous feedback from the user
- Due to short sprints and constant feedback, it becomes easier to cope with the changes
- Daily meetings make it possible to measure individual productivity. This leads to the improvement in the productivity of each of the team members
- Issues are identified well in advance through the daily meetings & hence can be resolved speedily
- It is easier to deliver a quality product in a scheduled time
- Agile Scrum can work with any technology/programming language but is particularly useful for fast moving web 2.0 or new media projects
- The overhead cost in terms of process & management is minimal thus leading to a quicker, cheaper result

Additional read - http://www.dummies.com/careers/project-management/10-key-benefits-of-scrum/

Disadvantages of SCRUM:

- Agile Scrum is one of the leading causes of scope creep because unless there is definite end date, the project management stakeholders will be tempted to keep demanding new functionality is delivered
- If a task is not well defined, estimating project costs and time will not be accurate. In such a case, the task can be spread over several sprints
- If the team members are not committed, the project will either never complete or fail
- It is good for small, fast moving projects as it works well only with small team
- This methodology needs experienced team members only. If the team consists of people who are novices, the project cannot be completed in time
- Scrum works well when the Scrum Master trusts the team they are managing. If they practice too strict control over the team members, it can be extremely frustrating for them, leading to demoralization and the failure of the project
- If any of the team members leave during a development it can have a huge inverse effect on the project development
- Project quality management is hard to implement and quantify unless the test team are able to conduct regression testing after each sprint

SCRUM Life Cycle:

- Sprint Backlog
- Product Backlog
- 7-30 days Sprint
- Daily Scrum Meetings- 15 min
  - What did you do today?
  - What will you do today?
  - What are the obstacles/bottlenecks?
- Scrum Master & other pig roles talk
- At the end of Sprint, an output or a potentially shippable product
- Sprint Planning Meeting- 4 hours at the beginning of every Sprint Cycle
- Sprint Review Meeting- 4 hours at the end of every Sprint Cycle
- Sprint Retrospective Meeting- 3 hours (Lessons learnt)
  - What went well during the Sprint?
  - What can be improved in future sprints?

ESTIMATION TECHNIQUES

PLANNING POKER:

Planning Poker is an agile estimating and planning technique that is consensus based. To start a poker planning session, the product owner or customer reads an agile user story or describes a feature to the estimators.

Each estimator is holding a deck of Planning Poker cards with values like 0, 1, 2, 3, 5, 8, 13, 20, 40 and 100, which is the sequence we recommend. The values represent the number of story points, ideal days, or other units in which the team estimates.

The estimators discuss the feature, asking questions of the product owner as needed. 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 their estimates. The high and low estimators should especially share their reasons. After further discussion, each estimator reselects an estimate card, and all cards are again revealed at the same time.

The poker planning process is repeated until consensus is achieved or until the estimators decide that agile estimating and planning of a particular item needs to be deferred until additional information can be acquired.

When should we engage in Planning Poker?

Most teams will hold a Planning Poker session shortly after an initial product backlog is written. This session (which may be spread over multiple days) is used to create initial estimates useful in scoping or sizing the project.

Because product backlog items (usually in the form of user stories) will continue to be added throughout the project, most teams will find it helpful to conduct subsequent agile estimating and planning sessions once per iteration. Usually, this is done a few days before the end of the iteration and immediately following a daily standup since the whole team is together at that time anyway.

T-SHIRT SIZE:

Occasionally team members align the story points with the hours of effort, which can create confusion. To avoid this, it may be more effective to switch to a numeric estimation technique.

Teams who don't enjoy playing with numbers that much use T-shirt size as another popular technique. So when the teams tend to over analyze the numbers, T-shirt sizes come in handy, as T-shirt sizes are easier to relate to. With T-shirt sizing, the team is asked to estimate whether they think a story is extra- small, small, medium, large, extra-large, or double extra-large. By removing the implied precision of a numerical score, the team is free to think in a more abstract way about the effort involved in a story.

Some teams even adopt creative approaches, such as using dog breeds, to estimate stories. For example, "That story's clearly a Chihuahua, but the other one is a Great Dane." Engaging the fun, creative side of the team while they're estimating technical stories can be effective at getting them out of their analytical thought processes and into a more flexible, relative mindset.

There are some practical issues to consider when adopting T-shirt sizing for story estimation. For one, non-numerical scales are generally less granular. Although that can speed the voting process by reducing the number of options, it may also reduce the accuracy of velocity estimates.

In addition, the ability to compare stories with each other can be a little bit more complicated, because there is no clear mathematical relationship between a medium and an extra-small. T-shirt size scales also require extra effort on the part of the person coordinating the Agile process. The T-shirt sizes need to be converted to numerical values for the sake of tracking effort over time and charting an estimated velocity for the team. For this reason, while T-shirt sizes can be very effective for teams just starting out with Agile, eventually it's a good idea to move the team toward a more rational numerical scale

RELATIVE MASS VALUATION:

When adopting Agile as a new technique for a team, frequently there will be a large backlog of stories that need to be estimated all at once.

One of the biggest advantages of Agile estimation is that stories are estimated relative to each other, not on the basis of hourly or daily effort. It's usually clear to a team, regardless of their level of experience, if one story is going to be more difficult than another, even when nobody has any idea how long it may take to complete individual stories. But going through the process of individual point estimation for a huge list of stories can be daunting.

Relative mass valuation is a quick way to go through a large backlog of stories and estimate them all as they relate to each other. To use this approach, first write up a card for each story. Then, set up a

large table so that the stories can be moved around easily relative to each other.

Start by picking any story, then get the team to estimate whether they think that it is relatively large, medium, or small. If it's a large story, place it at one end of the table. If it's a small story, it goes at the other end of the table. A medium story goes in the middle. Now select the next story, and ask the team to estimate if it's more or less effort than the story that you just put down. Position the story card on the table relative to the previous card, and go to the next card.

By using this technique, it's possible to go through 100 or more backlog stories and estimate their relative effort in as little as an hour. Everyone on the team will feel a sense of accomplishment when they see the scope of their work laid out in front of them, estimated in the order of effort.

The next step is to assign point values based on the position of the stories on the table. Start with the easiest story that is worth assigning points to, and call it a 1. Then, move up the list of cards, assigning a value of 1 to every story until you get to one that seems at least twice as difficult as the first one. That story gets a 2. You may need to remind the team not to get caught up in the fine details. The idea is to get a rough point estimate, not a precise order.

Ultimately, any story may be completed in any order based on the business value and priority assigned by the product owner, so all the team needs to estimate is how many points one story will take relative to another

BUCKET SYSTEM:

The Bucket System is a way to estimate large numbers of items with a small- to medium-sized group of people and to do it quickly. The Bucket System has the following qualities that make it particularly suitable for use in Agile environments:

- It's fast! A couple of hundred items can be estimated in as little time as one hour.
- It's collaborative. Everyone in a group participates roughly equally.
- It provides relative results, not absolute estimates (points vs. hours).
- The results are not traceable to individuals and so it encourages group accountability.
- Works with teams to estimate effort or with stakeholders to estimate value.

The Bucket System of estimation works as follows:

1. Set up the physical environment, as per the diagram below. Ensure that all the items to be estimated are written on cards.
2. Choose an item at random from the collection and read it to the group and place it in the "8" bucket. This item is our first reference item.
3. Choose another item at random from the collection and read it to the group. The group discusses its relative position on the scale. Once consensus has been reached, put the item in the appropriate bucket.
4. Choose the third item at random and, after discussion and consensus are reached, place it in the appropriate bucket.
5. If the random items have clearly skewed the scale toward one end or the other, rescale the items (e.g., the first item is actually very small and should be in the "1" bucket).
6. Divide and conquer. Allocate all the remaining items equally to all the participants. Each participant places items on the scale without discussion with other participants. If a person has an item that they truly do not understand, then that item can be offered to someone else.
7. Sanity check! Everyone quietly reviews the items on the scale. If a participant finds an item that they believe is out of place, they are welcome to bring it to the attention of the group. The group then discusses it until a consensus is reached, and it is placed in one of the buckets.
8. Write the bucket numbers on the cards so that the estimates are recorded

DOT VOTING:

Dot voting is a technique that allows participants to vote their preferences among a set of items by placing a coloured dot on items that they believe are a higher priority than other items. Items with more dots are a higher priority than items with fewer dots. This technique is frequently used during the sprint retrospective activity.

Dot voting is effective for ranking multiple options. All individuals have the same number of votes, represented as dots, which they can spread across one or more distinct options. Each item on the list has to be granular and well understood by everyone, and then the voting begins.

For example, perhaps there is a list of ten items identified in a retrospective, and the team wants to narrow the list to a couple they will work on. Everyone has three dots to vote on the retrospective items. Some will put one dot each on three different items, others might choose to put all three dots on one option, or perhaps two dots on one option and one on another.

Teams often use stickers and take turns applying them to a list of choices on a flip chart, or they use dry erase pens on a whiteboard to create their dots. There are electronic versions for distributed Agile teams. In many cases, a single round of dot voting will identify the top ranked choices. If not, multiple rounds of voting can be used as the list gets shorter

The method is summarized as follows:

- Post the user stories on the wall by using yellow sticky notes or in some manner that enables each item to receive votes.
- Give four to five dots to each stakeholder.
- Ask the stakeholders to place their votes. Stakeholders should apply dots (using pens, markers, or, most commonly, stickers) under or beside written stories to show which ones they prefer.
- Order the product backlog from the most number of dots to the least.
- When you are done with this first pass, it is almost certain that the stakeholders will not be completely happy with the outcome of the vote. If that is the case, you should review the voting and optimize it

Here's what you can do:

- Arrange the votes into three groups to represent high, medium, and low priorities.
- Discuss stories in each group.
- Move items around to create a high-priority list.
- Make a new vote with items in the high-priority list

AFFINITY MAPPING:

Items are grouped by similarity – where similarity is some dimension that needs to be estimated. This is usually a very physical activity and requires a relatively small number of items (20 to 50 is a pretty good range). The groupings are then associated with numerical estimates if desired.

PRIORITIZATION TECHNIQUES

https://www.hotpmo.com/blog/moscow-kano-prioritize

MoSCoW:

- **M - MUST** have this
- S - SHOULD have this if at all possible

- C - COULD have this if it does not affect anything else
- W - WON'T have this time but would like in the future

The "Must" requirements are non-negotiable, if they are not delivered then the project is a failure, therefore it is in everybody's interest to agree what can be delivered and will be useful. Nice to have features are classified in the other categories of "Should" and "Could.
"Must" requirements must form a coherent set. They cannot just be "cherry picked" from all the others. If they are then what happens is that by default all the other requirements automatically become "Must", and the entire exercise is wasted.
Requirements marked as "Won't" are potentially as important as the "Must" category. It is not immediately obvious why this is so, but it is one of the characteristics that makes MoSCoW such a powerful technique. Classifying something as "Won't" acknowledges that it is important, but can be left for a future release. In fact, a great deal of time might be spent in trying to produce a good "Won't" list. This has three important effects:

- Users do not have to fight to get something onto a requirements list.
- In thinking about what will be required later, affects what is asked for now.
- The designers seeing the future trend can produce solutions that can accommodate these requirements in a future release.

KANO MODEL: [Refer online]

The **Kano model** is a theory of product development and customer satisfaction developed in the 1980s by Professor Noriaki Kano, which classifies customer preferences into five categories. [MARIO]

- Must-Be
- One-Dimensional
- Attractive
- Indifferent
- Reverse

VALUE STREAM MAPPING: [Check PDF for more]

- It visualizes the whole process from idea to customer release as a series of connected tasks. Both value and non-value adding tasks should be defined and tracked from customer requirement to delivery
- By looking at the holistic representation of your complete development cycle, and whether or not the individual steps are adding value, you can create a visual model of your process and easily identify waste in the system

WALKING SKELETON:

- Tiny implementation of a system that performs a small end-to-end function
- It need not use the final architecture, but it should link together the main architectural components
- A vertical slice through all the architectural layers
- Minimal implementation that proves the architecture through some end-to-end functionality
- The architecture & functionality can then evolve in parallel
- Help get early feedback if chosen architecture & technology is suitable
- Build quality becomes consistent as the architecture is defined early

BUSINESS VALUE BASED:

- In this case, each requirement carries a business value it could potentially generate to the company. The business value would be decided either by the Product Owner or the Product owner team.
- The requirement with highest business value is implemented during earlier releases .

