Contents

The backlog

A backlog is a list of features or technical tasks which the team maintains and which, at a given moment, are known to be necessary and sufficient to complete a project or a release:

- If an item on the backlog does not contribute to the project's goal, it should be removed;
- On the other hand, if at any time a task or feature becomes known that is considered necessary to the project, it should be added to the backlog.

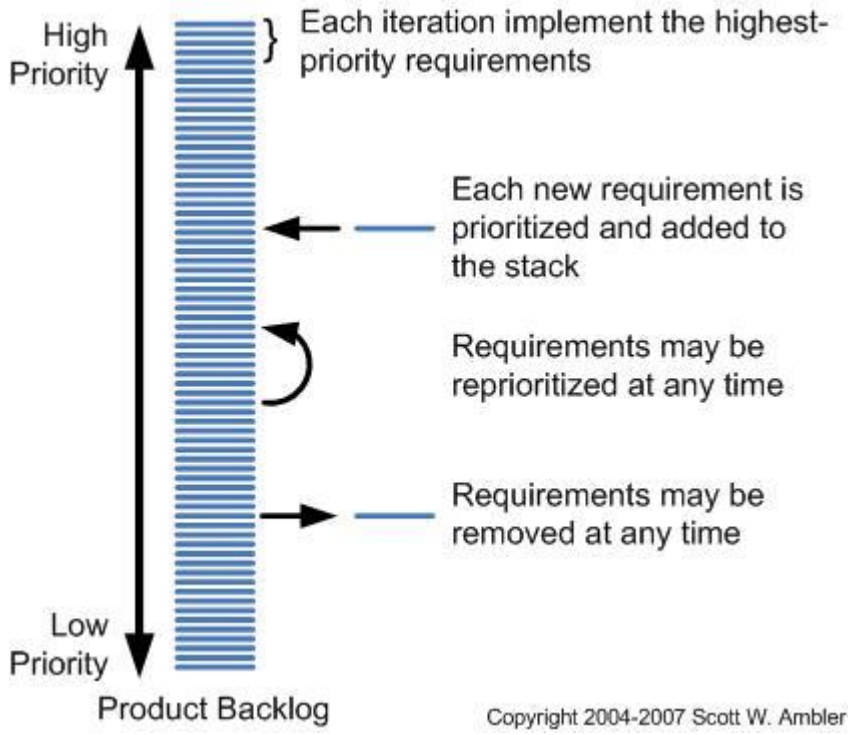
These "necessary and sufficient" properties are assessed relative to the team's state of knowledge at a particular moment; the backlog is expected to change throughout the project's duration as the team gains knowledge.

The backlog is the primary point of entry for knowledge about requirements, and the single authoritative source defining the work to be done.

Product Backlog: Simplistic

The below figure overviews the Scrum approach to managing requirements where your software development team has a stack of prioritized and estimated requirements which need to be addressed (Scrum calls this prioritized stack a "product backlog"). There are several important points to understand:

- New requirements are prioritized by your project stakeholders and added to the stack in the appropriate place.
- Fundamentally a single person needs to be the final authority when it comes to requirement prioritization. In Scrum, and Disciplined Agile Delivery (DAD) which adopts this role from Scrum, this person is called the product owner.
- Although there is often many project stakeholders, including end users, managers, architects, operations staff, to name a few and the product owner is responsible for representing them all.
- The stack/backlog is initially filled as the result of your requirements envisioning efforts at the beginning of the project (in Scrum they call this "populating the backlog").
- Each iteration (a "sprint" in Scrum terminology) your team pulls an iteration's worth of work off the top of the stack and commits to implementing it by the end of the iteration.
- Your project stakeholders have the right to define new requirements, change their minds about existing requirements, and even reprioritize requirements as they see fit.
- Stakeholders are responsible for making decisions and providing information in a timely manner. On some projects a business analyst, often in the role of product owner, may take on this responsibility. Whoever is in this role will need to work together with the other stakeholders to ensure everyone is represented fairly, often a difficult task.
- The priorities of non-requirement work items are either negotiated by the team with stakeholders or are addressed as part of slack time within the schedule. Many Scrum teams are now putting more than just requirements, such as defects, on their backlogs.



