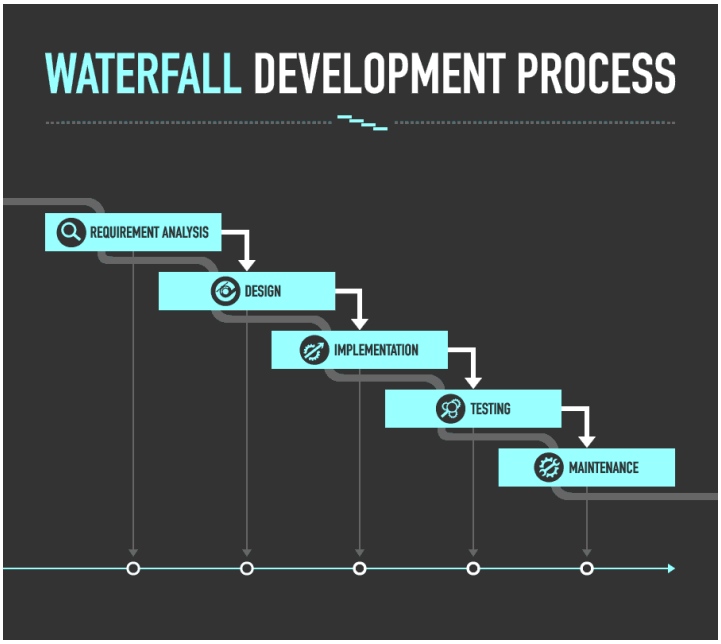
Agile and Scrum

History

- Throughout the history of the Software engineering there were several models of software development.

Waterfall model

- Created in the 1970's



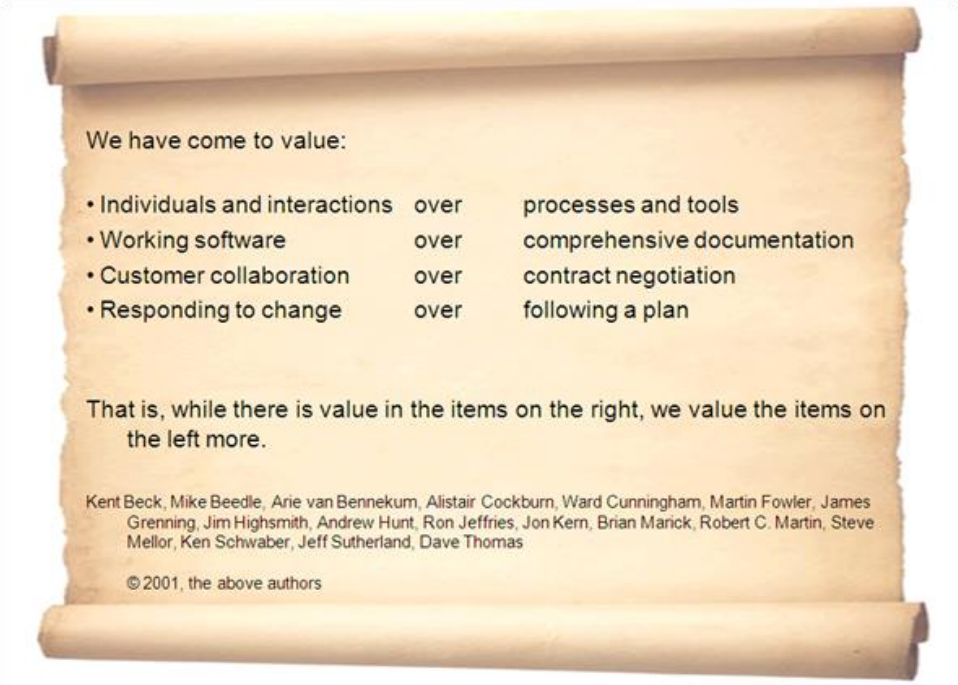
- Pros:
  - Many design errors are identified early
  - Measuring progress is easy
  - Accurate cost estimates can be provided
- Cons:
  - Change is difficult
  - Projects often run late
  - It assumes you know all requirements

Iterative development

- Created in the 1990's
- Requirements are implemented and tested separately and then presented to the costumer, he can object to some aspects and then change can be made quickly and cheaply.

Agile

- In 2002 the agile manifesto was created:



Agile Principles

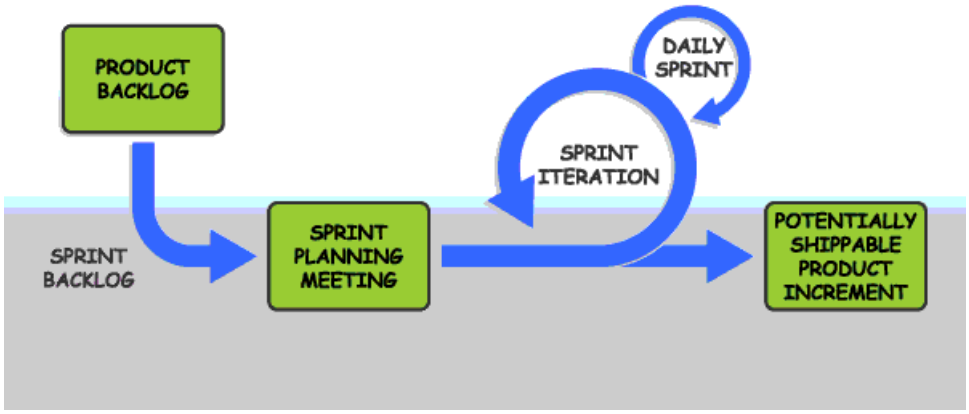
The Agile Manifesto also describes twelve principles of agile development:

- Provide customer satisfaction through early and continuous software delivery
- Accommodate changing requirements throughout the development process
- Supply frequent delivery of working software
- Collaborate between the business stakeholders and developers throughout the project
- Support, trust, and motivate the people involved
- Enable face-to-face interactions
- Measure progress through working software
- Use Agile processes to support a consistent development pace
- Enhance agility through attention to technical detail and design
- Keep things simple by developing just enough to get the job done for right now
- Self-organize teams to encourage great architectures, requirements, and designs
- Reflect regularly on how to become more effective

Summary of Agile frameworks

Scrum

- A team works in short cycles (called *sprints*) that are typically 2-4 weeks long.
- A prioritized list of requirements called the *product backlog* is created.
- Before each sprint, a number of features are chosen from the product backlog to be part of the cycle. The team will choose a list of features they believe they can complete during that sprint.
- Each day, the team meets briefly in a *stand-up* meeting to discuss progress.
- At the end of the sprint, the completed work should be in a state that is ready for release. The team then reviews the sprint and reflects on what they have learned and what they can improve upon for the next cycle.



Kanban

Kanban is an Agile methodology that encourages **flow** and seeks to keep work items from being stuck, blocked, or delayed.

The idea is that the team works on fewer items at a time focusing on reducing the time spent on each stage of development. This way there is not a lot of time between when tasks or features start and finish.

Implementation guidlines:

- **Limit "works in progress."** The key to Kanban is to limit the number of items in development (tasks or features), not so that you do less but rather that you start and complete more items.
- **Visualize workflow.** Put all work items on a wall and use columns to denote their status. This allows the team and the whole office to see the progress.
- **Manage flow.** By analyzing the point at which items get stuck, blocked, or slowed down you can identify (and then remove) bottlenecks.
- **Improve collaboratively.** Continuous improvement and teamwork are vital concepts in Kanban

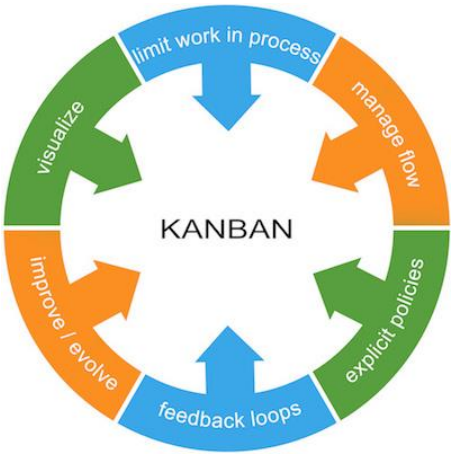
Extreme Programming (XP)

Extreme Programming or XP is an Agile framework that focuses a lot on the quality of practice and the habits of the software practitioner (i.e., the developers on the team). Its main guidelines are as follows:

- Developers will adhere to coding standards, all writing code the same way.
- Use test-driven development. This is a process where developers write the code for a test that a feature should pass (or validate) before proceeding. It is a key part of XP.
- Developers write code in pairs. Usually, one developer writes the test code and the other writes feature code.
- Work is done in short iterations (usually two weeks) and planning happens before each iteration.
- Just enough design and architecture are involved to build the features for the current iteration.
- Code is frequently checked against the master code base so errors can be instantly detected (this is called continuous integration of code with the code base).

Kanban in detail

Key Elements of Kanban



There are 6 key elements in Kanban:

- **Limit Work In Progress** - Rather than working on 20 features at the same time, it is better to focus on starting, finishing, and releasing a smaller number (say 5). With fewer items, you spend less time producing them. You can finish quicker than when you focus on a large number of items simultaneously. The time from beginning a feature, or a task, to its completion, is reduced and customers see value sooner.
- **Manage Flow** - Flow is the concept of moving items quickly through the system. Teams manage their flow by tracking how long an item spends in each possible status (how long it is waiting to be tested, how long before someone starts work on it, etc). Reducing these times becomes a goal for the team.
- **Feedback Loops** - Taking moments to reflect on what is working and to consider improvements is vital in Kanban.
- **Improve/Evolve** - Continuous improvement is a core concept in Kanban. A culture of running experiments and trying to improve and evolve is essential.
- **Visualize** - The team is empowered to make effective decisions when the work is visible using a board which shows all of the status, bottlenecks, and system limits
- **Explicit Policies** - You can't improve something that you don't understand. Any policy that relates to status updates enables the team to effectively decide if a given task is suitable to be worked on and when the status has changed.

Kanban Board

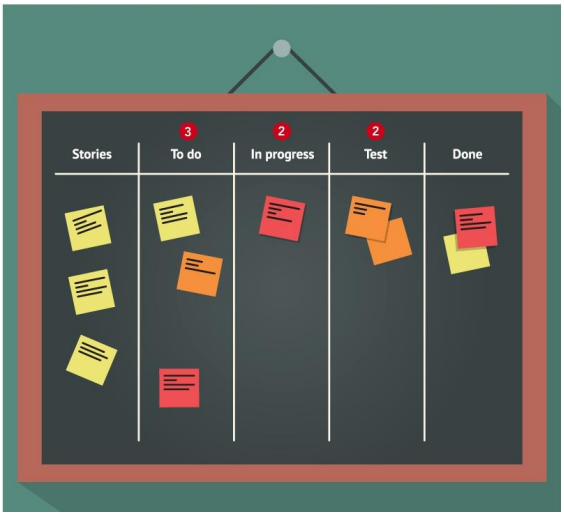
Perhaps the most important factor in Kanban's effectiveness is that all work items and their statuses are shown on a board.