TECHNOLOGY RISK BASED:

- In this method, requirements are prioritized based on the risk associated with implementing it. The risk is typically based on the technology.
- The requirement with highest technology risk is implemented during the earlier iterations.

100 POINT METHOD:

- The 100-point method is a prioritization method that can be used to prioritize items in a group environment. Each person within the group is given 100 points which they can distribute as votes across the available items. The votes do not need to be distributed equally; rather a weighted distribution can be used to reflect the higher priority that some items warrant.
- Imagine that a group was trying to prioritize 5 items. A person could decide that each item is of equal importance and assigned 20 points to each. Or, they could decide that item 1 is more important than 2 which is more important than 3, and so on, and therefore spread the votes out in a weighted fashion where item 1 gets 40 votes, item 2 gets 30 votes, item 3 gets 15 votes, etc, until all of the votes are allocated. Each person allocates their 100 points. Then all of the votes are added to determine the final vote count for each item resulting in the prioritized list.

SLICING TECHNIQUES:

Horizontal versus vertical splitting?

Horizontal Splitting involves breaking up user stories by the ***kind of work that needs to be done*** or the ***application layers*** that are involved. So, work that has to be done for the UI, databases, a front-end, a back-end and testing is split up into groups of tasks. This is typically how teams work in a more traditional (waterfall) development context. Although this approach has its use in waterfall development, it is most certainly not recommended for breaking up user stories when Scrumming.

- Individual items do not deliver business value
  - If the team would split up the work horizontally, they would end up with work for design, database, front-end and testing. Although the items are certainly smaller, they don't deliver any business value separately. After all, a product owner cannot go live when only the UI is finished, or when the database is set up
- Increases bottlenecks, instead of reducing them
  - Horizontal splitting is often accompanied by so-called 'silo thinking'. Every member is taken from one of the silos required for software development. The 'design guy' will take care of design, the 'database guy' will set up the database, the 'developer' writes the code and the 'tester' does the testing. If team members are not interchangeable (which is often the case when using this approach) there is a good chance of bottlenecks
- Horizontal slices can't be prioritized
  - Because none of the items delivers business value in themselves, a product owner will be unable to prioritize the work. There is also a good chance that the slices will be fairly technical, which makes it even harder and creates distance and misunderstanding between the product owner and the team

Vertical Splitting is more useful. If a user story is split vertically, it is split in such a manner that the *smaller items still generate some business value*. The functionality will not be split across technical layers or tasks but across functional layers

https://techblog.holidaycheck.com/brilliant-ways-to-slice-user-stories/
https://blog.agilistic.nl/8-useful-strategies-for-splitting-large-user-stories-and-a-cheatsheet/

Strategies of Vertical Splitting:

1. Split by workflow steps
   - If user stories involve a workflow of some kind, the item can usually be broken up into individual steps
   - By breaking up a large user story like this, we have improved our understanding of the functionality and our ability to estimate. It will also be easier for a product owner to make decisions about priority. Some workflow steps may not be important right now and can be moved to future sprints

2. Split by business rules
   - Many user stories involve a number of explicit or implicit business rules
   - Business rules are often implicit, so it will take some skill to unearth them. It may help to a strategy described below (#7); how is the functionality going to be tested?
   - Often, the test cases imply important business rules. Once the business rules have been identified, it will have improved our understanding and ability to estimate.
   - The product owner may decide that some business rules are not important for now or can be implemented in a simplified form

3. Split by happy / unhappy flow
   - Functionality often involves a happy flow and one or more unhappy flows. The happy flow describes how functionality behaves when everything goes well. If there a deviations, exceptions or other problems, unhappy flows are invoked
   - Unhappy flows describe exceptions. By identifying the various flows, we more clearly understand the required functionality. A product owner can also more easily decide what is important right now
   - Again, by splitting up functionality we can more accurately ask questions about the business value, and make decisions accordingly

4. Split by input options / platform
   - Many web applications have to support various input options and/or platforms, like desktops, tablets, mobile phones or touch screens. It may be beneficial to break up large items by their input options
   - By splitting up large items, the product owner can more easily prioritize which input options or platforms are more important

5. Split by data types or parameters
   - Some user stories can be split based on the data types they return or the parameters they are supposed to handle.
   - By breaking up the functionality based on parameters or functionality, we now more clearly understand what kind of search parameters will be used. This allows us to more accurately estimate the functionality, but it also allows a product owner to make decisions about priority
   - Another example of this strategy is when breaking up functionality involving management information based on the returning data types. Some of the information can be presented visually in charts or graphs (data types), but can also be displayed in a tabular format (data type) for now. Perhaps the product owner is ok with exporting the data to Excel and manually creating all the graphs and charts there for the time being

6. Split by operations
   - User stories often involve a number of default operations, such as Create, Read, Update or Deleted (commonly abbreviated as CRUD).
   - CRUD operations are very prevalent when functionality involves the management of entities, such as products, users or orders

- When presented with this strategy, many teams wonder if the smaller items actually deliver business value. After all, what's the point of only creating entities, when you cannot update or delete them afterwards?
- But perhaps the product owner has such a limited number of products, that editing or deleting can be done through a database manager directly. Sometimes, an operation can be easily implemented in a simplified form. Deleting a product can be done in two ways; you can completely drop the record and all associated records, or you can 'soft delete' a product. In that case, the product is still in the database, but it is marked as 'deleted'. Sometimes one of these approaches is easier to implement and can be 'good enough' for now

7. Split by test scenarios / test case
   - This strategy is useful when it is hard to break down large user stories based on functionality alone. In that case, it helps to ask how a piece of functionality is going to be tested. Which scenarios have to be checked in order to know if the functionality works?
   - This strategy actually helps you apply the other strategies implicitly. By thinking about testing, you automatically end up with a number of business rules (#1, #2, #3 and #4) , (un)happy flows (#1, #2 and #3) and even input options (#5). Sometimes, test scenarios will be very complicated because of the work involved to set up the tests and work through them.
   - If a test scenario is not very common, to begin with or does not present a high enough risk, a product owner can decide to skip the functionality for the time being and focus on test scenarios that deliver more value. In other cases, test scenarios can be simplified to cover the most urgent issues, Either way, relevant test scenarios can be easily translated into backlog items and added to the sprint or the backlog

8. Split by roles
   - User stories often involves a number of roles (or groups) that performs parts of that functionality
   - By breaking up functionality into the roles that have to perform bits of that functionality, we more clearly understand what functionality is needed and can more accurately estimate the work involved.
   - Writing user stories is a great way to apply this strategy. It also helps the product owner prioritize. Some roles may be less relevant at the moment and can be implemented later

The advantages of vertical slices:
When we take the time to slice our stories into very small pieces, a lot of cool agile magic happens.

- The smaller slices are much easier to understand, so there is less likelihood of a lack of consensus among team members.
- The small size also tends to make estimates more accurate, if your team is estimating these slices.
- When we decompose into small pieces, we often realize that not every single piece is really required by the user, so we can take advantage of the Pareto Principle and eliminate from our plan some of the “nice to haves” until we get validation from users that they won’t those pieces.
- Smaller slices give us a faster feedback loop – we find defects in design, usability, code, and integration faster. We can get them to users sooner for their feedback.
- If your team has specialized people on it (testers, DB people, etc.), they’re not sitting around waiting while a bunch of work is done in a previous step – since the slices are small, little pieces of work are flowing through the system quickly.
- According to Teresa Amabile’s awesome book “The Progress Principle,” the single biggest factor (by far) in the engagement at work is small progress every day. Small slices allow the team to get small wins on an almost daily basis, leading to greater engagement.
- Your daily scrums become far more interesting and useful. Instead of “yeah, I’m 20% done adding all of the new schemas to the DB”, you get “we got the time stamps working in the UI, and today we’re going to make the data persist when we go offline”.

When is a user story small enough?

- There is no clear-cut rule on how small a user story should ideally be. This greatly depends on the length of the sprints, the nature of the application and the size of the team. **In general, it is a good idea to have at least 5 to 10 user stories in a sprint.** Some of those can be a bit bigger, with a larger number of smaller items. This gives the team enough flexibility
- Whenever a story is larger, it is broken up. But this all depends on preferences. Some teams may prefer a cut-off value that is smaller or larger
- A sprint with many small items is always preferable to a sprint with a handful of large items
- Make good use of the retrospective to reflect on the burndown and determine if the user stories are perhaps too large or too small

How to Decompose User Stories into Tasks?

A task is a piece of activity that is required to get a story done. Here are some effective tips for breaking down a user story into tasks.

1. Create Meaningful tasks
    - Describe the tasks in such a way that they convey the actual intent. For example, instead of saying Coding, describe the tasks as "Develop the login class", "Develop the scripting part for login functionality", "Develop the password encryption for the login functionality", "Create user table and save the login data to DB" etc.
    - Such tasks are more meaningful rather than just saying coding and testing

2. Use the Definition of Done as a checklist
    - The DOD defines the completeness criteria of a story. It includes all items that have to be completed to claim that the story is done by the development team. A simple example:
        - Acceptance criteria is verified during testing
        - Coding tasks completed.
        - Exploratory Testing completed and signed.
        - Regression test reviewed and passed.
        - Unit testing – written and passed.
        - Code reviews conducted.
        - Defects are in an "acceptable" state to the Product Owner.
        - User story accepted by the product owner.
        - Regression tests run and passed
        - Smoke / automation tests run (if applicable)
        - Check for memory leaks
        - Automated unit tests are checked in

3. Create tasks that are right sized
    - Another common syndrome is that tasks which are very small, broken down to a minute level like, 10 min, 30 min, 5 min tasks, for example: Write Accept User Name Password, Validate Login, and Give Error Messages. Breaking the user stories with too many details is an overhead.
    - What is the ideal size of the tasks? One guideline is to **have tasks that span less than 8 hours so that each one of them can be completed in at least a day**

4. Avoid explicitly outlining a unit testing task
    - If possible, make unit testing, not a separate task but part of the implementation task itself. This encourages people to practice Test Driven Development as an approach. However, this practice may not be ideally suitable for new Scrum teams

5. Keep your tasks small

- Do not have tasks that span across days together. It makes it difficult to know the actual progress.
- In some mature teams, I have seen, they do not do the task breakdown at all. They just pull in the user stories and complete them, but it is a journey for new Scrum teams to get there and requires a strong cohesive team, and many sprints of working together

SPLITTING EPICS

EXAMPLES OF EPICS:

As a VP Marketing, I want to review the performance of historical promotional campaigns so that I can identify and repeat profitable campaigns.

This epic can be split as follows:

- As a VP Marketing, I want to select the time frame to use when reviewing the performance of past promotional campaigns, so that …
- As a VP Marketing, I can select which type of campaigns (direct mail, TV, email, radio, etc.) to include when reviewing the performance of past so that …

The second story can also be further broken down into:

- As a VP Marketing, I want to see information on direct mailings when reviewing historical campaigns so that…
- As a VP Marketing, I want to see information on TV ads when reviewing historical campaigns so that…
- As a VP Marketing, I want to see information on email ads when reviewing historical campaigns so that…
  and so on for each type of ad campaign.

As a hotel operator, I want to set the optimal rate for rooms in my hotel.

- As a hotel operator, I want to set the optimal rate for rooms based on prior year pricing.
- As a hotel operator, I want to set the optimal rate for rooms based on what hotels comparable to mine are charging.
  - As a hotel operator, I can define a 'Comparable Set' of hotels. [Comparable Set was a term used widely in the company.]
  - As a hotel operator, I can add a hotel to a Comparable Set.
  - As a hotel operator, I can remove a hotel from a Comparable Set.
  - As a hotel operator, I can delete a Comparable Set I no longer wish to use.
  - As a hotel operator, rates charged at hotels in a Comparable Set are used to determine rates at my hotel.
- As a hotel operator, I want to set the optimal rate for rooms based on currently projected occupancy.
- As a hotel operator, I can set parameters related to optimal rates such as target occupancy, minimum acceptable occupancy and maximum acceptable occupancy [could be more than 100%].