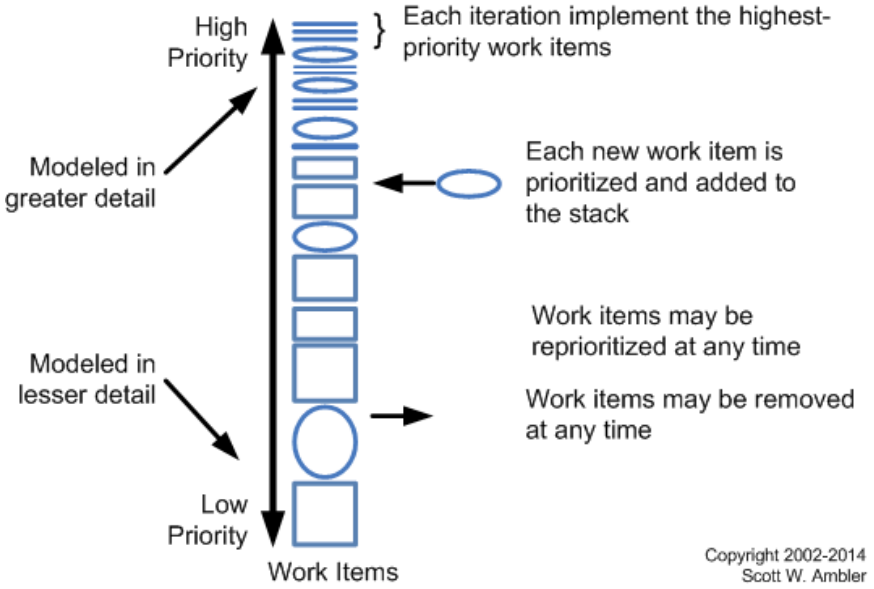
Work Item List: Disciplined Agile

The approach depicted in the above figure is fairly simplistic, a reflection of what I would consider agile construction level thinking. The below figure however overviews a more disciplined approach which extends the approach described above to managing the work items. This approach is the default suggested by Disciplined Agile Delivery (DAD) both of which reflect the real-world realities faced by agile delivery teams. There are a few interesting improvements that you should consider:

- **Go beyond functional requirements**. We know that we do more than just implement new requirements each iteration, we often do non-requirement related work such as take training, review products of other teams, address defects (I believe that defects are simply another type of requirement) and so on. The point is that more than just requirements need to be on the stack, we really need work items.
- **Take a risk-value approach**. Disciplined agile teams recognize that there are common risks faced by development teams, risks which they want to address as soon as possible. Such risks include the need to come to stakeholder consensus early in the project, a risk which can be addressed through requirements envisioning and perhaps the development of a shared vision or project charter. Another common risk is the need to prove that your

architecture strategy, identified via architecture envisioning, actually works. The most effective way to do this is to prove your architecture with working code by building an end-to-end skeleton, or steel frame, for your system which addresses the major technological risks faced by your team. Disciplined Agile Delivery (DAD) teams will look at their work item stack early in the project, typically during the Inception/"iteration 0"/"sprint 0" part of the project to identify requirements which exhibit these technically risky features. If these requirements aren't at the very top of the stack, and they often are because risk and reward (value) have a tendency to be co-related, then they discuss the issue with their product owner and see if they can motivate that person (who is responsible for prioritization) to move those requirements to the top of the stack too. If a high-risk requirement is currently near the bottom of the stack then you should question whether that requirement is actually needed because chances are good you'll never actually get around to working on it as higher priority work will always take precedent.

- **Model a bit ahead**. Because we know that all requirements, let alone work items in general, are not created equal we shouldn't naively assume that we should just wait to pull an iteration's worth of work off the top of the stack at the beginning of the iteration. What if a work item is very complex, requiring a bit more thinking that what generally occurs in iteration planning sessions? Disciplined agile teams will adopt the Look-Ahead Modeling practice and look an iteration or two down the work item stack and invest the time to explore complex upcoming work items to reduce their overall project risk. Modeling a bit ahead is called backlog grooming in Scrum, revealing some of the unnecessary conceptual coupling amongst practices in Scrum.

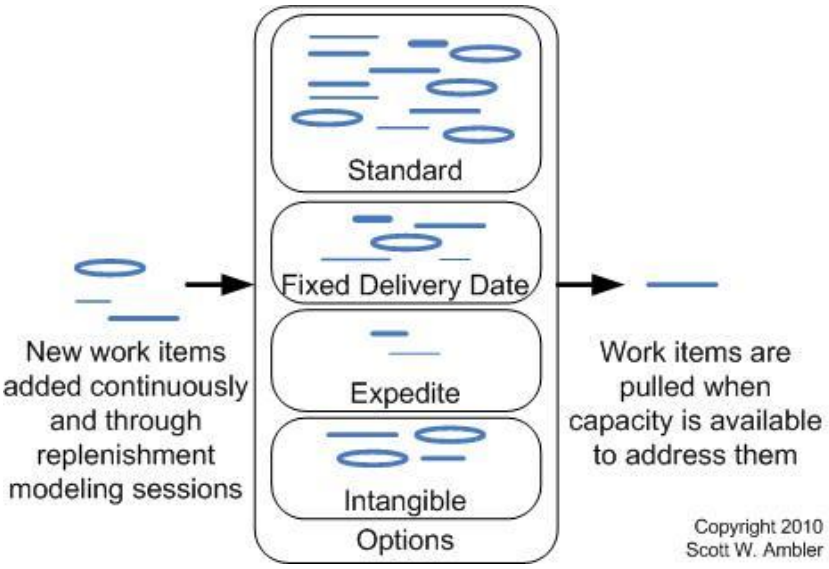


Option Pool: Lean

The below figure depicts a lean approach, common on Kanban teams, to requirements management. There are several key differences between the agile approach to requirements/work management and the lean approach:

- **Options**. The work items are viewed as potential options to be addressed in the solution, not as required work items. As an aside, the term "requirement" in IT has always been questionable at best (if something is a requirement then how can some or all of it be dropped from your delivery scope?) but it is a term which I fear we will never be able to abandon.
- **Options are managed as a pool**. Where agile teams manage work items as a prioritized queue, lean teams manage options as a pool from which they work with stakeholders to select the highest value work when they have the capacity to perform the corresponding work. In effect prioritization is done on a just in time (JIT) with the team's stakeholders. This can be complex because individual stakeholders will have different priorities, so tradeoffs will need to be made, and because there may be different classes of service being supported by the team. Of course the risk mitigation issues described by the disciplined agile strategy should also be considerations in which option is selected next.
- **Options are potentially organized into classes of service**. Not all options are created equal. In the book Kanban, David J. Anderson, describes several potential categories of options (which Anderson refers to as classes of service) which your team may consider. Many work items fall into the "standard" class where their prioritization is based primarily on their business value to your organization. Some work items have a fixed delivery date, this is common for requirements resulting from government legislation or contracts signed with your external customers. You may also have a (hopefully) small number of expedited work items, such as critical fixes for production problems or high-priority customer requirements. Finally, you may have some low-priority "intangible" work items that you know you'll need to address at some point in the future that you decide to address when the team believes is appropriate (perhaps to streamline other work, to take a rest from more challenging work, ...). It is important to note that the categories described here are not carved in stone, your team may identify different ones or may not need to categorize at all. Also, classes of service could be supported in the agile strategies described above as multiple queues.
- **The goal is to limit work in progress (WIP)**. When agile teams strive to choose a fixed amount of functionality, just enough that they can implement in a single iteration, lean teams choose to produce a continuous flow of functionality pulling work into their "delivery system" when they have the capacity to do so. Limiting WIP increases the team's delivery predictability, improves quality (due to shorter feedback cycles), and increases productivity.

- **Old work items are culled**. Lean teams will often strive to identify aging rules where if a work item is in the pool longer than a certain amount of time, it is removed from the pool under the assumption that if it's gone that long without being important enough to be selected from the pool then it likely never will be. If the work item does prove to be needed it can always be added to the pool at a future date. Note that the aging rules will vary between teams, perhaps three months for one team although five months for another. Also, this strategy can be applied to either the product backlog or work item list approaches described earlier.



Source: www.agilemodeling.com

No one strategy is best in all situations, and you will need to identify choose and then tailor the one that works best for you.

Strategy	Advantages	Disadvantages	When to use it
Product Backlog	Simple	· Overhead associated with keeping the stack prioritized as priorities change · Requires multiple stacks, or a more complex prioritization scheme, to support classes of service · Typically requires teams to have an	• Simple situations (typically co-located agile teams developing fairly straightforward software)

		"overhead fudge factor", sometimes upwards to 30%, to account for non-requirements work each iteration	
Work item list	• Easily supports different types of work items • "Overhead fudge factor" required is much smaller or non-existent compared with product backlog approach	• Overhead associated with keeping the stack prioritized as priorities change • Requires multiple stacks, or a more complex prioritization scheme, to support classes of service	• For agile teams in not-so-simple situations, particularly teams in scaling situations.
Option collection	· Works for both agile and non-agile teams · No overhead of managing a prioritized queue · The overhead of the "planning game" at the beginning of each iteration goes away.	• Requires stakeholders who are sophisticated enough to prioritize the work on a JIT basis	• Lean teams (e.g. Kanban) will adopt a variant of this strategy.

Grooming vs. Requirement analysis

Grooming

The purpose of backlog grooming is to keep the product backlog up to date and clean. Scrum doesn't prescribe how you should do backlog grooming, and different approaches are used by product owners and teams to do this.

"Yes, keep the backlog groomed, but don't make the whole team do it. Do a bit of it on your own, and lots of it with your closest collaborators – a balanced product discovery team."

"With an efficient development team thanks to the scrum practices it's more important than ever to make sure that a backlog is produced that give the developers the information they need to be efficient. (…) If we spend so much time to improve the development process I think it should be natural to also look at the process that leads up to the development phase. Unfortunately I think it's quite rare to a have a well-defined workflow for getting items into the backlog. (…) It is

evident just after only a few months that a proper product backlog management process with good tool support helps."

The intent of a "grooming" meeting is to ensure that the backlog remains populated with items that are relevant, detailed and estimated to a degree appropriate with their priority, and in keeping with current understanding of the project or product and its objectives. Unlike a more formal "requirements document" the backlog is understood as a dynamic body of information. For instance, not all user stories need to have been broken down to a fine-grained level at the onset of the project, or given detailed estimates; but it is important that at any moment a "sufficient" number of stories should be ready for scheduling in the next few iterations. An Agile project is, no less than any other, subject to "scope creep", in the form of user stories which do not really yield substantial value but were thought "good ideas at the time", and entered into the backlog lest they be forgotten. In the absence of explicit efforts aimed at managing this inflation, this inflation would result in the too well known pathologies of schedule and budget overruns.

Pre-grooming

Frequency

As many times as needed, iterative process

Participants

Mandatory – Release leads, UI/UX, program and product owners (story writer)

As needed- QA leads, Stream leads and developers as SMEs

Optional – Head of development, head of product

Description

Pre-grooming is the process by which a story is evaluated for readiness to go into the backlog for grooming by the team.

The story is ready if it is well articulated, concise and contains all of the necessary information that a developer would need to point it and immediately start on the feature. If a story is not ready to be started it should not go into the grooming process or into the backlog.

On the other hand if the story will not be executed, or at least started, sometime in the next two to three sprints (depending on the sprint length) they should not go into the pre-grooming process at all. The reason for this is that the story may still change. Even though it may be considered complete if there is no desire to start development immediately it indicates that there are other higher priority stories that should be tackled first.