You can check the status of an item by seeing which column it's in. Teams decide which columns make sense for their workflow. Therefore, a Kanban board can vary from team to team.

When the status of an item changes, a team member will move it into another column. For example, if you decided to begin working on an item called "New login button," then you would move its card from the "To Do" column to the "In Progress" column.

At the very least, you can expect a column/status for:

- Tasks "To Do" (i.e., not started)
- Tasks "in Progress"
- Tasks that are "Ready to Test"
- Tasks that are "Done"



Note that limits are shown in red circles above certain columns. For example, a "3" above the "To Do" column indicates that only three items can be in that particular section at one time. The "In Progress" column is limited to two items.

Kanban table examble

Scrum in detail

Theory and Values Behind the SCRUM Framework

What is SCRUM

SCRUM is an Agile methodology and process framework that can be applied to the effective building of complex products (such as software, websites, or apps).

This framework details:

- Member roles (i.e., which type of team members are in the SCRUM team as well as their responsibilities)
- Certain artifacts that team members must participate in creating
- A set of rules for how work is to be done

As the set of rules is clearly defined, an environment is created where each team member knows their responsibilities and everybody outside the team can easily understand what progress is being made.

The SCRUM Guide Definition

The SCRUM Guide is a document written by Ken Schwaber and Jeff Sutherland (the inventors of SCRUM) that outlines all the rules of SCRUM. The SCRUM Guide describes SCRUM as follows:

- Scrum is a process framework that has been used to manage work on complex products since the early 1990s. Scrum is not a process, technique, or definitive method. Rather, it is a framework within which you can employ various processes and techniques. Scrum makes clear the relative efficacy of your product management and work techniques so that you can continuously improve the product, the team, and the working environment.
- The Scrum framework consists of Scrum Teams and their associated roles, events, artifacts, and rules. Each component within the framework serves a specific purpose and is essential to Scrum's success and usage.
- The rules of Scrum bind together the roles, events, and artifacts, governing the relationships and interaction between them.

SCRUM as a Framework

As mentioned above, SCRUM is a framework and not a method for effective software delivery and building products. The distinction between framework and method is an important one in SCRUM.

The Collins Dictionary gives two definitions for a framework:

- A framework is a particular set of rules, ideas, or beliefs which you use in order to deal with problems or to decide what to do.

and

- A framework is a structure that forms a support or frame for something.

Both definitions can help you to understand SCRUM. Sticking to a set of rules and beliefs will help you work effectively, but it can't tell you every step to follow when building a product. Rather, it is a guideline that each team will have to apply to their own circumstances. For example, if you are working with a team that is based in multiple locations, then your daily update calls may be at different times of the day. The method in which developers review each other's code could also be adapted to virtual teams.

Empirical Process Control

The SCRUM Guide tells us that:

Scrum is founded on empirical process control theory, or empiricism. Empiricism asserts that knowledge comes from experience and then making decisions based on what is known. Scrum employs an iterative, incremental approach to optimize predictability and control risk.

It can be helpful to think of a factory production line when trying to understand empirical process control. In a factory, you have:

- Inputs
- Process
- Outputs

In order to understand when the work being done in the factory is effective, you typically **inspect** the outputs to see if they are high quality.

If they are not, then you would typically **inspect** *y*our inputs and process to see if they can be improved. If you can improve them, then make the necessary **adaptations***,* run the process again, and then **inspect** your new outputs. In this way, you become effective over time.

Note that in order for this to work the process must be **transparent** and observable by anyone.

As you saw in context above, the three pillars of empirical process control are:

- Transparency
- Inspection
- Adaptation

The values behind SCRUM are heavily influenced by the three pillars of Empirical Process Control.

The Values Behind SCRUM

These values are:

- **Commitment** - The SCRUM team as a whole will themselves define what is possible within a set time period (say a two week period). Each member of the team then commits to delivering on this promise. They also commit to helping each other and working as a team to deliver the best possible value to the customer.
- **Openness** - Being transparent requires openness and honesty. In particular, when a team member faces challenges in their work, it is important to be open about that so the process can be inspected and improved. Being an effective team member sometimes means asking for help when needed.
- **Courage** - Being transparent in your work also requires courage. There are times when inspecting the process means that people have made mistakes or have been wrong about something. Admitting this usually endears respect from colleagues, but it is never easy to admit you are wrong or that you have made a mistake.
- **Focus** - Each team will commit to focusing on and delivering a set of outcomes. There may be plenty of work that you can do which is valuable but not if it is at the expense of the team commitments .
- **Respect** - It is normal for there to be some conflict or disagreement in a team. Having respect for team members means that you can criticize each other in a respectful way, as well as help each other to improve in areas where one team member is an expert and the other has something to learn.

Members of the SCRUM team

SCRUM Roles

Scrum defines three main "roles" for its team members:

- the Product Owner
- the SCRUM Master
- the SCRUM team (other members of the tech team who are neither the Product Owner nor the SCRUM Master)

The Product Owner

The Product Owner is responsible for making sure the team is working on the highest priority items. The Product Owner (or PO) does this by maintaining a **backlog** of future work items and ordering the backlog so that the highest priority items are at the top.

We've talked about the backlog in a previous chapter, but as a reminder, the SCRUM Guide defines a product backlog as:

an ordered list of everything that is known to be needed in the product. It is the single source of requirements for any changes to be made to the product.