15 Ways to Split an Epic:

1. Extract a smaller story by focusing on a particular user role or persona. (“Prioritize your users first, then your user stories.” -- Jeff Patton) E.g.: “first time user,” “social networker,” “my mom,” etc.

2. Extract a smaller story by substituting basic utility for usability. (First, make it work, then make it pretty.)
3. Extract a smaller story by splitting on CRUD (Create, Read, Update, Delete) boundaries.
4. Extract a smaller story by focusing on distinct scenarios, such as the "happy path" (main success scenario) vs. alternate (exception) flows.
5. Extract a smaller story by focusing on a simplified data set.
6. Extract a smaller story by focusing on a simplified algorithm.
7. Extract a smaller story by buying some component(s) instead of building everything yourself.
8. Extract a smaller story by discarding technologies that increase hassle, dependency, and vendor lock.
9. Extract a smaller story by substituting some manual processes for full automation.
10. Extract a smaller story by substituting batch processing for online processing.
11. Extract a smaller story by substituting generic for custom.
12. Extract a smaller story by reducing supported hardware/OS/client platforms.
13. Extract a smaller story from the acceptance criteria of another story.
14. Extract a smaller story by substituting "1" for "many."
15. Extract a smaller story by scanning for keywords such as "and," "or," periods, and other kinds of separators

EXTRAS

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 totalling the Points for all fully completed User Stories
- To calculate velocity of your agile team, simply *add up the estimates of the features, user stories, requirements or backlog items successfully delivered* in an iteration
- Velocity measures how much functionality a team delivers in a sprint
- Velocity measures a team's ability to turns ideas into new functionality in a sprint
- Velocity is measured in the same units as feature estimates, whether this is story points, days, ideal days, or hours that the Scrum team delivers – all of which are considered acceptable

- Along with release and iteration burndown charts, measuring the velocity of agile teams has proven to provide tremendous insight/visibility into project progress and status. A velocity chart shows the sum of estimates of the work delivered across all iterations
- Typically velocity will stabilize through the life of a project unless the project team makeup varies widely or the length of the iteration changes. As such, velocity can be used for future planning purposes

How is the first iteration's velocity estimated?

- For an agile team's first iteration, a general guideline is to *plan **initial velocity** at **one-third of the available time***. If you're estimating ideal programmer time then this accounts for meetings, email, design, documentation, rework, collaboration, research, etc.
- As an example, with six programmers and two-week iterations, a total of 60 programmer-days (6 programmers x10 days) are available. In this situation, a good start would be to plan 20 ideal days worth of work in the iteration. If using actual time, include enough of a buffer to account for standard project 1) overhead and 2) estimation inaccuracy

- Velocity will typically fluctuate within a reasonable range, which is perfectly fine. If velocity fluctuates widely for more than one or two iterations, the Scrum team may need to re-estimate and/or renegotiate the release plan
- For most agile development teams velocity will typically stabilize between 3 and 6 iterations

TYPES OF VELOCITY:

- Optimal/Ideal Velocity:
  - Maximum velocity the team has achieved in the past and they would want to keep achieving

- Initial Velocity:
  - Usually for the first/conditional sprint if the team is mature
  - It is predicted on previous factors or on capacity

- Planned Velocity:
  - Based on Focus Factor
  - Planned for 1/3 of the effort
  - IRL, ScrumMaster discusses with Product Owner which features they want to the end of the Sprint (Commitment drove)

- Average Velocity:
  - Velocity calculated based on the last few sprints

STORY POINTS VS HOURS:

Traditional software teams give estimates in a time format: days, weeks, months. Many agile teams, however, have transitioned to story points. Story points rate the relative effort of work in a Fibonacci-like format: 0, 0.5, 1, 2, 3, 5, 8, 13, 20, 40, 100. It may sound counter-intuitive, but that 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.

Teams starting out with story points use an exercise called Planning Poker. At Atlassian, planning poker is a common practice across the company. The team will take an item from the backlog, discuss it briefly, and each member will mentally formulate an estimate. Then everyone holds up a card with the number that reflects their estimate. If everyone is in agreement, great! If not, take some time (but not too much time–just couple minutes) to understand the rationale behind different estimates. Remember though, estimation should be a high-level activity. If the team is too far into the weeds, take a breath, and up-level the discussion

FEATURE vs STORY vs TASK:

- A **feature** is *something tangible that works and which we could potentially deliver* if it was enough to provide business benefit. Usually, a bit of a screen or page, together with all the logic and validation associated with it, is a feature. A feature is a highest-level thing which a developer could work on without doing significant bits of analysis. It's a part of an idea made at least concrete enough to imagine.

- A **story** is a *part of a feature* on which we *can get feedback from relevant stakeholders*. For instance, we might code the UI, then the validation and messages, then an aspect or two of the

behaviour behind the UI. Each of these would be a possible story. The only real reason for splitting features into stories is to get faster feedback and a better idea of progress.

- A **task** is a part of a story which *doesn't allow us to get feedback from relevant stakeholders*. If you can get feedback, it's probably a story rather than a task, because it affects the UI in some way, or at least gives something that can be visibly verified and critiqued and which stakeholders care about. There are lots of UIs and lots of stakeholders and users, so "put logging in to help maintenance debug the situation where the elephant appears on the screen" might be a perfectly good story. Anything that's just a part of a story is a task.

WHY DO SOME TEAMS SPLIT STORIES INTO TASKS?

- **It can lead to more accurate estimates.** Splitting stories into tasks can help developers to think about the amount of work involved in a story, and find any pieces they missed.
- **It can help with collaboration.** By splitting a story into tasks, developers can each take that task – usually a different horizontal slice – and work with minimal merge conflicts and just a bit of adjustment to get their piece working with other people's. Splitting a story into tasks is useful for swarming, and a lot of swarming teams do this instinctively, even if they haven't written the tasks down.
- **It can help senior developers mentor junior developers** and verify that they're taking a good approach towards a story. Some mistakes are worth avoiding.
- They can be **used as a measure of progress** to someone tracking the team, like a PM or Scrum Master.

HOW TO SPLIT A "USER STORY"?

Refer document shared by Aadav

"DONE" VS "READY": WHAT IS THE DIFFERENCE?

- The Definition of Done is a set of rules or criteria that the team adopts as a guide for when they can legitimately claim that a given user story (or even a task) is "done" and ready for review and approval by the product owner. In the past, I've seen the following as common criteria for "done":
  - Code review completed by another developer;
  - Automated unit tests are written and running;
  - Smoke tests performed on dev/staging environment;
  - QA sign off or paired programming session;
  - Meets acceptance criteria stated in the story.

Definition of "Done" is the criteria of satisfaction at the product level to avoid technical debt

- The item should be **properly tested**.
- The item should be **refactored**.
- The item should be **potentially shippable**.

- The **Definition of Ready** is a set of rules or criteria that the team adopts as a guide for when a story can legitimately be moved from the backlog into a Sprint -- either during Sprint Planning or during a Sprint where all committed stories have been completed. I've honestly rarely seen this adopted in practice, but could see the following criteria applying:
  - Story has been **reviewed and estimated** by the team;
  - Story is complete in format - User X needs to Y so they can Z;
  - **Acceptance criteria** are clear and agreed upon;
  - Product Owner has **approved** the story

ACCEPTANCE CRITERIA:

- User Stories are specific requirements outlined by various stakeholders as they pertain to the proposed product or service. Each User Story will have associated User Story Acceptance Criteria (also referred to as "Acceptance Criteria"), which are the objective components by which a User Story's functionality is judged.
- Acceptance Criteria are developed by the Product Owner according to his or her expert understanding of the customer's requirements. The Product Owner then communicates the User Stories in the Prioritized Product Backlog to the Scrum Team members and their agreement is sought. Acceptance Criteria should explicitly outline the conditions that User Stories must satisfy. Clearly, defined Acceptance Criteria are crucial for timely and effective delivery of the functionality defined in the User Stories, which ultimately determines the success of the project

CAPACITY PLANNING:

VELOCITY DRIVEN PLANNING:

- Velocity-driven sprint planning is based on the premise that the amount of work a team will do in the coming sprint is roughly equal to what they've done in prior sprints. This is a very valid premise
- The steps in velocity-driven sprint planning are as follows:
  - Determine the team's historical average velocity.
  - Select a number of product backlog items equal to that velocity.
  - Some teams stop there. Others include the additional step of:
  - Identify the tasks involved in the selected user stories and see if it feels like the right amount of work. Some go further & .
  - Estimate the tasks and see if the sum of the work is in line with past sprints

COMMITMENT DRIVEN PLANNING: (Read from Mountain Goat):

A commitment-driven sprint planning meeting involves the product owner, ScrumMaster and all development team members. The product owner brings the top-priority product backlog items into the meeting and describes them to the team, usually starting with an overview of the set of high-priority items

RETROSPECTIVE TECHNIQUES:

Silent Writing:

- In this technique, each member in the meeting is given a stack of post-it notes in which they write one observation per Post-It note about the previous Sprint. The discussion is held off until everyone is finished writing.
- The Scrum Master then facilitates dot-voting to identify top issues from the paper groupings. ***This method identifies top issues while giving everyone in the room a chance to be heard in a collaborative environment*** – and it's very helpful for engaging introverts who may not want to speak up until they are comfortable. It also allows the team to get on the same page (literally). Once the team identifies an issue to discuss (a challenge, for example) they can work together to figure out a solution.

Happiness Histogram:

- Prepare a visual with a horizontal scale from 1 (Unhappy) to 5 (Happy). Using sticky notes or virtual techniques, ask each person to signify where they would place themselves on the happiness scale. After everyone has identified where they are, have each person comment.
- Depending on how you want to facilitate this, you can either have the team do a deep dive right then and there or postpone a more nuanced conversation about what people observe until later in the retrospective.
- The reason this is referred to as a histogram is that if one person has the same happiness score as another, they place their sticky above the one with the same score

ORID Technique:

ORID stands for Objective, Reflective, Interpretive and Decisional.
Interpreted as a consecutive order of stages, it reflects the natural order in which humans think through an issue

Engage the team in a series of questions that help slow down decisions and gather insights before moving to recommendations:

- O: Ask objective questions first. "What was happening? What did you notice?" Have team members work in small groups to gather these items.
- R: Ask reflective questions next. "What reaction did you have to that? What was challenging or helpful?"
- I: Ask interpretive questions. "What does that say about how we work? What might be some recommendations for our work?"
- D: Ask decisional questions. "Given what we have recorded here, what new agreements or practices might we invite into our next iterations?"

