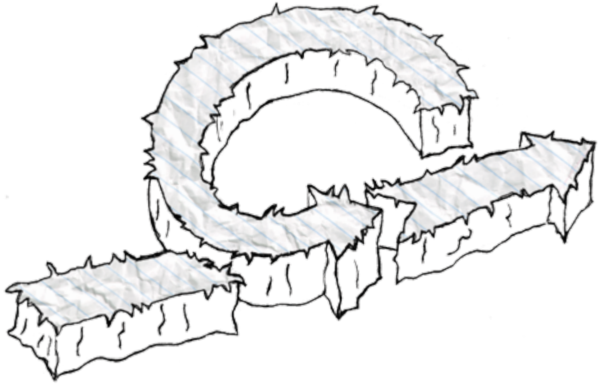
Exploring Agile: The Seapine Agile Expedition

Jeff Amfahr, Alan Bustamante, and Paula Rome

Seapine Software™



Contents

Welcome to the Seapine Agile Expedition!

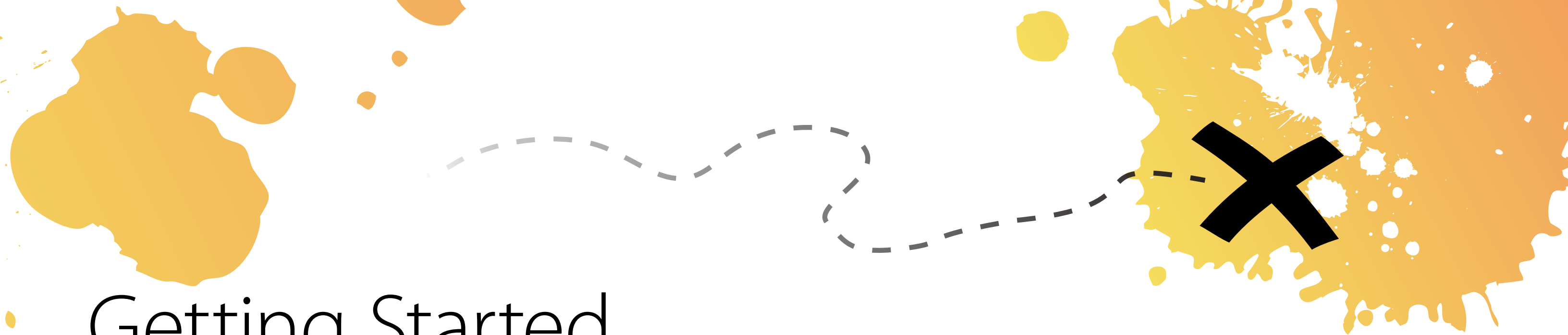
The Agile Expedition is a journey into the world of Agile. The adventure starts with the product and sprint backlogs, hikes through running your first sprint all the way to releasing your product, and finishes on the other side with an exploration of metrics. With Seapine's Agile Service experts guiding you, you'll discover how Agile development can benefit your organization and customers. Once you reach the end of the book, you may find that you're really at the beginning of your Agile journey.

Becoming Agile is not easy or for the faint of heart, but as you gain experience you will find that the transition is much more about the journey than it is the destination. Enjoy your journey!

Backlogs: The Foundation to Your Agile Success

What is the Product Backlog?

An organized and categorized product backlog is the foundation of delivering what your customer needs. The **product backlog** is used to prioritize customer requests and ensure the team is working on the most important features for your business. Agile practices work because they deliver value to the customer in increments, and give customers the opportunity to offer relevant feedback at defined intervals throughout the development cycle. An outdated backlog may cause your team to spend effort on features that are no longer the best value for the business. Worse yet, if the product backlog does not contain enough features to keep the development team busy, they may sit idle until new features are added. However, this scenario can generally be mitigated by having the team work through technical debt.



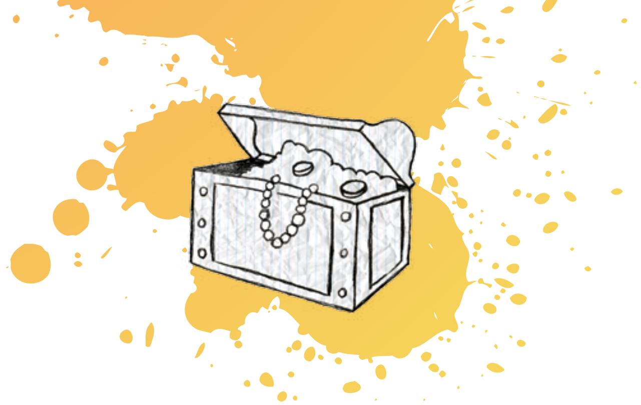
Getting Started

Before your team starts delivering features, the Product Owner must first build a product backlog for the product or project. If you're starting a project from scratch, this process is straightforward. The Product Owner meets with the customer and end users to discuss what critical problems the project will solve or challenges the customer is hoping to overcome with this product. Depending on the project, the discovery process can take anywhere from a few hours to a few weeks to outline enough features for the first few sprints.