The Product Owner must ensure that the next items to be worked on are clearly visible to the entire organization. He or she must also make sure that these items are ready to be worked on. If there are elements such as UX designs, copy, or translations which are required, the Product Owner ensures that they are ready. In particular, any major design decisions or logical decisions need to have been considered. If the tech team asks any questions when planning the feature, the Product Owner must provide answers.

The Product Owner will therefore decide what is worked on in the future. He or she maintains relationships with major stakeholders in order to know the organization's highest priority.

The SCRUM Master

The SCRUM Master is the person on the team who contributes the most to helping the team and the organization understand and respect the rules and behaviors of SCRUM.

It can be helpful to think of the SCRUM Master's contribution in three ways:

1. Service to the SCRUM Team

The SCRUM Master provides support to the SCRUM team by being the principal point of contact when issues arise. If one of the team members is blocked from doing their work, they should inform the SCRUM Master right away. The SCRUM Master will do his or her best to remove the impediment.

The SCRUM Master ensures that the team understands SCRUM and adheres to its rules. He or she also promotes a culture of self-organization, ownership, and effective communication. In addition, he or she will often facilitate meetings or events and help team members who are dealing with people in the organization who may not understand SCRUM.

2. Service to the Organization

The SCRUM Master plays a leading role in helping the organization as a whole to understand SCRUM principles. He or she is the reason why the team behaves the way it does (when adhering to the framework). If there are teams that are introducing SCRUM for the first-time, the SCRUM Master will play a major coaching role, ensuring that key stakeholders understand the benefits. In particular, if any managers try to "break" SCRUM by requesting/insisting that the team engage in outside activities , the SCRUM Master will step in and help "police" the situation so that SCRUM rules are again observed.

3. Service to the Product Owner

The SCRUM Master may be a member of a team where the Product Owner is new to working in SCRUM. In that case, he or she will assist them in understanding their role and responsibilities. Also, the SCRUM Master may facilitate meetings at the request of the Product Owner.

The SCRUM Team

The developers, designers, and testers (quality assurance members) of the team are known as the SCRUM team. They do all the actual work of building, testing, and releasing the software/product. The SCRUM team has the following characteristics:

- **Self-organizing -** No one (not even the Scrum Master) tells the team how to turn the current work iterations into deliverables. The team organizes itself to get the job done in the way it feels best. For example, the order in which tasks are done and jobs are assigned is decided by team members. Although challenging at first, the team members eventually develop a capacity for self-organization that results in high quality output.
- **Cross-functional -** Teams have the skills necessary to do the work required.
- **Equal member status -** No job titles or sub-teams are recognized in SCRUM. All team members are equal and the whole team is accountable. Either the (entire) team succeeds or it fails to obtain its goals.

SCRUM events

There is a set of prescribed events in SCRUM serving a specific purpose. These events have been designed to reduce the need for other events (that are not prescribed by SCRUM). They are all time-boxed which means that they have a maximum length.

There are five main SCRUM events:

- The sprint
- Sprint planning
- Daily stand-up
- Sprint demo
- Sprint retrospective

Apart from the sprint, the other events are an opportunity for inspection and transparency. They allow teams the chance to examine what is working and to improve efficiencies. Therefore, not scheduling these other four events would result in less inspection, less transparency, less opportunity for the team to improve.

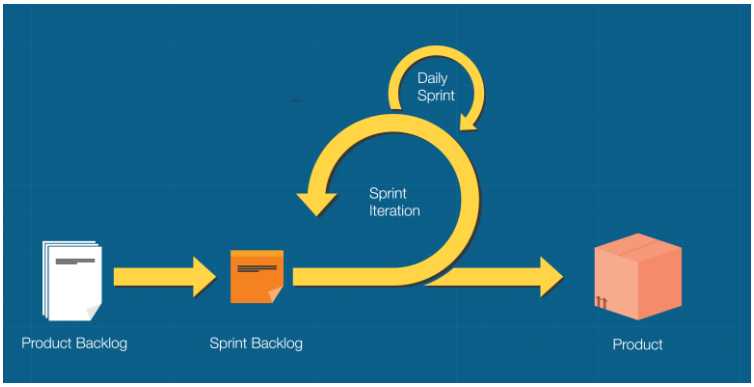
1. The Sprint

The sprint is the heart of SCRUM.

A sprint is usually 2 or 3 weeks in duration (although 1-week or 4-week sprints are sometimes seen). The team will attempt to begin, finish, and release work during this time. Once a sprint length is set, it cannot be changed. If your team is doing 2-week sprints, then the sprint must last exactly 2 weeks.

Some characteristics of sprints:

- Each sprint lasts for the same fixed duration (e.g., 2 weeks).
- The next sprint begins immediately after the previous one.
- The team attempts to begin, work on, complete, and release shippable features.
- Some items are selected from the product backlog that the team will attempt to complete for a sprint. These items become your sprint backlog.



2. Sprint Planning

The process of selecting which items to work on during the sprint is done during the sprint planning meeting. Sprint planning is time-boxed to a maximum of 8 hours.

Sprint planning is focused on answering the following questions:

- What can be delivered in the upcoming sprint?
- How will we achieve the work needed to deliver these items?

The Product Owner begins the planning session by discussing the objective of the sprint (the sprint goal) and which items, if delivered, would result in the goal being met.

The team discusses how it will deliver these backlog items. This is typically done by breaking up the items into smaller tasks and estimating the length of time needed. By crafting a plan for how it will deliver these items, the team will then calculate how many items it can confidently deliver.