The Wheel (also known as the Starfish):

1. Draw a large circle on a whiteboard and divide it into five equal segments
2. Label each segment 'Start', 'Stop', 'Keep Doing', 'More Of', 'Less Of'
3. For each segment pose the following questions to the team:
   a. What can we start doing that will speed the team's progress?
   b. What can we stop doing that hinders the team's progress?
   c. What can we keep doing to do that is currently helping the team's progress?
   d. What is currently aiding the team's progress and we can do more of?
   e. What is currently impeding the team's progress and we can do less of?
4. Encourage the team to place stickies with ideas in each segment until everyone has posted all of their ideas
5. Erase the wheel and have the team group similar ideas together. Note that the same idea may have been expressed in opposite segments but these should still be grouped together
6. Discuss each grouping as a team including any corrective actions

The Sail Boat (Or speed boat. Or any kind of boat):

1. Draw a boat on a whiteboard. Include the following details:
   a. Sails or engines – these represent the things that are pushing the team forward towards their goals
   b. Anchors – these represent the things that are impeding the team from reaching their goals
2. Explain the metaphors to the team and encourage them to place stickies with their ideas for each of them on appropriate area of the drawing
3. Wait until everyone has posted all of their ideas
4. Have the team group similar ideas together
5. Discuss each grouping as a team including any corrective actions going forward

Mad Sad Glad:

1. Divide the board into three areas labelled:
   a. Mad – frustrations, things that have annoyed the team and/or have wasted a lot of time
   b. Sad – disappointments, things that have not worked out as well as was hoped
   c. Glad – pleasures, things that have made the team happy

2. Explain the meanings of the headings to the team and encourage them to place stickies with their ideas for each of them under each heading
3. Wait until everyone has posted all of their ideas
4. Have the team group similar ideas together
5. Discuss each grouping as a team identifying any corrective actions

Guess who:

- It's a retrospective from a different perspective. It is an 'empathy hack' retro designed for team members to see others' points of view by selecting a colleague's name from a hat. Team members are encouraged to play the role of their colleagues and view the world from a different perspective.
- Guess who helps to increase the power of talking about each other

Futurespectives:

Scrum proposes that the retrospective is done at the end of the sprint (iteration). But you can also use retrospectives at the beginning of an iteration or when a new team is assembled. Such a retrospective is called a put respective: A retrospective where you start from the goal and explore ways how to get there

In futurespective teams places themselves in the future by imagining that their goal has been reached. They start such retrospectives by discusses the team goals to assure that team members build a common understanding. The goals are formulated and written d8own so that they are visible for everybody.

Optionally teams can write down which benefits they got from attaining their goals. If teams like to party they can even do a small celebration for having reached the goals, which can help to make teams aware of the importance of reaching their goal.

Next teams discuss their imaginary past and explore how they have gotten to their goals. There are two things that teams questions themselves:

- What are the things that have helped us to get here?
- Which things made it hard for us to reach our goal?

Optionally teams can also discuss:

- What did we learn as a team along the way towards reaching our goal?

Teams can use sticky notes, flip-overs and/or white boards to capture the results from the discussions. When you do the exercise with remote teams you can use a Google Doc or tools like tools like Lino or Group map.

Now the teams go back to the present. The results from exploring the past are used to agree how to work together in teams to reach the goal. This can, for example, be done by:

- Defining a Definition of Done
- Making a list of tools that will be used, processes/practices that teams will use, etc
- Defining actions that are needed to work together effectively

The Perfection Game:

The Perfection Game can be used to get feedback on a product or service that has been provided. It is also a retrospective exercise usable to discover strengths and define effective improvement actions. The perfection game gives power to the teams and helps them to self-organize and become agiler.

To get feedback on a perfection game you ask people to provide answers to the following questions:

- I rate the product/service … on a scale from 1-10
- What I liked about it …

- To make it perfect …

People have to rate the value that they received on a scale from 1 to 10, based on how much value they think they could add themselves by improving the product or service. For example, when there is nothing that they think they can improve, they should rate with a 10. If they think that they could make it twice as valuable, they should give it a 5

Speed Dating:

Agile speed dating is a good way to discuss delicate subjects. After the first sprints, the easy problems are solved and the more difficult ones remain. To solve these difficult subjects agile speed dating is a recommended method.

Setting up the venue – 5 minutes
Before the group enters the space, place sets of chairs facing one another throughout the room. Explain the rules to your team and have them divide into two groups. Group A will remain stationary. Group B will rotate clockwise upon hearing the buzzer or bell. Each individual within both groups should be given a card that contains their name.

Speed Dating – 5 minutes per pair
Assuming you have 10 teammates, there will be 5 rotations in total. Each encounter should be time boxed to 5 minutes. This gives each teammate the chance to voice their thoughts around the iteration and have an open dialogue around opinions they may share or even differences of opinion. As the host, it is your responsibility to sound a buzzer at each 5-minute interval so that Group B can rotate to the next pairing and begin their discussion promptly.

Speed Dating Retrospective – 5 minutes
By now, the rotation has completed and each member of Group A has had a conversation with each member of Group B. Instruct each teammate to write the name of someone in the opposite group whom they felt either the most in sync with in regards to their iteration thoughts, or someone that shared something of great value. Once you collect the cards, you will take a few moments to hopefully find a few that match.

Speed Dating Results – 15 minutes
Invite the matches up to the front of the room and congratulate them for recognizing and appreciating one another's opinions. Have each one of them take a moment to explain something that stuck out around their conversation.

Speed Dating "Duds" / Wrap up – 15 minutes
While not everyone may have established a "match", we know there was still a lot of great dialogue. This is an opportunity for people to talk about meaningful conversation points that they shared during the last hour. Encourage your teammates to share points that were discussed by others vs. those that they initiated themselves.

Safety Check:

- At the start of an agile retrospective, you can do a safety check by asking people to write down how safe they feel in the retrospective. If the score indicates that people feel unsafe, then that will have a serious impact on the retrospective. Here are some suggestions how you can deal with this when facilitating retrospectives.
- In a safety check, people can use a score from 1 to 5. The scoring is done anonymously. A score of 5 means that they feel that they will be fully open and honest in the meeting and are willing to talk about anything, where a 1 means that they don't feel safe at all and don't want to speak up

PRODUCT BACKLOG ITEMS NEED TO BE INVEST:

- Independent
  - Stories are easiest to work with if they are independent. That is, we'd like them to not overlap in concept, and we'd like to be able to schedule and implement them in any order

- Negotiable:
  - A good story is negotiable. It is not an explicit contract for features; rather, details will be co-created by the customer and programmer during development. A good story captures the essence, not the details. Over time, the card may acquire notes, test ideas, and so on, but we don't need these to prioritize or schedule stories

- Valuable:
  - A story needs to be valuable. We don't care about value for just anybody; it needs to be valuable to the customer. Developers may have (legitimate) concerns, but these framed in a way that makes the customer perceive them as important.
  - This is especially an issue when splitting stories. Think of a whole story as a multilayer cake, e.g., a network layer, a persistence layer, a logic layer, and a presentation layer. When we split a story, we're serving up only part of that cake. We want to give the customer the essence of the whole cake, and the best way is to slice vertically through the layers. Developers often have an inclination to work on only one layer at a time (and get it "right"), but a full database layer (for example) has little value to the customer if there's no presentation layer.
  - Making each slice valuable to the customer supports XP's pay-as-you-go attitude toward infrastructure

- Estimable:
  - A good story can be estimated. We don't need an exact estimate, but just enough to help the customer rank and schedule the story's implementation
  - Being estimable is partly a function of being negotiated, as it's hard to estimate a story we don't understand. It is also a function of size: bigger stories are harder to estimate
  - Finally, it's a function of the team: what's easy to estimate will vary depending on the team's experience. (Sometimes a team may have to split a story into a (time-boxed) "spike" that will give the team enough information to make a decent estimate, and the rest of the story that will actually implement the desired feature.)

- Small:
  - Good stories tend to be small. Stories typically represent at most a few person-weeks worth of work. (Some teams restrict them to a few person-days of work.) Above this size, and it seems to be too hard to know what's in the story's scope. Saying, "it would take me more than a month" often implicitly adds, "as I don't understand what-all it would entail." Smaller stories tend to get more accurate estimates.
  - Story descriptions can be small too (and putting them on an index card helps make that happen). Alistair Cockburn described the cards as tokens promising a future conversation. Remember, the details can be elaborated through conversations with the customer

- Testable:
  - A good story is testable. Writing a story card carries an implicit promise: "I understand what I want well enough that I could write a test for it."
  - Several teams have reported that by requiring customer tests before implementing a story, the team is more productive. "Testability" has always been a characteristic of good requirements; actually writing the tests early helps us know whether this goal is met.
  - If a customer doesn't know how to test something, this may indicate that the story isn't clear enough, or that it doesn't reflect something valuable to them, or that the customer just needs help in testing.

- A team can treat non-functional requirements (such as performance and usability) as things that need to be tested. Figure out how to operationalize these tests will help the team learn the true needs

SPIKE:

- A *time boxed period* used to *research a concept* or *create a simple prototype*
- Spikes can either be planned to take place in between Sprints or, for larger teams, a spike might be accepted as one of many Sprint delivery objectives
- Spikes are *often introduced before the delivery of large or complex Product Backlog Items* in order to secure budget, expand knowledge, or produce a proof of concept
- The *duration* and objective(s) of a spike is agreed between Product Owner and Development Team *before the start*
- Unlike Sprint commitments, spikes *may or may not deliver tangible, shippable, valuable functionality*
- For example, the objective of a spike might be to successfully reach a decision on a course of action. The spike is over when the time is up, not necessarily when the objective has been delivered
-

How to integrate a Spike with Scrum?

- Spike is a timebox for the Dev Team to get enough knowledge for estimating a story. The value for the product owner is, that he can then do his release planning, which would otherwise not be possible. During a Spike, the Dev Team can create a functional prototype, even neglecting the DoD - but it has to throw it away afterwards. It's just for learning, not for creating product value
- I usually do not have Spikes with experienced teams. With inexperienced teams, they do have some value, especially in contexts where Agility is not yet fully understood and the wish for precise estimates is high
- Usually, the Product Owner does not carry around Spikes in his Backlog. Only when during the Sprint preparation meeting the Team can't estimate something and asks for a Spike, he might negotiate the timebox, take it on the backlog, and only keep it there for the next Sprint Planning meeting.
- While this is a controversial practice, I have seen situations where this tool was beneficial. Just make sure the Scrum rules aren't broken!

Types of Spikes:

1. Functional Spike: is *used when there is uncertainty how a user might interact with the system. They are evaluated through prototyping, by using wireframes, mockups etc.*

2. Technical Spike*: are used to researching various technical approaches in the solution domain. For e.g. to evaluate the potential performance or load impact of a user story.*

SCRUM RELEASE PLANNING:

- A *very high-level plan* for *multiple* Sprints (e.g. three to twelve iterations) is created during the Release planning. It is a guideline that *reflects expectations about which features will be implemented and when* they are completed. It also serves as a *base to monitor progress* within the project. Releases can be intermediate deliveries done during the project or the final delivery at the end. To create a Release Plan the following things have to be available:
  - A prioritized and estimated Scrum Product Backlog
  - The (estimated) velocity of the Scrum Team
  - Conditions of satisfaction (goals for the schedule, scope, resources)
- Depending on the type of project **(feature- or data-driven)** the release plan can be created in different ways: If the project is feature-driven, the sum of all features within in a release can be

divided by the expected velocity. This will then result in the number of sprints needed to complete the requested functionality.

- If the project is data-driven we can simply multiply the velocity by the number of Sprints and we'll get the total work that can be completed within the given timeline
- Like the Scrum Product Backlog, the Release plan is not a static plan. It will change during the whole project when new knowledge is available and e.g. entries in the Scrum Product Backlog are changed and re-estimated. Therefore the Release Plan should be revisited and updated at regular intervals, e.g. after each Sprint

EXPECTATION CHARTS:

One of the most important tasks for any development team is to appropriately set expectations. One way to help set expectations with stakeholders is to use the **expectation charts** below and ask them to choose either "fixed date delivery" or "fixed scope delivery".
With a **fixed date delivery**, what can be delivered is unknown, but the team should be able to provide low expectation and high expectation projections that create scope boundaries.

With a **fixed scope delivery**, when the project will be delivered is unknown, but the team should be able to provide slow expectation and fast expectation projection estimates that create time boundaries).

SPRINT ZERO: (RESEARCH MORE):

- Idea is to prepare before the first sprint
- It is not a sprint as there is no value delivered or a Potentially Shippable Product
- It is not always required
- Scrum teams can straight up dive in and start working

- 2-3 days is a good length
  - Team engages
  - Start finding users
  - Customers & developers come together to form a team & Chat
- Good way to find more users
- Clarify goals in Sprint Zero. Free-flowing format & brainstorming
- Discuss Product Backlog & start populating it with items
- What should a Minimum Viable Product have?
  - Team prioritizes the backlog and then figures what an MVP would have

WHAT IS SWAG?