If you're working on an existing product, building the initial backlog may be more complex. For example, a large legacy application may have thousands of feature requests, bugs, and other tasks that have been cataloged over the years. The problem isn't filling the product backlog; it's deciding which items are the most important for the team to work on next. Start by adding known higher-priority features that support your release or sprint goals. This should give the team enough to choose from for the first couple of sprints. As the project moves forward, the Product Owner will continually prune and prioritize the product backlog.

Add Features to the Product Backlog

Three types of work items fit in the backlog:

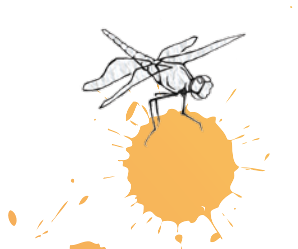


Bugs

Bugs and defects are problems found by development, testing, and end users. In a Waterfall process, testing is typically the last step of the development lifecycle and it's quite common to release code that includes defects. Bugs pile up over the years and should be included in the backlog and prioritized accordingly.

Technical Debt

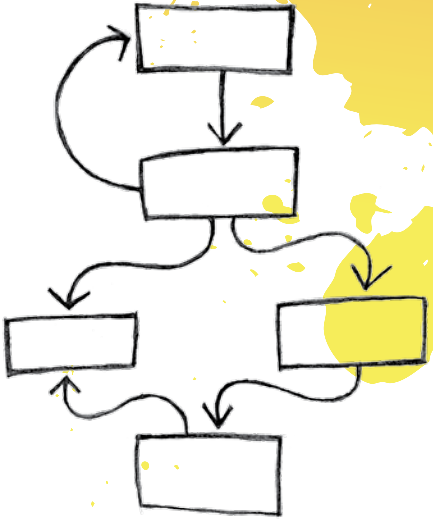
Over time, the direction and scope of a product usually changes. Performance and scalability expectations change. New technology or best practices become available. Everyone can agree that it's a good idea to update the existing solution to address these types of issues. In practice, however, it can be difficult for the Product Owner to prioritize them over highly visible features requested by customers. These types of backlog items are often referred to as technical debt because



they frequently accumulate over time. Examples of technical debt include upgrading to the latest third-party libraries, making architectural changes to support better scalability and configurability, or refactoring the source code for easier maintainability in the future. These tasks need to be included in the product backlog and prioritized along with defects so they have visibility in the planning cycle.

New Features

Feature requests come from a variety of sources, including end users, sales, support, and product management. They can be the hardest to prioritize as you balance the competing needs of satisfying your existing customer base, satisfying the needs of near-term sales opportunities, and working toward a longer-term vision of your product. The Product Owner must routinely monitor these sources and arbitrate potentially conflicting requests to ensure the backlog contains the features that will attract new customers and build loyalty with existing customers.



Organizing the Product Backlog

Next, you need to classify and prioritize the product backlog. It doesn't do any good to have a backlog if you can't quickly analyze it by different criteria. Classification and prioritization are largely manual processes, but this goes faster than you might expect.

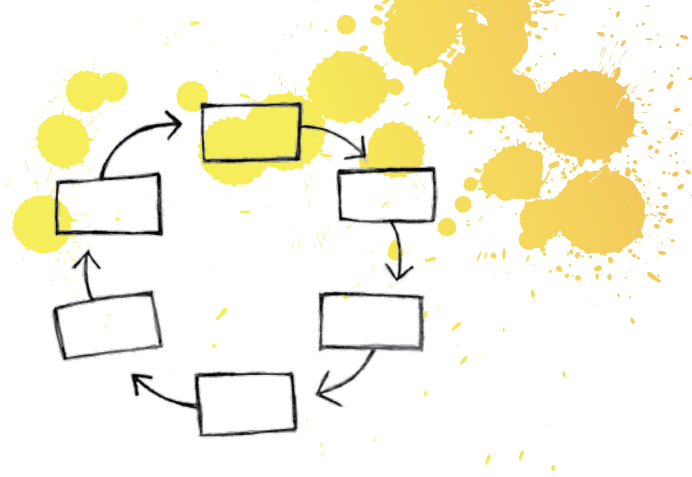
When first getting started, don't worry about organizing your entire backlog. Find the features that are clearly high priority, mark them as high priority, and ask the development team to classify them as outlined below. Slowly expand prioritization and classification to more and more of the backlog as needed, based on your expected sprint schedule.