The Product Owner will discuss these items with the team during planning, answer questions, and often make trade-offs. For example, perhaps a feature cannot fit into the sprint but part of the feature could be done. The Product Owner would decide whether that is a good approach. At the end of the meeting, the team agrees and commits to achieving the sprint goal through delivering a set of sprint backlog items via a plan.

3. Daily Stand-up

The daily stand-up is a daily meeting of no more than 15 minutes where each member of the team gives a brief update (ideally less than one minute) of how their work is progressing.

It is more common to have this meeting in the morning, although some teams working across multiple time zones may schedule the meeting for later in the day.

It is imperative that the meeting last no more than 15 minutes. Physically standing up helps keep the meeting short (It's easier to take a long time if you're comfortably seated).

The goal of this meeting is to let each team member know what the others are doing and plan to do. Team members can also use this time to indicate if they are stuck or blocked from progressing on their tasks.

The SCRUM Master is fully in control of this meeting. He will ask each of the members:

- What did you do yesterday?
- What you are going to do today?
- Do you have any obstacles or impediments?

It is important that each member listens to the update of the person speaking - no side conversations are allowed! Meetings should be held in the same place and at the same time. The meeting can be held in front of a progress board if one is used. That way members can update the board and see the status of the whole sprint in one place.

Benefits of Daily Stand-up Meetings

Although the benefits of keeping each team member up-to-date may seem obvious, it is worth looking at how the daily stand-up can positively impact the team's performance:

- Each team member will know what each other member of the team is doing.
- Team members will know if someone is working on something that might be affected by the tasks they are currently working on.
- Team members can give input on other team members' tasks, helping them if they are blocked or stuck.
- Each team member can surface blockers which can quickly be remedied.
- A self-managing culture is born from the team working together, solving problems together, and unblocking each other. You may hear conversations such as "if you pick up that task, then we can get this done" or "I can help with this if you're stuck or if the team needs it."
- The SCRUM Master can ensure that nobody is blocked for more than a day, can see if a task is taking longer than expected, and can take action. If some feature is going to take longer, a conversation with the Product Owner about whether some features or tasks should be removed from the sprint can be had. Ultimately the team wants to meet the sprint goal.

4. Sprint Demo

At the end of a sprint, the SCRUM Master will organize a sprint demo where the items of the sprint that have just ended can be shown to key stakeholders. Many organizations choose to have an open invitation for sprint demos. Anyone in the company who is interested can turn up and watch the team demonstrate what they have built.

Doing a demo at the end of the sprint has many great advantages:

- It gives the team confidence to show their work and receive positive feedback from the rest of the organization.
- Key stakeholders know they have a moment when they will see built features before they are released. This gives them an opportunity to give input if they see something seriously wrong or misunderstood, before customers are affected.
- Feedback can come from many sources if the whole of the organization is invited. Sometimes someone from the support team, marketing team, etc. can deliver an incredibly valuable piece of insight from their domain.
- When the tech team knows they have an upcoming demo with a large audience, they have an added motivation to deliver a great demonstration.

5. Sprint Retrospective

SCRUM teams will hold a retrospective meeting at the end of each sprint so that all team members can give input on what is working well and what is not working well. With well run teams, it shouldn't require more than 30 minutes (with 2 hours being an absolute max).

The following questions should be asked to the group with any member of the team suggesting answers:

1. What worked well during the last sprint?
2. What did not work very well during the last sprint?
3. What should we improve?

The SCRUM Master ensures that the tone of the meeting is productive and encourages things to be added to the improve list (a set of actions) when necessary.

The events aren't the only thing that characterizes SCRUM; next we will explore some key artifacts that help the team share information in SCRUM.

SCRUM events

Dictionary.com defines an artifact as:

- an object made by a human being

SCRUM artifacts are of vital importance as they help to share or "radiate" information. For this reason, artifacts in SCRUM are sometimes considered "radiators" of information.

SCRUM artifacts include:

1. Product backlog
2. Sprint backlog
3. Burndown charts
4. Product increment

1. PRODUCT BACKLOG

A product backlog is a living document, which means it changes and gets updated as the team learns more, as market conditions change, as competitors bring out new products, and as customers give input into what they need.

For each item in the product backlog, you should add some additional information:

- Description
- Order (as in the priority order within the whole backlog list)
- Estimate
- Value to the business

Adding to the Backlog

The Product Owner is responsible for capturing the needs of all stakeholders within the backlog. If members the sales, marketing, legal, support teams, etc. have desired product changes or improvements, the Product Owner will first listen, seek to understand these requests, and then add items to the backlog to represent these improvements or fixes.

It is important that the stakeholders feel they are listened to. For that reason, the backlog should be somewhere where other people in the organization can see it (and can see that their observation has been captured!).

The Product Owner will also add to the product backlog when he or she receives ideas from the development team or customers, as well as from doing competitive analysis

Backlog Grooming

Although the product backlog will always be added to, it is important to also edit (to groom) the items that are already in there. This means a periodical review which serves to identify, among other things, whether:

- Some backlog items are no longer necessary
- The estimates have changed for some backlog items
- The intended outcome for an item was achieved by another feature
- A particular problem has been fixed

It is also important to write specifications prepare documentation and assets for upcoming items on the backlog. For example, if the SCRUM team is working on Sprint 10, then the Product Owner is making sure the documentation and assets for any backlog items planned for Sprint 11 are being readied. In this way, the Product Owner works "a sprint ahead" of the team members.