- Swag stands for Scientific Wild Ass Guess and is a slang word meaning a rough estimate from an expert in the field, based on his/her experience and intuition. A Scientific Wild Ass Guess is more precise than a Wild Ass Guess since Swag is estimated by an expert
- In the Agile world, Swag is to portfolio items what story points are to backlog items. Swag provides a way to compare the size, time, and effort that it will take to complete a set of features without going through detailed backlog estimating activities
- Swag does not have to be the same unit as your backlog item estimate. In fact, it's better to use a higher level unit for Swag as a constant reminder that it is even less precise (though not necessarily less accurate). A unit such as Team Weeks or Ideal Months can be much more appropriate for a Swag than, say, story points
- Similar to estimating backlog items, you estimate portfolio item using SWAG. To get started, choose a typical portfolio item as your baseline and assign a nice round number to it. Perhaps you give your typical portfolio item a Swag of 10. Then, a portfolio item that is twice as complex

might get a Swag of 20. A portfolio item that is very small might get a Swag of 1. Over time, you'll get better at assigning a Swag that is relative to your baseline
- Portfolio Items include fields for Risk and Value. You can use the Risk and Value fields to estimate portfolio items instead of Swag. If you prefer to use backlog item Estimate Rollups as portfolio item size indicators, you can hide the portfolio item estimating fields in your project workspace

- What does a good user story look like? What is its structure?

A good user story:
- has a description,
- defines acceptance criteria,
- can be delivered within a single sprint
- has all UI deliverables available,
- has all (probably) dependencies identified,
- defines performance criteria,
- defines tracking criteria, and
- is estimated by the team

REQUIREMENTS CHURN & WHO IS RESPONSIBLE?
- Requirements churn is the *change to backlog items from the time they are entered until the product goes into production*
- 10-15% churn is not a problem. But 50-200% churn means someone is wasting a lot of time putting stuff in the backlog that is going to change, they do not have much of a sense of what is certain and what is not
- High churn is often a sign the backlog is used to try to deter change and insulate the organization from uncertainty, rather than creating a process that is good at responding to events as they unfold and allowing the organization to deal with uncertainty
- Either the *Product Owner* could be responsible for this, as he/she would be adding any incoming requirements to the Product Backlog. Or it could be someone from the Development team adding items that do not add value, which shows that there is uncertainty about requirements

- Capacity of a Team

FOCUS FACTOR:
- Focus factor is a simple mathematical formula for ***forecasting the number of deliverables possible in an iteration***. In an Agile environment, this number can be arrived at by considering the capacity and velocity of a development team. The prerequisites for using focus factor are:
  - Velocity calculated as an average of completed story points over the past three to five iterations
  - Requirements (such as user stories) estimated using story points
  - The development team's capacity (excluding nonworking days and time spent in meetings)
- A team's focus factor is calculated as **Focus Factor = Velocity/ Capacity**
- Example: If my team has 7 members who are productive for 6 days each, and as a team, they have a velocity of 31, then the focus factor is calculated as Focus factor = 31 / (7 * 6) = 0.74
- This focus factor can now be *used to forecast the deliverables for the future iterations*. So if only five members are available during an iteration, the achievable story points are calculated as: ***Forecast = Focus factor * Capacity = 0.74 * 5 * 6 = 22 [story points]***
- Maintaining the average velocity and focus factor of the past three to five iterations provides a good forecast of the immediate next iteration. This also helps us analyse and adjust the team's current performance and capability.

MINIMAL VIABLE PRODUCT (MVP)
- Building a Minimum Viable Product (MVP) is a strategy for avoiding the development of products that customers do not want

- The idea is to rapidly build a minimum set of features that is enough to deploy the product and test key assumptions about customers' interactions with the product
- The MVP is called minimum, as you should spend as little time and effort to create it
- MVP is a strategy for iteratively learning about your customers. It "helps entrepreneurs start the process of learning as quickly as possible." In contrast to building a product you think has value, MVP proposes that you learn about your vision by getting potential customers to test an idea/prototype/initial offering, then adapt your product based on the feedback collected – rather than invest a lot of time and money in a fully-fledged product before you know whether it has the required value

POTENTIALLY SHIPPABLE PRODUCT INCREMENT (PSPI):

- The output of every Sprint is called a Potentially Shippable Product Increment. The work of all the teams must be integrated before the end of every Sprint—the integration must be done during the Sprint.

SCRUM KEYWORDS:

- Velocity= no of Stories x no of Story Points in one Sprint
- Story Points
- Use Cases
- Done vs Ready vs PSP
- Testing & Refactoring

JENKINS FORMAT:

- Jenkins is an open-source continuous integration software tool written in the Java programming language for testing and reporting on isolated changes in a larger code base in real time. The software enables developers to find and solve defects in a code base rapidly and to automate testing of their builds.

CONTINOUS INTEGRATION:

- Continuous Integration (CI) is a development practice that requires developers to **integrate code into a shared repository several times a day**. **Each check-in** is then ***verified by an automated build***, allowing teams to detect problems early
- By integrating regularly, you can detect errors quickly, and locate them more easily
- Teams practising continuous integration seek two objectives:
  - minimize the duration and effort required by each integration episode
  - be able to deliver a product version suitable for release at any moment

REGRESSION TESTING:

- After implementing new features or bug fixes, you **re-test scenarios which worked in the past**. Here you cover the *possibility in which your new features break existing features*
- Verifying that new defects are not introduced into existing code. These tests can be manual or automated

REFACTORING:

Refactoring consists of improving the internal structure of an existing program's source code while preserving its external behaviour.

FOR FURTHER UNDERSTANDING

The Sprint

The heart of Scrum is a Sprint, a time-box of one month or less during which a "Done", useable, and potentially releasable product Increment is created. Sprints best have consistent durations

throughout a development effort. A new Sprint starts immediately after the conclusion of the previous Sprint.
Sprints contain and consist of the Sprint Planning, Daily Scrums, the development work, the Sprint Review, and the Sprint Retrospective.
During the Sprint:

- No changes are made that would endanger the Sprint Goal;
- Quality goals do not decrease; and,
- The scope may be clarified and renegotiated between the Product Owner and Development Team as more is learned.

Each Sprint may be considered a project with no more than a one-month horizon. Like projects, Sprints are used to accomplish something. Each Sprint has a definition of what is to be built, a design and flexible plan that will guide building it, the work, and the resultant product.
Sprints are limited to one calendar month. When a Sprint's horizon is too long the definition of what is being built may change, complexity may arise, and risk may increase. Sprints enable predictability by ensuring inspection and adaptation of progress toward a Sprint Goal at least every calendar month. Sprints also limit risk to one calendar month of cost.

Cancelling a Sprint

A Sprint can be cancelled before the Sprint time-box is over. Only the Product Owner has the authority to cancel the Sprint, although he or she may do so under influence from the stakeholders, the Development Team, or the Scrum Master.
A Sprint would be cancelled if the Sprint Goal becomes obsolete. This might occur if the company changes direction or if market or technology conditions change. In general, a Sprint should be cancelled if it no longer makes sense given the circumstances. But, due to the short duration of Sprints, cancellation rarely makes sense.
When a Sprint is cancelled, any completed and "Done" Product Backlog items are reviewed. If part of the work is potentially releasable, the Product Owner typically accepts it. All incomplete Product Backlog Items are re-estimated and put back on the Product Backlog. The work done on them depreciates quickly and must be frequently re-estimated.
Sprint cancellations consume resources since everyone has to regroup in another Sprint Planning to start another Sprint. Sprint cancellations are often traumatic to the Scrum Team and are very uncommon.

Sprint Planning

The work to be performed in the Sprint is planned at the Sprint Planning. This plan is created by the collaborative work of the entire Scrum Team.
Sprint Planning is time-boxed to a maximum of eight hours for a one-month Sprint. For shorter Sprints, the event is usually shorter. The Scrum Master ensures that the event takes place and that attendants understand its purpose. The Scrum Master teaches the Scrum Team to keep it within the time-box.
Sprint Planning answers the following:

- What can be delivered in the Increment resulting from the upcoming Sprint?
- How will the work need to deliver the Increment be achieved?

Topic One: What can be done this Sprint?

The Development Team works to forecast the functionality that will be developed during the Sprint. The Product Owner discusses the objective that the Sprint should achieve and the Product Backlog items that, if completed in the Sprint, would achieve the Sprint Goal. The entire Scrum Team collaborates on understanding the work of the Sprint.
The input to this meeting is the Product Backlog, the latest product Increment, projected capacity of the Development Team during the Sprint, and past performance of the Development Team. The number of items selected from the Product Backlog for the Sprint is solely up to the Development Team. Only the Development Team can assess what it can accomplish over the upcoming Sprint.
After the Development Team forecasts the Product Backlog items it will deliver in the Sprint, the Scrum Team crafts a Sprint Goal. The Sprint Goal is an objective that will be met within the Sprint

through the implementation of the Product Backlog, and it provides guidance to the Development Team on why it is building the Increment.

Topic Two: how will the chosen work get done?

Having set the Sprint Goal and selected the Product Backlog items for the Sprint, the Development Team decides how it will build this functionality into a "Done" product Increment during the Sprint. The Product Backlog items selected for this Sprint plus the plan for delivering them is called the Sprint Backlog.

The Development Team usually starts by designing the system and the work needed to convert the Product Backlog into a working product Increment. Work may be of varying size or estimated effort. However, enough work is planned during Sprint Planning for the Development Team to forecast what it believes it can do in the upcoming Sprint. Work planned for the first days of the Sprint by the Development Team is decomposed by the end of this meeting, often to units of one day or less. The Development Team self-organizes to undertake the work in the Sprint Backlog, both during Sprint Planning and as needed throughout the Sprint.

The Product Owner can help to clarify the selected Product Backlog items and make trade-offs. If the Development Team determines it has too much or too little work, it may renegotiate the selected Product Backlog items with the Product Owner. The Development Team may also invite other people to attend in order to provide technical or domain advice.

By the end of the Sprint Planning, the Development Team should be able to explain to the Product Owner and Scrum Master how it intends to work as a self-organizing team to accomplish the Sprint Goal and create the anticipated Increment.

Sprint Goal

The Sprint Goal is an objective set for the Sprint that can be met through the implementation of Product Backlog. It provides guidance to the Development Team on why it is building the Increment. It is created during the Sprint Planning meeting. The Sprint Goal gives the Development Team some flexibility regarding the functionality implemented within the Sprint. The selected Product Backlog items deliver one coherent function, which can be the Sprint Goal. The Sprint Goal can be any other coherence that causes the Development Team to work together rather than on separate initiatives.

As the Development Team works, it keeps the Sprint Goal in mind. In order to satisfy the Sprint Goal, it implements the functionality and technology. If the work turns out to be different than the Development Team expected, they collaborate with the Product Owner to negotiate the scope of Sprint Backlog within the Sprint.

Sprint Review

A Sprint Review is held at the end of the Sprint to inspect the Increment and adapt the Product Backlog if needed. During the Sprint Review, the Scrum Team and stakeholders collaborate about what was done in the Sprint. Based on that and any changes to the Product Backlog during the Sprint, attendees collaborate on the next things that could be done to optimize value. This is an informal meeting, not a status meeting, and the presentation of the Increment is intended to elicit feedback and foster collaboration.

This is a four-hour time-boxed meeting for one-month Sprints. For shorter Sprints, the event is usually shorter. The Scrum Master ensures that the event takes place and that attendants understand its purpose. The Scrum Master teaches all to keep it within the time-box.

The Sprint Review includes the following elements:

- Attendees include the Scrum Team and key stakeholders invited by the Product Owner;
- The Product Owner explains what Product Backlog items have been "Done" and what has not been "Done";
- The Development Team discusses what went well during the Sprint, what problems it ran into, and how those problems were solved;
- The Development Team demonstrates the work that it has "Done" and answers questions about the Increment;
- The Product Owner discusses the Product Backlog as it stands. He or she projects likely completion dates based on progress to date (if needed);

- The entire group collaborates on what to do next so that the Sprint Review provides valuable input to subsequent Sprint Planning;
- Review of how the marketplace or potential use of the product might have changed what is the most valuable thing to do next; and,
- Review of the timeline, budget, potential capabilities, and marketplace for the next anticipated release of the product.