Attributes of a Good Story

Each story should be independent from all other stories in the sprint. In other words; a developer should be able to complete it and releases it as a single independent feature

If there are related stories or stories that are dependent of each other they should be grouped under a common epic

If the story is too big it needs to be split into multiple stories and grouped under a common epic

If the story is too small it should be combined with other small stories if possible and logical

The details of the story can and should be negotiated between the story writer and the team conducting the pre-grooming and grooming if it is needed

Each story should be written so that the value to the customer is clear to all who read it

Story should focus on the “What” not the “How” of how it will be executed

Each feature articulated in the story should be testable and, if possible, the testing metrics articulated

Details for the story should appear as attachments and annotations, not as part of the main story. Not all of this will come from the story writer, but the story writer, program manager and development lead should all work together to gather this information during the pre-grooming process. Some examples of this are:

- Annotated wireframes, mock ups or creative. Only what is changing or relevant to this one story should be included
- Error messaging and page text
- SEO information
- Sample XML or any other data that a developer will have to code against for this story
- Relevant schemas

Goal

Through this process the participants attempt to anticipate developer and QA questions prior to grooming and provide all of the necessary collateral for development and test writing to begin.

Output

- Finished stories ready for grooming and in the backlog
- All of the necessary supporting information, data and creative required to start the development and testing process
- Approvals required to move to the next step

Grooming

Frequency

Ideally once per sprint, per agile team, but no more than twice. If a story is deemed to not have enough information to point by the development team at the time of grooming, and the requested information or answers cannot be provided in time for the second scheduled grooming meeting, then the story will be sent back to pre-grooming for one more pass.

Participants

Mandatory – Team (stream) leads, developers and QA analysts in the agile team that will be developing this story, product owners (story writer) and program manager/SCRUM Master

Optional – Dev head, product head, release leads, QA leads

Description

The grooming is intended to allow the actual developers and QA analysts who will be responsible for delivering the feature(s) being groomed to get answers to any remaining questions and for them to point the story(s). This is not intended as a group exercise and should not be attended by anyone who is not working on the stories being discussed. The only exception is if there is a team working on stories that are dependent on the stories being groomed, or if the stories being groomed are dependent on the stories being worked on by the other team in question.

Steps in a Good Grooming

Ensure everyone attending has read the stories being presented before the meeting. If not scrap the meeting

Determine if all of the QA and development resources that will be working on the story fully understand the requirements.

Stream lead and program management must ensure that the team asks questions and is comfortable with their understanding of the requirements before the team points the feature

Negotiate the details of the story between the story writer and the development team as needed

Provide point estimation of the effort to bring the story to fruition. Ideally this should be done through the use of Pointing Poker. This estimation of effort needs to take into account the following efforts:

- Dev/QA prep
- Unit test development from the developers
- QA test automation from the QA team
- The writing of the test cases
- The effort to code the feature and enough time to account for the time that will be needed during bug remediation
- The effort to QA the feature and re-QA the feature after bug fix
- The effort required to create or gather data needed during the QA process

Goal

To provide estimation of effort and to determine how many stories should go into a sprint. Prioritize the order of those stories and decide which stories should go into the backlog. To negotiate on the details of each story to make sure it is achievable in the most efficient manner possible.

Output

- Assignment of story to correct agile team
- Point estimation of each story
- List of stories in sprint
- Priority order of stories in sprint
- List of stories in backlog
- Approvals required to move to the next step

Source: https://agilesoftwareengineer.com

Definition of done

In software, a Definition of Done may be: “Done means coded to standards, reviewed, implemented with unit Test-Driven Development (TDD), tested with 100 percent test automation, integrated and documented.”

In a services context, it might look something like this: "Done means every task under the User Story has been completed and any work created is attached to the User Story so the Product Owner can review it and make sure it meets his or her expectations."

The Definition of Done ensures everyone on the Team knows exactly what is expected of everything the Team delivers. It ensures transparency and quality fit for the purpose of the product and organization.

Having a “definition of done” has become a near-standard thing for Scrum teams. The definition of done (often called a “DoD”) establishes what must be true of each product backlog item for that item to be done.

Definition of done is crucial to a highly functioning Scrum team. The following are characteristics that you should look for in your team’s definition of done. Verifying that your team’s DoD meets these criteria will ensure that you are delivering features that are truly done, not only in terms of functionality but in terms of quality as well.

DoD is a checklist of valuable activities required to produce software.

Definition of Done is a simple list of activities (writing code, coding comments, unit testing, integration testing, release notes, design documents, etc.) that add verifiable/demonstrable value to the product. Focusing on value-added steps allows the team to focus on what must be completed in order to build software while eliminating wasteful activities that only complicate software development efforts.

DoD is the primary reporting mechanism for team members.

Reporting in its simplest form is the ability to say, “This feature is done.” After all, a feature or Product Backlog Item is either done or it is not-done. DoD is a simple artifact that adds clarity to

the “Feature’s done” statement. Using DoD as a reference for this conversation a team member can effectively update other team members and the product owner.

DoD is informed by reality.

Scrum asks that teams deliver “potentially shippable software” at the end of every sprint. To me, potentially shippable software is a feature(s) that can be released, with limited notice, to end users at the product owner’s discretion. Products that can be released to end users with two days can be reasonably said to be in potentially shippable state. Ideally, potentially shippable is equivalent to the Definition of Done.