2. SPRINT BACKLOG

After the sprint planning, the team has committed to a developed plan to work on a chosen set of items in the next sprint.

Those items are then removed from the product backlog and moved into the sprint backlog. The sprint backlog is guided by the sprint goal (which is the stated outcome objective of the sprint).

As the sprint evolves, it is possible that slight changes to the sprint backlog will occur. A new understanding of the feature the team is working on could mean that something is added to or removed from the sprint backlog.

The sprint backlog is a real-time picture of the work that the team currently plans to complete during the sprint. It should be easily visible as stakeholders may want to keep an eye on progress.

3. BURNDOWN CHARTS

A burndown chart is a graphical representation of the amount of estimated remaining work. Typically the chart will feature the amount of remaining work (perhaps in hours) on the vertical axis with time along the horizontal axis.

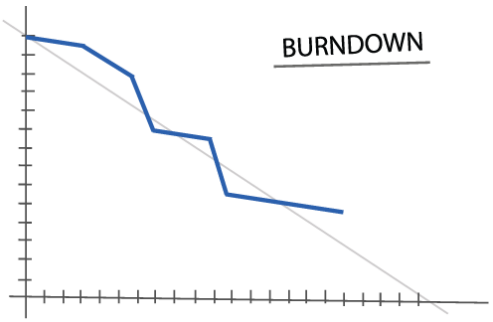
Burndown Example

Consider the following example of a two week sprint that has an estimate of 200 hours of work.

Before the sprint starts, the team has estimated 200 hours of work.

- After day 1, you would expect to have 180 hours of estimated work remaining.
- After day 2, you would expect to have 160 hours of estimated work remaining.
- After day 9, you would expect to have 20 hours of estimated work remaining.
- After day 10, you would expect to have all work complete (0 hours of estimated work remaining).

Things don't always go exactly to plan. A ten hour task might take 12 hours. Or an eight hour task might take one hour. So you should have a graph that tends towards zero as you move to the right. The image below represents the expected shape of our burndown chart:



In the chart above, the gray line represents the estimated number of hours left. You can see that this should tend to zero as the sprint comes towards its conclusion.

The blue line is the reality: the actual hours left. This should also tend towards zero as the sprint comes to its conclusion, but the line bounces around a bit to reflect the variability of each days' actual remaining work.

4. PRODUCT INCREMENT

The most important artifact is the actual product improvements, or in other words, the new code which has been written to enhance the features or usability of your product.

When a sprint ends, the team should have produced "potentially shippable code." Of course, the Product Owner may decide not to release for a few days/weeks/months, but she could release if she wanted to. Each sprint adds to the product. Therefore, the aggregation of all previous sprints together make up your product. When you add a new set of code from your sprint, then this new code has to combine with the existing code to work well. If your new features work well on their own but "break" some existing feature in your product, your product increment is not "potentially shippable."

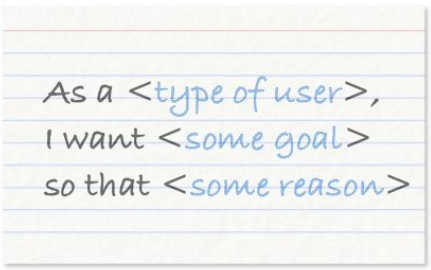
Successful SCRUM teams therefore create product increments that are potentially attainable and improve the existing product by adding new features or usability that works well with the existing code.

User Stories

Requirements in SCRUM are captured using user stories. **User stories** are short descriptions of a product enhancement (i.e., a product feature) expressed in the following format:

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

A very common practice is to write user stories on an index card (like the image below) and then put all user stories on a wall in the office.



Examples of User Stories

The best way to understand what user stories are is to look at some examples. Have a look at the examples below:

- As a blogger, I want to show a Twitter share button at the end of each blog page so that my readers can share this article in their Twitter feed with one click if they want to.
- As a Snapchat user, I want to be notified if another user takes a screenshot of my chat so that I can know if my message has not been deleted.
- As a Facebook group organizer, I want to see the behavior of my community members so that I can see which of my promotions are working.
- As a marketing manager, I want to apply free delivery to all shoes listed on our website for a specified period so that I can run promotions.
- As an OpenClassrooms teacher, I want to create a quiz so that students can be evaluated on their knowledge of a course they are taking.

User Stories are Simplified Requirements

User stories are simplified versions of real requirements. Let's take one of the user stories above:

As a blogger, I want to show a Twitter share button at the end of each blog page so that my readers can share this article in their Twitter feed with one click if they want to.

If you show this user story to your engineering team in a planning session, they will probably ask questions like:

- If the reader is logged-in to Twitter, then do we expect that the message posts directly?
- If the reader is not logged-in to Twitter, what should happen?
- Can a reader post this link on their Twitter feed more than once? Is there a limit?
- What if the URL to the blog post is greater than the Twitter character limit?
- Do you need to track how many people click on this link? Should we use a link management tool like http://bit.ly?
- What should happen if the Twitter service is down? (God forbid!)