The result of the Sprint Review is a revised Product Backlog that defines the probable Product Backlog items for the next Sprint. The Product Backlog may also be adjusted overall to meet new opportunities.

Sprint Retrospective

The Sprint Retrospective is an opportunity for the Scrum Team to inspect itself and create a plan for improvements to be enacted during the next Sprint.

The Sprint Retrospective occurs after the Sprint Review and prior to the next Sprint Planning. This is a three-hour time-boxed meeting for one-month Sprints. For shorter Sprints, the event is usually shorter. The Scrum Master ensures that the event takes place and that attendants understand its purpose. The Scrum Master teaches all to keep it within the time-box. The Scrum Master participates as a peer team member in the meeting from the accountability over the Scrum process.

The purpose of the Sprint Retrospective is to:

- Inspect how the last Sprint went with regards to people, relationships, process, and tools;
- Identify and order the major items that went well and potential improvements; and,
- Create a plan for implementing improvements to the way the Scrum Team does its work.

The Scrum Master encourages the Scrum Team to improve, within the Scrum process framework, its development process and practices to make it more effective and enjoyable for the next Sprint. During each Sprint Retrospective, the Scrum Team plans ways to increase product quality by adopting the definition of “Done” as appropriate.

By the end of the Sprint Retrospective, the Scrum Team should have identified improvements that it will implement in the next Sprint. Implementing these improvements in the next Sprint is the adaptation to the inspection of the Scrum Team itself. Although improvements may be implemented at any time, the Sprint Retrospective provides a formal opportunity to focus on inspection and adaptation.

Scrum Artifacts

Scrum’s artifacts represent work or value to provide transparency and opportunities for inspection and adaptation. Artifacts defined by Scrum are specifically designed to maximize transparency of key information so that everybody has the same understanding of the artifact.

Product Backlog

The Product Backlog is an ordered list of everything that might be needed in the product and is the single source of requirements for any changes to be made to the product. The Product Owner is responsible for the Product Backlog, including its content, availability, and ordering.

A Product Backlog is never complete. The earliest development of it only lays out the initially known and best-understood requirements. The Product Backlog evolves as the product and the environment in which it will be used evolves. The Product Backlog is dynamic; it constantly changes to identify what the product needs to be appropriate, competitive, and useful. As long as a product exists, its Product Backlog also exists.

The Product Backlog lists all features, functions, requirements, enhancements, and fixes that constitute the changes to be made to the product in future releases. Product Backlog items have the attributes of a description, order, estimate and value.

As a product is used and gains value, and the marketplace provides feedback, the Product Backlog becomes a larger and more exhaustive list. Requirements never stop changing, so a Product Backlog is a living artifact. Changes in business requirements, market conditions, or technology may cause changes in the Product Backlog.

Multiple Scrum Teams often work together on the same product. One Product Backlog is used to describe the upcoming work on the product. A Product Backlog attribute that groups items may then be employed.
Product Backlog refinement is the act of adding detail, estimates, and order to items in the Product Backlog. This is an ongoing process in which the Product Owner and the Development Team collaborate on the details of Product Backlog items. During Product Backlog refinement, items are reviewed and revised. The Scrum Team decides how and when refinement is done. Refinement usually consumes no more than 10% of the capacity of the Development Team. However, Product Backlog items can be updated at any time by the Product Owner or at the Product Owner's discretion.
Higher ordered Product Backlog items are usually clearer and more detailed than lower ordered ones. More precise estimates are made based on the greater clarity and increased detail; the lower the order, the less detail. Product Backlog items that will occupy the Development Team for the upcoming Sprint are refined so that any one item can reasonably be "Done" within the Sprint time-box. Product Backlog items that can be "Done" by the Development Team within one Sprint are deemed "Ready" for selection in a Sprint Planning. Product Backlog items usually acquire this degree of transparency through the above-described refining activities.
The Development Team is responsible for all estimates. The Product Owner may influence the Development Team by helping it understand and select trade-offs, but the people who will perform the work make the final estimate.

Monitoring Progress Toward a Goal

At any point in time, the total work remaining to reach a goal can be summed. The Product Owner tracks this total work remaining at least every Sprint Review. The Product Owner compares this amount with work remaining at previous Sprint Reviews to assess progress toward completing projected work by the desired time for the goal. This information is made transparent to all stakeholders.
Various projective practices upon trending have been used to forecast progress, like burn-downs, burn-ups, or cumulative flows. These have proven useful. However, these do not replace the importance of empiricism. In complex environments, what will happen is unknown. Only what has happened may be used for forward-looking decision-making.

Sprint Backlog

The Sprint Backlog is the set of Product Backlog items selected for the Sprint, plus a plan for delivering the product Increment and realizing the Sprint Goal. The Sprint Backlog is a forecast by the Development Team about what functionality will be in the next Increment and the work needed to deliver that functionality into a "Done" Increment.
The Sprint Backlog makes visible all of the work that the Development Team identifies as necessary to meet the Sprint Goal.
The Sprint Backlog is a plan with enough detail that changes in progress can be understood in the Daily Scrum. The Development Team modifies the Sprint Backlog throughout the Sprint, and the Sprint Backlog emerges during the Sprint. This emergence occurs as the Development Team works through the plan and learns more about the work needed to achieve the Sprint Goal.
As new work is required, the Development Team adds it to the Sprint Backlog. As work is performed or completed, the estimated remaining work is updated. When elements of the plan are deemed unnecessary, they are removed. Only the Development Team can change its Sprint Backlog during a Sprint. The Sprint Backlog is a highly visible, real-time picture of the work that the Development Team plans to accomplish during the Sprint, and it belongs solely to the Development Team.

Monitoring Sprint Progress

At any point in time in a Sprint, the total work remaining in the Sprint Backlog can be summed. The Development Team tracks this total work remaining at least for every Daily Scrum to project the likelihood of achieving the Sprint Goal. By tracking the remaining work throughout the Sprint, the Development Team can manage its progress.

Increment

The Increment is the sum of all the Product Backlog items completed during a Sprint and the value of the increments of all previous Sprints. At the end of a Sprint, the new Increment must be “Done,” which means it must be in useable condition and meet the Scrum Team’s definition of “Done.” It must be in useable condition regardless of whether the Product Owner decides to actually release it.

Artifact Transparency

Scrum relies on transparency. Decisions to optimize value and control risk are made based on the perceived state of the artifacts. To the extent that transparency is complete, these decisions have a sound basis. To the extent that the artifacts are incompletely transparent, these decisions can be flawed, the value may diminish and risk may increase.

The Scrum Master must work with the Product Owner, Development Team, and other involved parties to understand if the artifacts are completely transparent. There are practices for coping with incomplete transparency; the Scrum Master must help everyone apply the most appropriate practices in the absence of complete transparency. A Scrum Master can detect incomplete transparency by inspecting the artifacts, sensing patterns, listening closely to what is being said, and detecting differences between expected and real results.

The Scrum Master’s job is to work with the Scrum Team and the organization to increase the transparency of the artifacts. This work usually involves learning, convincing, and change. Transparency doesn’t occur overnight but is a path.

Definition of "Done"

When a Product Backlog item or an Increment is described as “Done”, everyone must understand what “Done” means. Although this varies significantly per Scrum Team, members must have a shared understanding of what it means for work to be complete, to ensure transparency. This is the definition of “Done” for the Scrum Team and is used to assess when work is complete on the product Increment.

The same definition guides the Development Team in knowing how many Product Backlog items it can select during a Sprint Planning. The purpose of each Sprint is to deliver Increments of potentially releasable functionality that adhere to the Scrum Team’s current definition of “Done.” Development Teams deliver an Increment of product functionality every Sprint. This Increment is useable, so a Product Owner may choose to immediately release it. If the definition of "done" for an increment is part of the conventions, standards or guidelines of the development organization, all Scrum Teams must follow it as a minimum. If "done" for an increment is not a convention of the development organization, the Development Team of the Scrum Team must define a definition of “done” appropriate for the product. If there are multiple Scrum Teams working on the system or product release, the development teams on all of the Scrum Teams must mutually define the definition of “Done.”

Each Increment is additive to all prior Increments and thoroughly tested, ensuring that all Increments work together.

As Scrum Teams mature, it is expected that their definitions of “Done” will expand to include more stringent criteria for higher quality. Any one product or system should have a definition of “Done” that is a standard for any work done on it.

IMPORTANT POINTERS

FORCED RANKING:

- Forced ranking is a controversial workforce management tool that uses intense yearly evaluations to **identify a company's best and worst performing employees**, using person-to-person comparisons. In theory, each ranking will improve the quality of the workforce

- Forced ranking is a business tool for **managing limited resources**. This becomes especially useful in software projects for scheduling features and functions, prioritizing bugs, scheduling resources for work activities, selecting infrastructure and organizing testing and deployment activities
- All forced ranking of business requirements must be **business driven**
- Used in Scrum in **Backlog Grooming**
- It is also incorporated indirectly in the "minimum viable product" or MVP concept
- The benefits of forced ranking are largely for helping to manage limited resources towards achieving the essential parts of a system to meet primary goals

CONTINOUS INTEGRATION & DIFFERENT TOOLS USED FOR IT:

Continuous Integration (CI) is a software engineering practice in which isolated changes are immediately tested and reported on when they are added to a larger code base. The goal of CI is to provide rapid feedback so that if a defect is introduced into the code base, it can be identified and corrected as soon as possible. Continuous integration software tools can be used to **automate the testing** and **build a document trail**.

Tools:

- Jenkins - It is an open source automation server written in Java. Jenkins helps to automate the non-human part of the whole software development process, with now common things like continuous integration, but by further empowering teams to implement the technical part of a Continuous Delivery. It is a server-based system running in a servlet container such as Apache Tomcat. It supports SCM tools including AccuRev, CVS, Subversion, Git, Mercurial, Perforce, Clear case and RTC, and can execute Apache Ant and Apache Maven based projects as well as arbitrary shell scripts and Windows batch commands.

- Bamboo - It is a continuous integration server from Atlassian, the makers of JIRA, Confluence and Crowd. It is used to build, test and deploy applications automatically as per requirements and thus helps speed up the release process. Bamboo supports builds in a number of programming languages using various build tools and can also integrate with a large number of software for a variety of purposes. Bamboo is free for open-source projects. Commercial organizations are charged based on the number of build agents needed.

- Codeship – It is a continuous deployment solution that's focused on being an end-to-end solution for running tests and deploying apps.

BLOCKERS VS IMPEDIMENTS:

- An ***impediment*** is anything that slows down or diminishes the pace of the Team. When the Team is confronted with impediment (or obstacles), the Team could move forward but in advancing they may generate waste. Or the whole process of making progress is more difficult than it should be (think of the little girl in the picture).
- In contrast, a ***blocker*** is anything that stops the delivery of the product. Without the elimination of the blocker, the Team cannot advance at all. Clearly, eliminating blockers is more important than resolving impediments

USER STORY MAPPING:

A user story map arranges user stories into a useful model to help understand the functionality of the system, identify holes and omissions in your backlog, and effectively plan holistic releases that deliver value to users and business with each release

https://www.scrumalliance.org/community/articles/2013/august/creating-an-agile-roadmap-using-story-mapping

FEATURE TEAMS:

- A feature team is a long-lived, cross-functional, cross-component team that completes many end-to-end customer features—one by one
- The characteristics of a feature team are listed below:
  - long-lived—the team stays together so that they can 'jel' for higher performance; they take on new features over time
  - cross-functional and cross-component
  - ideally, co-located
  - work on a complete customer-centric feature, across all components and disciplines (analysis, programming, testing, …)
  - composed of generalizing specialists
  - in Scrum, typically 7 ± 2 people

- A common misunderstanding: every member of a feature team needs to know the whole system. Not so, because:
  - The team as a whole—not each individual member—requires the skills to implement the entire customer-centric feature. These include component knowledge and functional skills such as test, interaction design, or programming. But within the team, people still specialize… preferably in multiple areas.
  - Features are not randomly distributed over the feature teams. The current knowledge and skills of a team are factored into the decision of which team works on which features

THEMES, FEATURES, EPICS & USER STORIES:

- **Themes** may be thought of as groups of related stories. Often the stories all contribute to a common goal or are related in some obvious way, such as all focusing on a single customer. However, while some stories in a theme may be dependent on one another, they do not need to encapsulate a specific work flow or be delivered together
- "Theme" is a collection of user stories. We could put a rubber band on that group of stories I wrote about monthly reporting and we'd call that a "theme." Sometimes it's helpful to think about a group of stories so we have a term for that. Sticking with the movie analogy above, in my DVD rack I have filed the James Bond movies together. They are a theme or grouping.