In reality, many teams are still working towards a potentially shippable state. Such teams may have a different DoD at various levels:

- Definition of Done for a feature (story or product backlog item)
- Definition of Done for a sprint (collection of features developed within a sprint)
- Definition of Done for a release (potentially shippable state)

There are various factors which influence whether a given activity belongs in the DoD for a feature, a sprint or a release. Ultimately, the decision rests on the answer to the following three questions:

- Can we do this activity for each feature? If not, then
- Can we do this activity for each sprint? If not, then
- We have to do this activity for our release!

DoD is not static.

The DoD changes over time. Organizational support and the team’s ability to remove impediments may enable the inclusion of additional activities into the DoD for features or sprints.

DoD is an auditable checklist.

Features/stories are broken down into tasks both during sprint planning and also within a sprint. The DoD is used to validate whether all major tasks are accounted for (hours remaining). Also, after a feature or sprint is done, DoD is used as a checklist to verify whether all necessary value-added activities were completed.

It is important to note that the generic nature of the definition of done has some limitations. Not all value-added activities will be applicable to each feature since the definition of done is intended to be a comprehensive checklist. The team has to consciously decide the applicability of value-added activities on a feature-by-feature basis. For example, following user experience guidelines for a feature that provides integration point (eg: web service) to another system is not applicable to that particular feature; however, for other features within the system that do interface with a human being, those user experience guidelines must be followed.

Source: http://www.scrumalliance.org

Planning vs. Test plan

Planning

Sprint planning

In Scrum, the sprint planning meeting is attended by the product owner, Scrum Master and the entire Scrum team. Outside stakeholders may attend by invitation of the team, although this is rare in most companies.

During the sprint planning meeting, the product owner describes the highest priority features to the team. The team asks enough questions that they can turn a high-level user story of the product backlog into the more detailed tasks of the sprint backlog.

The product owner doesn't have to describe every item being tracked on the product backlog. A good guideline is for the product owner to come to the sprint planning meeting prepared to talk about two sprint's worth of product backlog items.

To make an example really simple, suppose a team always finishes five product backlog items. Their product owner should enter the meeting prepared to talk about the top 10 priorities.

There are two defined artifacts that result from a sprint planning meeting:

- A sprint goal
- A sprint backlog

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 e-Commerce 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.

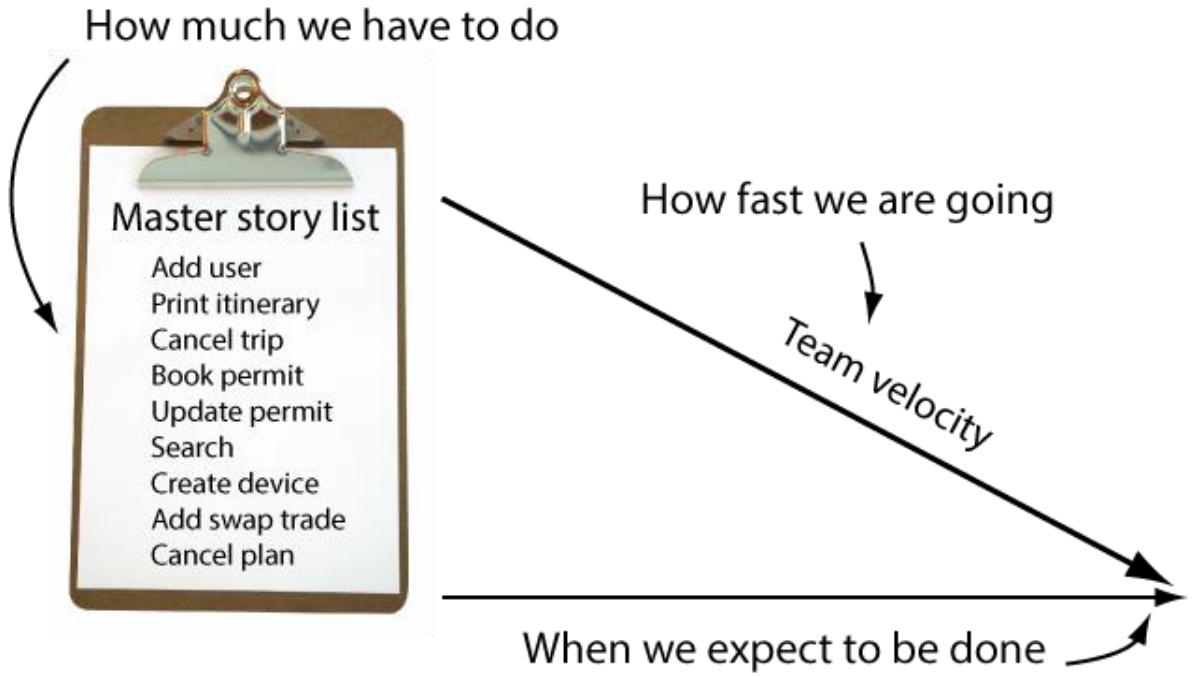
The success of the sprint will later be assessed during the sprint review meeting against the sprint goal, rather than against each specific item selected from the product backlog.

The sprint backlog is the other output of sprint planning. A sprint backlog is a list of the product backlog items the team commits to delivering plus the list of tasks necessary to delivering those product backlog items. Each task on the sprint backlog is also usually estimated.