You can see that the engineering team asked a lot of relevant "what if" questions! This information (the answer to all these questions) is not contained within the actual user story itself. This is good news because:

1. It forces the whole team to have a conversation about the user story.
2. The user story is expressed from the perspective of the user. By understanding the user's goal, you can investigate whether the feature suggested is the best way of helping them achieve that goal! Sometimes the engineering team thinks of an idea that is better than what is written on the card.
   Writing short descriptions like this in non-technical language means that the whole business (not just tech people) can understand what the team is working on.
3. As a team, it is easy to handle new requests; put a user story on an index card, (or post-it note) and add it to the user stories wall in your office. You'll figure out the details once you're ready to work on it (i.e., you can answer all the "what if" questions just before you start work).
4. In the above scenario, where a product manager chooses a user story and then discusses it with the engineering team in a planning session (often called a Requirements Workshop), is perfectly normal. After a healthy discussion, more details are known and written down. This allows the team to define a set of scenarios known as **acceptance tests**, which represent the details of the user story.
5. User stories may be written by various stakeholders including clients, users, managers, or development team members. Most commonly, it will be the product manager who writes them.

The Invest Model

Agile consultant and author Bill Wake describes the characteristics of good user stories by using the INVEST model.

Independent - When user stories are independent, they do not overlap each other in functionality. Thus you can begin work on a user story in any order you choose because there is no dependency on any other user story.

Negotiable - Part of the beauty of the user story is that it reveals a need and why it is needed. As Bill Wake says, "A good story captures the essence, not the details." The details will be ironed out later by stakeholders and developers just before and during development.

Valuable - The user story must be valuable as perceived by the customer. This should lead the development team to try to deliver a visible value for each user story (not a database layer which is not something the customer can see).

Estimable - When discussing a user story, if the team says it cannot estimate the work required for the user story due to missing information, then you need more information.

Small - Good user stories are small and typically can be completed in a few days or weeks by a developer. When user stories are bigger, you may be able to split them.

Testable - If the team doesn't know how to test a user story, then they probably don't understand what is needed. Knowing exactly what to test is equivalent to knowing exactly what to build.

Epics

An epic is just a big user story. We manage epics by breaking them down or "splitting" them into several user stories.

Consider the following user story:

As a group organizer, I want to schedule events for my members so that I can encourage them to attend upcoming events.

In reality, this may break down into a set of user stories like this:

- As a group organizer, I want to create an event in the system so that I can promote my real-life events.
- As a group organizer, I want to see a list of created events so that I can manage events.
- As a group organizer, I want to be able to pause an event so that invitations are no longer sent.
- As a group organizer, I want to be able to cancel an event so that any people who registered get notified of the cancellation.
- As a group organizer, I want to choose a targeted audience the event is sent to so that I can cater my events to the right people.
- As a member, I want to receive an email for events so that I can be alerted to invitations.
- As a member, I want to receive an in-app invitation notification so that I am aware of upcoming events.
- As a member, I would like to RSVP for an event so that I am guaranteed a place.
- As a member, I would like to pay for the event in advance with a credit card.

Improving your User Stories

AgileKRC shares the following Big Three Questions that you (or your team) can ask after creating user stories:

1. Immediately after reading the user story, is it obvious what the user story is about?
2. Does each element of the user story add significant value and therefore avoid duplication or partial duplication of other elements?
3. Is it totally 100% free of "the how" (the solution)?

These are good questions to ask after you have written a user story, but before you start working on it!

Agile consultant Ben Linders adapts the "Perfection Game" pattern from the book *Software in Your Head* as a good way to learn about how to improve user stories after the team has worked on them. If you are organizing a "lessons learned" meeting (often called a "retrospective" meeting), you could take user stories that have been completed and get some members of the team to:

1. Rate the user story (a score between 1 and 10)
2. Explain what they liked about the user story
3. Explain what they would do to make it perfect

Acceptance tests

Card, Conversation, Confirmation

We mentioned in the last chapter that:

1. User stories are deliberately simplified versions of requirements and short enough to be written on a post-it note or an index card.
2. Since the user story does not contain all of the detailed information required to start work on it, the next step is to have a conversation with the entire team (perhaps during a planning session) to discuss details before implementing a solution.
3. The additional details that come out of these conversations enable you to make decisions as a team and determine which rules (i.e., which tests) represent the desired implementation of this feature. Writing these details down enables confirmation later on when the feature is built and ready to be tested. Specify up front what has to work (what tests it should pass) in order for this feature to be acceptable. For this reason, these additional scenarios are often called **acceptance tests.**

Writing Acceptance Tests

When writing acceptance tests, think of a series of examples as to how the system should behave. Take the following three examples of how an ATM machine might work:

1. If the customer tries to withdraw $20 and her account balance is over $20, then allow the withdrawal and reduce the account balance by $20.
2. If the customer tries to withdraw $20 but her account balance is under $20, then do not allow the withdrawal.
3. If the customer tries to withdraw $300, then do not authorize the withdrawal and show the message: "maximum withdrawal is $200."

Write this set of examples in the GIVEN-WHEN-THEN format:

- GIVEN: A set of initial circumstances (e.g., bank balance)
- WHEN: Some event happens (e.g., customer attempts a withdrawal)
- THEN: The expected result as per the defined behavior of the system

Note the following benefits of using the GIVEN-WHEN-THEN format:

- It is easy to see the set of initial conditions, the event, and the outcome.
- This makes it easy to understand as well as easy to test.

Look at the first example from above:

If the customer tries to withdraw $20 and her account balance is over $20, then allow the withdrawal and reduce the account balance by $20.

Writing that in GIVEN-WHEN-THEN format will look like this:

ID	GIVEN	WHEN	THEN
01	User balance = $23	User asks to withdraw $20	Withdrawal is authorized AND User balance is now $3

Using Examples

Sometimes, the rule is also written as part of the acceptance test.