- A **feature** is a distinct element of functionality which can provide capabilities to the business. It generally takes many iterations to deliver a feature. A user story is a part of the feature. By splitting a feature in smaller stories, the user can give early feedback to the developers to issues quickly.

- **Epics** resemble themes in the sense that they are made up of multiple stories. They may also resemble stories in the sense that, at first, many appear to simply be a "big story." As opposed to themes, however, these stories often comprise a complete work flow for a user. But there's an even more important difference between themes and epics. While the stories that comprise an epic may be completed independently, their business value isn't realized until the entire epic is complete. This means that it rarely makes sense to deliver an epic until all of the underlying stories are complete. In contrast, while stories comprising a theme are related, each is independent enough to be delivered separately and still provide some measurable sense of business value

- **User Stories:**

- It is a **software system requirement** formulated as **one or two sentences** in the everyday or business language of the user
- Each story is limited; so it fits on a **3x5in card**

- When are User Stories written?
  - **Throughout the Agile project.** Usually, story-writing workshop is held near the start of the agile project
  - **Everyone on the team** participates with the goal of **creating a product backlog** that **fully describes the functionality** to be added over the course of the project or a 3-6 months release cycle within
  - Some of these agile user stories will undoubtedly be epics. **Epics** will later be **decomposed into smaller stories** that fit more readily into a **single iteration**. Additionally, *new stories can be written & added to the product backlog at any time by anyone*

- How to write User Stories?
  - A user story briefly explains:
    - the person using the service **(actor)**
    - what the user needs the service for **(narrative)**
    - why the user needs it **(goal)**
  - As a <type of user>, I want <some goal> so that <some reason>

PBIs vs TASKS:

The PBI:
- is the requirement aka "the what"
- is what you talk about with a customer.
- It's what shows up on the Daily Project Update (DPU) for the sprint..... again because the DPU is customer facing.
- It's what the customer will talk about and refer in terms of estimates and budget.
- Might comprise one or more tasks.
- Is business oriented and described in business oriented / domain style language that the customer understands.
- Is what gets tested and accepted in User Acceptance Testing (UAT)

The Task:
- Is a piece of work required to materialize the PBI (requirement)
- Not something you talk about with a customer
- Doesn't show up on the DPU because you don't talk about them with customers
- Is estimated but has its estimates rolled up into the PBI
- Is a child to one and only one requirement.
- Can be described (and often is) using technical jargon
- Tested internally and signed off by test
- Not accepted or tested in isolation by the customer (they don't know they exist)

TEST DRIVEN DEVELOPMENT (TDD):

- "Test-driven development" refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).

- It can be succinctly described by the following set of rules:
  - write a "single" unit test describing an aspect of the program
  - run the test, which should fail because the program lacks that feature
  - write "just enough" code, the simplest possible, to make the test pass
  - "refactor" the code until it conforms to the simplicity criteria
  - repeat, "accumulating" unit tests over time

- Expected Benefits:
  - many teams report significant reductions in defect rates, at the cost of a moderate increase in the initial development effort
  - the same teams tend to report that these overheads are more than offset by a reduction in effort in projects' final phases
  - although empirical research has so far failed to confirm this, veteran practitioners report that TDD leads to improved design qualities in the code, and more generally a higher degree of "internal" or technical quality, for instance improving the metrics of cohesion and coupling

- Common Pitfalls:
  - Typical individual mistakes include:
    - forgetting to run tests frequently
    - writing too many tests at once
    - writing tests that are too large or coarse-grained
    - writing overly trivial tests, for instance omitting assertions
    - writing tests for trivial code, for instance, accessors
  - Typical team pitfalls include:
    - partial adoption - only a few developers on the team use TDD
    - poor maintenance of the test suite - most commonly leading to a test suite with a prohibitively long running time
    - abandoned test suite (i.e. seldom or never run) - sometimes as a result of poor maintenance, sometimes as a result of team turnover

ADD (ACCEPTANCE TEST DRIVEN DEVELOPMENT):

- Analogous to test-driven development, Acceptance Test Driven Development (ATDD) involves team members with different perspectives (customer, development, testing) collaborating to write acceptance tests in advance of implementing the corresponding functionality. The collaborative discussions that occur to generate the acceptance test are often referred to as the three amigos, representing the three perspectives of the customer (what problem are we trying to solve?), development (how might we solve this problem?), and testing (what about…).
- These acceptance tests represent the user's point of view and act as a form of requirements to describe how the system will function, as well as serve as a way of verifying that the system functions as intended. In some cases, the team automates the acceptance tests

Also Known As

- ATDD may also be referred to as Story Test Driven Development (SDD), Specification by Example or Behavior Driven Development (BDD). These different terms exist to stress some differences in approach that lead to similar outcomes.

Expected Benefits

- Just as TDD results in applications designed to be easier to unit test, ATDD favours the creation of interfaces specific to functional testing. (Testing through an application's actual UI is considered less effective.)

Common Pitfalls

- Even more than the use of automated acceptance tests, this practice is strongly associated with the use of specific tools such as **Fit/FitNess, Cucumber** or others.
- One major risk, therefore, is that the tool chosen will hinder rather than advance the main purpose of this practice: facilitating conversation between developers and product owners about product requirements. Tools should be adapted to meet product owners' needs rather than the other way around.

BDD (BEHAVIOR DRIVEN DEVELOPMENT):

Behavior Driven Development (BDD) is a synthesis and refinement of practices stemming from Test Driven Development (TDD) and Acceptance Test Driven Development (ATDD). BDD augments TDD and ATDD with the following tactics:

- Apply the "Five Why's" principle to each proposed user story, so that its purpose is clearly related to business outcomes
- thinking "from the outside in", in other words, implement only those behaviours which contribute most directly to these business outcomes, so as to minimize waste
- describe behaviours in a single notation which is directly accessible to domain experts, testers and developers, so as to improve communication
- apply these techniques all the way down to the lowest levels of abstraction of the software, paying particular attention to the distribution of behaviour, so that evolution remains cheap

Expected Benefits

- Teams already using TDD or ATDD may want to consider BDD for several reasons:
- BDD offers more precise guidance on organizing the conversation between developers, testers and domain experts
- notations originating in the BDD approach, in particular, the given-when-then canvas, are closer to everyday language and have a shallower learning curve compared to those of tools such as Fit/FitNesse
- tools targeting a BDD approach generally afford the automatic generation of technical and end user documentation from BDD "specifications"

Common Pitfalls

- Although Dan North, who first formulated the BDD approach, claims that it was designed to address recurring issues in the teaching of TDD, it is clear that BDD requires familiarity with a greater range of concepts than TDD does, and it seems difficult to recommend a novice programmer should first learn BDD without prior exposure to TDD concepts
- The use of BDD requires no particular tools or programming languages, and is primarily a conceptual approach; to make it a purely technical practice or one that hinges on specific tooling would be to miss the point altogether

APPROACHES TO RELEASE PLANNING:

A **very high-level plan for multiple Sprints (e.g. three to twelve iteration)** is created during the Release planning. It is a guideline that reflects expectations about *which features will be implemented and when they are completed*. It also serves as a *base to monitor progress within the project*. Releases can be intermediate deliveries done during the project or the final delivery at the end

- Date Driven
  - When you calculate the amount of work you can finish by a particular date
- Feature Driven
  - When you calculate how long it will take to finish a specific amount of work

FORMAT FOR USER STORY & ACCEPTANCE CRITERIA:

- *Acceptance Criteria as "Conditions that a software product must satisfy to be accepted by a user, customer or another stakeholder."* Google defines them as "Pre-established standards or requirements a product or project must meet."

- Acceptance Criteria are a set of statements, each with a clear pass/fail result, that specifies both functional (e.g., minimal marketable functionality) and non-functional (e.g., minimal quality) requirements applicable at the current stage of project integration. These requirements represent "conditions of satisfaction." There is no partial acceptance: either a criterion is met or it is not.

- These criteria define the boundaries and parameters of a User Story/feature and determine when a story is completed and working as expected. They add certainty to what the team is building.

- Acceptance Criteria must be expressed clearly, in simple language the customer would use, just like the User Story, without ambiguity as to what the expected outcome is: what is acceptable and what is not acceptable. They must be testable: easily translated into one or more manual/automated test cases.
-
- Acceptance Criteria may reference what is in the project's other User Stories or design documents to provide details, but should not be a re-hash of them. They should be relatively high-level while still providing enough detail to be useful. They should include:
  - Functional Criteria: Identify specific user tasks, functions or business processes that must be in place. A functional criterion might be "A user is able to access a list of available reports." A non-functional criterion might be "Edit buttons and Workflow buttons comply with the Site Button Design."
  - Non-functional Criteria: Identify specific non-functional conditions the implementation must meet, such as design elements. A non-functional criterion might be "Edit buttons and Workflow buttons comply with the Site Button Design."
  - Performance Criteria: If specific performance is critical to the acceptance of a user story, it should be included. This is often measured as a response time, and should be spelt out as a threshold such as "

- Acceptance Criteria should state intent, but not a solution (e.g., "A manager can approve or disapprove an audit form" rather than "A manager can click an 'Approve/Disapprove' radio button to approve an audit form"). The criteria should be independent of the implementation: ideally, the phrasing should be the same regardless of the target platform.

PSEUDO SOLUTIONS:

- Pseudo Solutions are the solutions that were initially developed for a different PBI or Task but can be applied to another one
- For Eg, if a PBI has two tasks, and the development of one task leads to the solution of another
- Pseudo Solutions may or may not be beneficial to the team

WATERFALL vs SCRUM:

WATERFALL-SCRUM HYBRID:

This hybrid approach, however, is for coexistence sake; to be able to satisfy both processes for the foreseeable life of the project.

The hybrid Scrum methodology proceeds as follows:

- **Release Planning** - a hybrid process element - is a series of planning sessions conducted prior to each release cycle. Planning is initially accomplished with IEEE 830 requirements only.
- **Predefined Release Schedule** - a hybrid artifact - is a regularly refined product of each Release Planning. At a high level, this shows the entire release schedule for the project.
- **Product Backlog** - a hybrid artifact - is a change to the normal Scrum Product Backlog. It is populated with IEEE 830 Requirements, and it is groomed each release cycle through the Release Planning.
- **User Story Workshop** - a normal Scrum process element.
- **Release Backlog** - a hybrid artifact - is an additional backlog which is used to define the scope of the current release. It is populated by the Scrum Team and PMO in User Story Workshops, and it serves the purpose that a Product Backlog serves in
- normal Scrum.
- **Sprint Planning Session** - a normal Scrum process element.
- **Sprint Planning Review Artifact** - a hybrid artifact - is a product of the Sprint Planning Session which captures the scope of the current sprint which was just captured
- **Sprint Backlog** - a normal Scrum process element.
- **Sprint and Daily Scrums** - are normal Scrum process elements.
- **Traceable UAT Artifact** - a hybrid artifact - is a clearly defined UAT for each user story which is used to test features and trace their verification from the signed UAT back to the IEEE 830 requirements which they satisfy.
- **Sprint Review** - a normal Scrum process element.
- **Sprint Review Artifact** - a hybrid artifact - is a product of the Sprint Review which captures the satisfactory accomplishment and acceptance of each of the stories in the sprint, with traceability back to original IEEE 830 requirements.
- **Shippable Product** - a normal Scrum process element

APPROACHES TO SPRINT PLANNING:

- Velocity driven Sprint Planning
- Commitment driven Sprint Planning

DEFINITION OF "DONE" & "READY":

Preparing a single definition of done that suits every situation is impossible. Each team should collaborate and come up with the definition that suits its unique environment.
Organizations that have just started with agile may find it difficult to reach a mature level immediately; therefore, they should take the steps, sprint-by-sprint, to improve their done definition. Feedback from retrospective meetings should also help to improve the definition of done.
- See more at https://www.scrumalliance.org/community/articles/2008/september/definition-of-done-a-reference#sthash.QaaI113G.dpuf

SPRINT "0":

Sprint zero is usually claimed as necessary because there are things that need to be done before a Scrum project can start. For example, a team needs to be assembled. That may involve hiring or moving people onto the project. Sometimes there is hardware to acquire or at least setup. Many

projects argue for the need to write an initial product backlog (even if just at a high level) during a sprint zero.