An important point to reiterate here is that it's the team that selects how much work they can do in the coming sprint. The product owner does not get to say, "We have four sprints left so you need to do one-fourth of everything I need." We can hope the team does that much (or more), but it's up to the team to determine how much they can do in the sprint.

Release planning

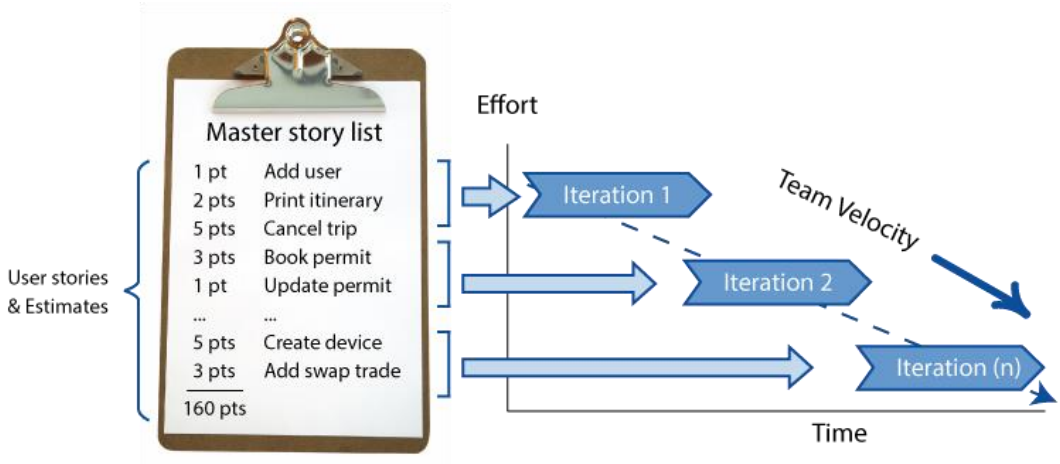
In its simplest form, agile planning is nothing more than measuring the speed a team can turn user stories into working, production-ready software and then using that to figure out when they'll be done.



Our to-do list on an agile project is called the master requirement list. It contains a list of all the features our customers would like to see in their software.

The speed at which we turn requirements/user stories into working software is called the team velocity. It's what we use for measuring our team's productivity and for setting expectations about delivery dates in the future.

The engine for getting things done is the agile iteration - one to two week sprints of work where we turn requirements/user stories into working, production-ready software.



To give us a rough idea about delivery dates, we take the total effort for the project, divide it by our estimated team velocity, and calculate how many iterations we think we'll require to deliver our project. This becomes our project plan.

Iterations = total effort / estimated team velocity

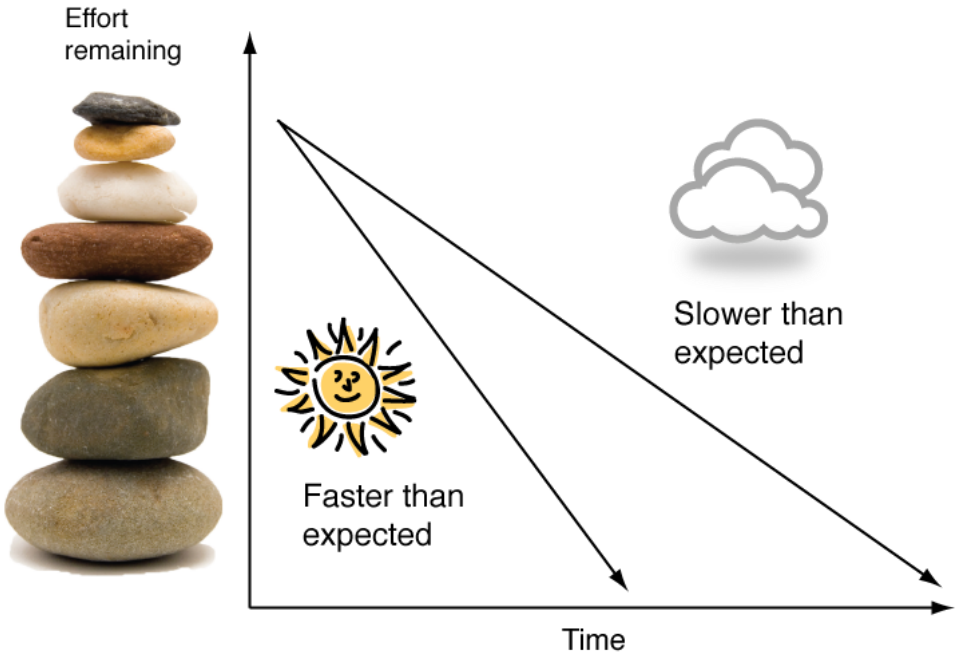
For example:

Iterations = 100 pts / 10 pts per iteration = 10 iterations

Now, as we start delivering, one of two things is going to happen. We are going to discover that:

a) We are going faster than expected

b) We are going slower than we originally thought.



Faster than expected means you and your team are ahead of schedule. Slower than expected (more the norm) means you have too much to do and not enough time.

When faced with too much to do, agile teams will do less (kind of like what you and I do when faced with a really busy weekend). They will keep the most important requirements/stories, and drop the least important. This is called adaptive planning and it's how agile teams work within their budgets and keep their projects real.