RULE: Only allow withdrawals when there is sufficient balance

ID	GIVEN	WHEN	THEN
01	User balance = $23	User asks to withdraw $20	Withdrawal is authorized AND User balance is now $3

The two examples of withdrawing $20 (one with a $23 balance and one with an $18 balance) are part of the same rule. Look at the second example:

RULE: Only allow withdrawals when there is sufficient balance

ID	GIVEN	WHEN	THEN
01	User balance = $23	User asks to withdraw $20	Withdrawal is authorized AND User balance is now $3
02	User balance = $18	User asks to withdraw $20	Withdrawal is NOT authorized AND User balance is still $18

Note the capitalized "NOT" in the second acceptance tests. This draws attention to the fact that there is something different here that might be missed if you were reading dozens of acceptance tests. It is just a little added emphasis, but it helps!

Now look at the daily withdrawal limit example and write it in GIVEN-WHEN-THEN format:

RULE: Only allow withdrawals up to the limit of $200

ID	GIVEN	WHEN	THEN
03	User balance = $230	User asks to withdraw $190	Withdrawal is authorized AND User balance is now $40
04	User balance = $230	User asks to withdraw $210	Withdrawal is NOT authorized AND User balance is still $230

Notice the ID beside each acceptance test. If you are discussing these tests later (maybe if you find issues when testing the feature once it's built) then it is very helpful to be able to direct team members to the fact that you are talking about Acceptance Test 03 rather than trying to describe it!

Putting it all together, you would end up with something like this:

RULE: Only allow withdrawals when there is sufficient balance

ID	GIVEN	WHEN	THEN
01	User balance = $23	User asks to withdraw $20	Withdrawal is authorized AND User balance is now $3
02	User balance = $18	User asks to withdraw $20	Withdrawal is NOT authorized AND User balance is still $18

RULE: Only allow withdrawals up to the limit of $200

ID	GIVEN	WHEN	THEN
03	User balance = $230	User asks to withdraw $190	Withdrawal is authorized AND User balance is now $40
04	User balance = $230	User asks to withdraw $210	Withdrawal is NOT authorized AND User balance is still $230

Now imagine for a moment that you are a new developer on the team. You missed the planning session which happened last week. But someone shares these acceptance tests with you. Ask yourself the following questions:

- Are these acceptance tests easy to understand?
- Did they take a long time to read?
- Does having both the example and the rule help?

- Do you think you would know what to build?

Most people would answer "yes" to all these questions - with the exception of "Did they take a long time to read?"

The Definition of Done

Knowing When a User Story is Complete

Have you ever been in the following situation? Your apartment needs a thorough cleaning and you decide to be the one to do it. When you tell your partner, she is ecstatic! Yet when she comes home, rather than give you a big thanks, she runs a finger over a surface and shows you the dust that remains. The place is cleaner - but in her eyes, the job has not been done completely.

The same thing can happen in software development! Different teams or different members of a team can have different ideas of what it means for a piece of work (e.g., a user story) to be "done" and to be "ready for production."

The term "production" comes from manufacturing industries where a component or product would be ready for use when it becomes part of the "production line." When a feature is "ready for production," it means it can be pushed live where it will be then seen by real customers.

Now imagine that you and your partner sit down and define a list for what a full cleaning of the apartment means! The checklist looks like this:

- Sink empty
- Dishwasher full
- All floors vacuumed
- Kitchen floor mopped
- Dusting done on top of bookcase, fridge, and TV
- Fridge emptied and all shelves cleaned
- Inside doors of fridge cleaned

Now, as you can imagine, the likelihood of one of you cleaning the apartment fully and the other disagreeing that the cleaning is done is much reduced!

What is a Definition of Done?

A Definition of Done is a checklist of activities that add value to the product which must be fully completed before a member of the team can refer to a user story as "done."

- All of the automated tests for the user story run without failing.
- Our QA (Quality Assurance) team has tested this feature and not found any defects.
- Our QA team has verified that all acceptance tests have passed.
- The code was peer-reviewed (another member of the tech team looked at the code and was happy).
- The product manager has tested and verified that they are happy.
- The Code Quality score is at least 3.5 (assuming that some system calculates this score!).
- No security issues have been found.
- All documentation has been updated.

How to Define Done?

The best way to come up with a Definition of Done for your team is to sit together as a team and come up with a checklist that the whole team agrees upon.

1. Ask the questions, "What tasks would we always want to do?" and "What kind of checks would we always make?" before pushing code to real customers ("pushing to production").
2. Consider any previous times that some team members referred to work as "done" but other team members felt that it was "not done." What were the differences? Are those items that should be added to your Definition of Done?
3. If you have a dedicated Quality Assurance (QA) team that does all the testing of your products before release, what do they consider as important?
4. Many teams write automated checks (or tests) on their codebase so that they know if some new code breaks any of the logic of the old code. If you have automated tests that run on your codebase, you will probably want to include that "no new code breaks any old code." All tests should pass!
5. Do you have any automated systems that run on your code? Some tools like Code Climate can even provide you with a quality score and can scan your code for bad practices, such as repeated chunks of code! If you have such tools, they can be incorporated into your Definition of Done.

Benefits of Having a Definition of Done

There are many benefits to having a clear Definition of Done:

- A checklist exists to make sure only high-quality work is released to real users.
- Team members will have much less reason to argue over what "done" means.
- User stories typically get finished (i.e., "done") before more are started. This means fewer items that are "works in progress" and more user stories are released to real customers.
- Measuring "quality" becomes part of the weekly/monthly routine.
- Fewer items get "pushed back" into development because they only "make it out" of development when they are genuinely ready.
- Fewer conversations like the one in the video below!