MINIMUM VIABLE PRODUCT (MVP):

A minimum viable product (MVP) is a development technique in which a new product or website is developed with sufficient features to satisfy early adopters. The final, complete set of features is only designed and developed after considering feedback from the product's initial users.
A minimum viable product (MVP) is the most pared down version of a product that can still be released. An MVP has three key characteristics:

- It has enough value that people are willing to use it or buy it initially
- It demonstrates enough future benefit to retain early adopters
- It provides a feedback loop to guide future development

ESTIMATION CHARTS:

- Sprint Burndown Chart
- Release Burndown Chart
- Release BurnUp Chart

TYPES OF SPIKE:

SCRUM FOR DATA WAREHOUSING PROJECTS:

Agile approach to data warehousing solves many of the thorny problems typically associated with data warehouse development—most notably high costs, low user adoption, ever-changing business requirements and the inability to rapidly adapt as business conditions change. The Agile approach can be used to develop any analytical database

- Highly desirable to be done in an evolutionary manner
- Need to be flexible
- Collaboration is crucial

- Design Data Models using simple language like "Who does What"?
- Ask business users for specific examples so you know how to build the data model
- Use test data from the users, and conduct ETL on the data
- High-Level Modeling performed early in the lifecycle
- Light weight modelling throughout the lifecycle
- Comprehensive regression test suites are developed
- Solution is developed in thin vertical slices
- Requirements changes are welcome

- User Stories needs to drive Scrum development for DW, not data
- Get to know what data and how do users use the data
- How are we going to use the information?
- Scrum DW teams must deliver new data or reports each iteration
- Agile DW teams strive to fix the data at the source itself via data refactoring
- Agile DW is Pragmatic
    - Sometimes, you need to accept data inconsistencies
    - You cannot get all data you need for a report

Agile Database Technique Stack (Important):

- Vertical Slicing
    - Slicing Strategies:

    - One new data element from a single data source
    - One new data element from several sources
    - Change to existing report
    - New report
    - New reporting view
    - New Data mart Table
- Clean Architecture and Design
    - Cleaner the design of the database, easier it is to evolve
- Agile Data Modeling
    - Gather details, make decisions in an iterative way
- Database Refactoring
    - Simple change to database schema that improves its design while retaining both its behavioural and informational semantics
- Database Regression Testing
    - Test Driven Database Development
    - Behavior Driven Database Development
- Continuous Database Integration
- Configuration Management

QUESTIONS

- When do you draw a line for scope creep?
- Consider yourself as a scrum master:
    - How long is your project - 1 year
    - How many sprints in last project - 22
    - How many user stories completed - 150
    - Story point completed - 704 (average: 5)
    - Velocity of the project - 32
    - Velocity of 1st 2nd 3rd - 23, 27, 30
    - Team size - 7
    - Duration of each sprint - 2 weeks
    - Highest velocity - 35
    - For many sprints, you had that high velocity - 15th and 16th
    - Effort hour - 42/day
    - How many effort hours for one story point - 13 effort hours
- Can you tell me a user story from your last project?
- How many modules , release functionality in your sprints?
- How did you work on these releases?
- If a developer leaves in the middle of the sprint, what will you do as a scrum master? How will you avoid it in the future?
- How do you handle the elements of dysfunction in your team as a scrum master? (5, Lack of fear, commitment, trust)
- How do you manage risks in the scrum? (Impediment list)
- Sprint started on Wednesday, today is Tuesday, if the user stories not ready, what will you do as a Scrum Master?
- Suppose PO is not available in the sprint planning meeting. What do you do as a scrum master? (Being well informed with the product owner)
- Will you have the sprints back to back?
- If you have a conflict, how will you handle?

- How will you handle the when people disagree within the team? (Reciprocity)
- Agile project manager - manages budget, schedule, resources - who actually manage the scrum on the whole.
- Rally- SAFE, JIRA -Scrum
  - How would you help the team's members find their role within the team?
  - How would you help the team to find a strong sense of purpose?
  - How would you help the team to maintain ideal Scrum Ceremonies?
  - How would you help the team to act as a team, and not just a bunch of individuals?
  - How would you help the team to Surface and Resolve Conflicts?
  - How would you help each individual to get better at what they do?
  - How would you help the team to raise the bar, so as to help them to continuously improve?
  - How would you help the team to identify possible improvements in their Engineering practices?
  - How would you help the team to establish effective team communication skills?
  - How would you encourage self-organization within your team?
  - Are you aware of the 5 dysfunctions of a team, and can you create exercises to improve them?
    - Absence of Trust
    - Fear of Conflict
    - Lack of Commitment
    - Avoidance of Accountability
    - Inattention to Results
  - How will you help the team to spread knowledge gained during development?
    - Knowledge sharing meet-ups
    - Knowledge transfer charts- Self rated
    - Who has shared the most? Reward at the end of the every few sprints
  - How would you coach the whole team to collaborate? (Give examples…)
  - Can you give examples that show that you allow the team to find their own answers instead of you giving them?
  - Are you focused on business value delivery? Give me examples of how you accomplish this.
  - How would you help and encourage everyone to express their opinions?
    - Silent Writing
    - Open sessions
    - Networking outside of work if possible
  - How would you help the team in identifying positive and negative changes during retrospectives?
  - How would you encourage team learning (fostering collaborative practices, pair programming, continuous integration, collective code ownership, short design sessions, specifications workshops, etc.)?
  - How would you encourage rotation in technical areas of concern: functionality, components/layers, roles, aspects, etc.?
    - Encourage the team to pick up PBIs that have varying degrees of these various components
  - How would you help the team in setting long term operating goals for all team members: agile practices to master, new skills to acquire, etc.?
    - Roadmap for every team member- 2 components: Internal & External
  - How would you examine what is missing in order to make the environment better for everyone?
    - Retrospective, Daily Stand-Ups
    - Happiness Histogram
  - How would you prioritize improvement activities and make them happen?

  - How would you help the team to find access to external sources of information?
    - Research

    - Meet-Ups
    - Networking
    - Training on skills they want to build upon
- How would you help the team to find techniques to help them be more collaborative?
    - Outdoor activities
    - Team activities
- How would you encourage and facilitate open communication among the team members and with external colleagues?
    - Networking
    - Parties
    - Discuss with various Scrum Masters during Scrum of Scrum
- How would you help and encourage healthy conflict during team meetings?
    - Get team members to know each other on a personal level
    - Have an open table discussion to throw in ideas
    - Reward people recommending improvements while building ideas upon ideas
- How would you help the team to mature their DoD?
    - When you create your Definition of Done, think about all of the tasks that must be done in order to put the story into production. Be imaginative and really include everything, even tasks that might be out of your own team control. Go for it, and include everything!
    - When you are done, you will get what I call the “Future Definition Of Done”. This is the most ambitious DoD that you will create.
    - The next step is to create a new Definition of Done. This is the one that will be used by the team. In order to build this DoD, just pick up the topics from the “Future Definition of Done” that the team can tackle at that moment. Most probably, this will be a reduced list of what you created before. Do not worry, that is fine. When you finish, you will have two different DoDs.
    - Now that you have two Definitions of Done, your job is to help the team pick a new topic from the Future DoD. This can be achieved, for example, in the Agile Retrospectives. From time to time, you can use your Agile Retrospective to ask the team, “What is the new topic they would like to include on the current DoD?”
    - At some point, you will be able to move all the topics from the Future DoD to the current DoD. At that point, you should be proud because you are a very mature team
- How would you help the team create transparency and urgency around continuous system integration?
- How would you help encourage the team to create small automated acceptance tests at the beginning and evolve from there?
    - Develop the culture of measuring success
    - What KPIs?
    - Encourage team to share each other's success
    - Point system for every time the PO says that the item is "Done"
- How would you help encourage the team to coach each other in TDD, refactoring, and simple design?

- How would you help encourage the team to use human-readable acceptance tests?
    - Does the PO understand it?
    - Get team members to present parts of the demo during Sprint Review, where they get to explain every item that is shipped, and why they think they are ready
- How would you help encourage the team to do pair and peer reviews?
    - 360degree review
    - Reward System
    - Member of the Sprint reward

SAFe

- Works at a Portfolio, Program and Team level
- Various Scrum Teams work to deliver a PSPI every Sprint. Sprint for the Program level is 5 weeks long, by default
- 4 sprints are planned. At the end of each Sprint is a special 5th sprint called HIP Sprint (Hardening, Innovation, Planning)
  - H: Final verification to make sure we have met PSI. Also, do testing
  - I: Time for teams to engage in Hackathons, explore creative ideas
  - P: Demo accomplishments as PSI, maintenance of RT, plan next PSI together
- **2 Days PI Planning** can also happen here during the 5th Sprint
- Architectural Runway: Each PSI lays down tracks for planning architecture
- Agile **Release Train (RT)** are people at the Program level and the Team level
- Content for each PSI managed by Product Manager (Program level) in the Program Backlog
- Train governed by Release Train Engineer (RTE) at the Program Level
- Teams capture dependencies between themselves on the Program Board
- **The biweekly meeting** between Scrum Masters and RTE to review progress. Also System Demo for all teams
- Program Level teams can be 50-125 people
- Team ------> Program ------> Value stream ------> Portfolio
- Value Stream only exists if there are more than 125 people (Mostly works around budget and resource allocation)
  - Value Stream Engineer (Scrum Master)
  - Solution Manager
  - Solution Architect
- Roles at Program level in SAFE
  - Product Manager
  - RTE
  - System Architect
  - Business Owner
  - Systems Team
- **Program Increment (PI)** is usually **10 weeks** (can be 8-12) long. It is at **Program Level** and **Value Stream Level**
  - Program increment is to Agile Release Train (or Value Stream) as an Iteration is to the Agile Team
  - Each PI is a development timebox that uses cadence and synchronization to facilitate planning, limit WIP, etc.
- At Portfolio level, big initiatives are called Epics. These are managed on Program Boards, which are in Kanban format
  - Enterprise Architect
  - Epic Owners
  - Program Portfolio Management
- Once approved on the Kanban board, these Epics move to and are managed in a **Portfolio Backlog**
- The next is the **Value Stream Backlog** which has **Capabilities**
- Then is the **Program Backlog**, which has **Features**
- **Program Increment (PI) Planning Meeting** is a meeting where all 125 people come together and discuss and negotiate work that can be done in the next 10 weeks. Usually 2-day long meeting
- Once PI Planning Meeting is done, each team has **PI Objectives**. Not a detailed plan, but only features

- Then comes the **Team Level Backlog**, which contains all the work that needs to be completed to achieve the PI Objective
- 3.0 has 3 levels and 4.0 has 4 levels (Value Stream added)
- **Scrum of Scrums (SoS)** happen of ScrumMasters, Product Managers, etc. which is to inspect and adapt
- Impediments, issues also are discussed
- **Release Management Meeting** focuses on commercial releases

- **System Teams** pick up the increment from team's every Sprint and integrate it together to show it to the Program Management level, which is called a **Demo able Increment of Systems**. The ceremony is called **System Demo.** This gives a feel of integrated increment
- At the **end of 2 weeks Sprint**, SaFe recommends having one integrated system demo from the Program team

- At Value Stream Level, there are **Solution Demos**, where multiple ARTs work is integrated and presented together
- At VS Level, there is Pre-PI Planning and Post-PI Planning meeting