Source: http://www.agilenutshell.com/

The test plan

A test plan documents the strategy that will be used to verify and ensure that a product or system meets its design specifications and other requirements. A test plan is usually prepared by or with significant input from test engineers.

Depending on the product and the responsibility of the organization to which the test plan applies, a test plan may include a strategy for one or more of the following:

- Design Verification or Compliance test - to be performed during the development or approval stages of the product, typically on a small sample of units.
- Manufacturing or Production test - to be performed during preparation or assembly of the product in an ongoing manner for purposes of performance verification and quality control.
- Acceptance or Commissioning test - to be performed at the time of delivery or installation of the product.
- Service and Repair test - to be performed as required over the service life of the product.

- Regression test - to be performed on an existing operational product, to verify that existing functionality didn't get broken when other aspects of the environment are changed (e.g., upgrading the platform on which an existing application runs).

A complex system may have a high level test plan to address the overall requirements and supporting test plans to address the design details of subsystems and components.

Test plan document formats can be as varied as the products and organizations to which they apply. There are three major elements that should be described in the test plan: Test Coverage, Test Methods, and Test Responsibilities.

Test coverage

Test coverage in the test plan states what requirements will be verified during what stages of the product life. Test coverage is derived from design specifications and other requirements, such as safety standards or regulatory codes, where each requirement or specification of the design ideally will have one or more corresponding means of verification. Test coverage for different product life stages may overlap, but will not necessarily be exactly the same for all stages. For example, some requirements may be verified during Design Verification test, but not repeated during Acceptance test. Test coverage also feeds back into the design process, since the product may have to be designed to allow test access.

Test methods

Test methods in the test plan state how test coverage will be implemented. Test methods may be determined by standards, regulatory agencies, or contractual agreement, or may have to be created new. Test methods also specify test equipment to be used in the performance of the tests and establish pass/fail criteria. Test methods used to verify hardware design requirements can range from very simple steps, such as visual inspection, to elaborate test procedures that are documented separately.

Test responsibilities

Test responsibilities include what organizations will perform the test methods and at each stage of the product life. This allows test organizations to plan, acquire or develop test equipment and other resources necessary to implement the test methods for which they are responsible. Test responsibilities also includes, what data will be collected, and how that data will be stored and reported (often referred to as "deliverables"). One outcome of a successful test plan should be a record or report of the verification of all design specifications and requirements as agreed upon by all parties.

IEEE 829 test plan structure

IEEE 829-2008, also known as the 829 Standard for Software Test Documentation, is an IEEE standard that specifies the form of a set of documents for use in defined stages of software testing, each stage potentially producing its own separate type of document. These stages are:

- Test plan identifier

- Introduction
- Test items
- Features to be tested
- Features not to be tested
- Approach
- Item pass/fail criteria
- Suspension criteria and resumption requirements
- Test deliverables
- Testing tasks
- Environmental needs
- Responsibilities
- Staffing and training needs
- Schedule
- Risks and contingencies
- Approvals

Daily standup vs. Daily reports

Daily meeting

Definition

Each day at the same time, the team meets so as to bring everyone up to date on the information that is vital for coordination: each team members briefly describes any "completed" contributions and any obstacles that stand in their way. Usually, Scrum's Three Questions are used to structure discussion. The meeting is normally held in front of the task board.

This meeting is normally time boxed to a maximum duration of 15 minutes, though this may need adjusting for larger teams. To keep the meeting short, any topic that starts a discussion is cut short, added to a "parking lot" list, and discussed in greater depth after the meeting, between the people affected by the issue.

Also Known As

- the "daily stand-up": from Extreme Programming, which recommended participants stand up to encourage keeping the meeting short
- the "daily scrum": by reference to the name of the Scrum framework, and alluding to the huddle-like appearance of a rugby scrum (somewhat paradoxically: see the historical note below)
- the "huddle", "roll-call", or any number of variants

Expected Benefits

- the daily meeting prevents a common failure mode of teams, where in the absence of an explicit occasion to share recent information, some critical knowledge may sometimes "fall through the cracks"

- regular peer-to-peer sharing of information in a short, focused and energetic meeting also contributes to team cohesion
- stand-up meetings are reliably shorter, more pleasant and more effective than sit-down meetings

Common Pitfalls

- perhaps the most common mistake is to turn the daily meeting into a "status report" with each member reporting progress to the same person (the team's manager, or the appointed Scrum Master) - exchanges in the daily meeting should be on a peer-to-peer basis
- a second common pitfall is a daily meeting which drags on and on; this is easy to address with a modicum of facilitation skills
- a third common issue is a team finding little value in the daily meeting, to the point where people will "forget" to have it unless the Scrum Master or project manager takes the initiative; this often reveals a lukewarm commitment to Agile
- one final common symptom: the "no problem" meeting, where no team member ever raises obstacles ("impediments" in Scrum parlance), even though the team is manifestly not delivering peak performance; this is sometimes an indication that the corporate culture makes people uncomfortable with discussing difficulties in a group setting.

Reports

Reports are a more formal way of communicating statuses, progress and issues. Target audience can vary from team members to high level stakeholders and the frequency can be mutually agreed for each reporting item based on needs (daily, weekly, monthly, per release, per test cycle, etc.)

Daily Status Report

The information that needs to be a part of an individual's "Daily status report" is:

- What did you do today?
- What are you planning to do tomorrow?
- Did you face any issues during your day? If yes, how did you resolve them or are they still open?
- Do you need any inputs for tomorrow? If yes, from whom and what are they?

Daily/Weekly Test Execution Report

- **What is it?** Generally, this is a communication sent out to establish transparency to the QA team's activities of the day during the test cycle – includes both defect information and test case run information.
- **Who should it go to?** – Normally, Development team, Environment support team, Business analyst and the project team are the recipients/meeting participants. The Test plan is the best place for you to find this information.

- Number of test cases planned for that day
- Number of test cases executed – that day
- Number of test cases executed overall
- Number of defects encountered that day/and their respective states
- Number of defect encountered so far/and their respective states
- Number of critical defects- still open
- Environment downtimes – if any
- Showstoppers – if any
- Attachment of the test execution sheet / Link to the test management tool where the test cases are placed
- Attachment to the bug report/link to the defect/test management tool used for incident management

How do we refine this information?

Show the overall status with a color indicator. Eg: Green – on time, Orange- Slightly behind but can absorb the delay, Red- Delayed.

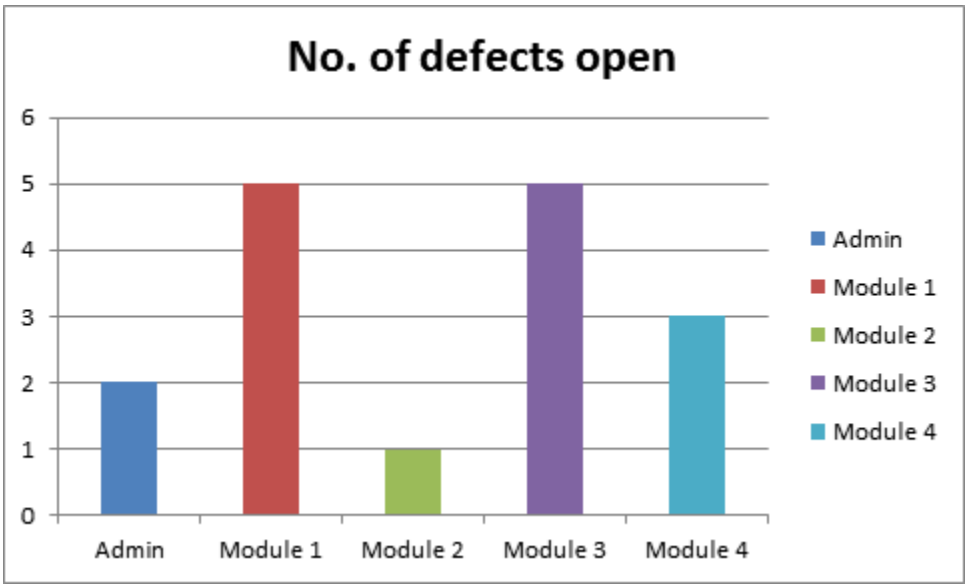
Include some simple metrics like: Pass % of test cases so far, defect density, % of severe defects; by doing this you are not just giving numbers, you are actually providing a glimpse of the quality of the product that you are testing.

If a significant phase is complete- highlight that.

If there is a critical defect that is going to block all/a part of future execution- highlight that.

If using a presentation, make sure to include some graphs to make a better impact.

For example, the below graph is a representation of the number of open defects, module-wise:



Sprint report

They are a summary of what happened in a Sprint. For example, they might contain:

- The date the Sprint started and the date it ended
- Who was in the team (including how many days each person worked)
- Which stories were in the Sprint Backlog (including their estimate and whether they were completed by the end of the Sprint)
- Metrics (e.g. the team's current velocity, how many builds happened during the Sprint, how many automated tests were built, how much the team spent / remaining funds, etc)

Retrospectives

No matter how good a Scrum team is, there is always opportunity to improve. Although a good Scrum team will be constantly looking for improvement opportunities, the team should set aside a brief, dedicated period at the end of each sprint to deliberately reflect on how they are doing and to find ways to improve. This occurs during the sprint retrospective.

The sprint retrospective is usually the last thing done in a sprint. Many teams will do it immediately after the sprint review. The entire team, including both the Scrum Master and the product owner should participate. You can schedule a scrum retrospective for up to an hour, which is usually quite sufficient. However, occasionally a hot topic will arise or a team conflict will escalate and the retrospective could take significantly longer.

Although there are many ways to conduct an agile sprint retrospective, our recommendation is to conduct it as a start-stop-continue meeting. This is perhaps the simplest, but often the most effective way to conduct a retrospective. Using this approach each team member is asked to identify specific things that the team should:

- Start doing
- Stop doing
- Continue doing

Conducting retrospectives this way is fast, easy, non-threatening and it works. A start, stop and continue meeting is very action-oriented. No time is spent focused on feelings. We don't ask team members how they felt during a sprint; were they happy or sad, warm or fuzzy.

Each item generated will lead directly to a change in behavior. The team will start doing something, or they will stop doing something, or they will continue doing something until it becomes a habit.

Many flavors have been defined on how to approach and achieve an effective retrospective. Some of the most popular recipes can be found here:

http://retrospectivewiki.org/index.php?title=Retrospective_Plans