To learn about organizing backlogs with TestTrack folders and custom fields, check out our *Backlog Tagging Best Practices* blog post: http://blogs.seapine.com/2010/06/backlog-tagging-best-practices.

Classification

We recommend starting with the following two classification categories: **functional area** and **theme**. Functional area is simply the area of the product a defect, technical debt, or new feature applies to. Developers can help classify work items because they have an in-depth understanding of the underlying code base in the project. For example, if refactoring the user security model is a top priority, then planning a sprint to address that area of the product is easier if your backlog is tagged by functional area.

Classifying by theme is really about grouping features together to make sprint planning a smoother process. Tagging each feature with a theme gives you greater flexibility when planning a sprint. Let's say that customers are struggling to find data in the application you're developing. If you tag your backlog with different theme tags—usability, for example—building a sprint to address immediate customer issues is easy. Simply pull out everything tagged 'usability' and use that subset of features to plan your sprint.

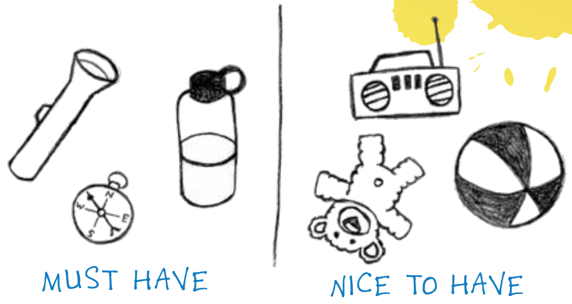


DANGER! Low priority items always seem to end up at the bottom of the pile. This is where classification helps by making it much easier to find related items, regardless of priority, during planning.

Prioritization

Next, the Product Owner prioritizes the features in the product backlog by business value. This is where the Product Owners' understanding of the needs of their customers and end users is critical. Although feature value may not necessarily be quantifiable in monetary value, features in the product backlog should have clear business value for them to be considered for development.

Ideally, the product backlog will have the highest priority feature at the top and the lowest priority feature at the bottom. However, in the case of large feature sets, it may be easier to use a bucket approach. This approach uses qualitative descriptions, such as low, medium, and high, to group features with similar priorities. After features are grouped into buckets, you can refine the buckets as many times as you want. An organized backlog, with a good classification and prioritization scheme, is essential before the development team can start working on the highest value features.



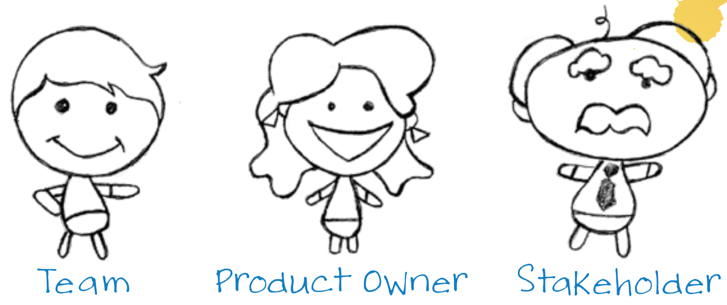
DANGER! Agile methodologies ensure the product reflects what the customer needs at the end of every sprint. It's easy to get side tracked and prioritize features according to the "loudest customer" or "this week's sales opportunity." If you use the bucket approach, set tolerances (e.g., no more than five in high priority) to ensure there are never more than a certain number of items in the high priority bucket. If you add something to the high priority bucket, you have to take something else out.

Maintaining the Product Backlog

Building and organizing the product backlog is always a work in progress. Following an Agile methodology is about making sure that what you're working on is what the customer actually needs. The Product Owner must commit to maintaining the backlog on a regular basis for a few reasons.

First, customer needs change over time, and the backlog must stay in sync with those changes. For instance, flip phones were all the rage a few years back, then it was mobile phones with QWERTY keyboards, and now it's touch screen smart phones.

The backlog for the input device of a mobile phone these days is much different than it was a couple years ago. The Product Owner is responsible for staying on top of those trends and adjusting priorities accordingly.

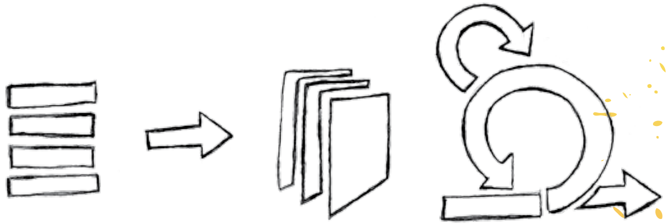


Implementing features can also change or eliminate the need for other seemingly unrelated features. For example, if a feature to implement credit card processing is in the product backlog, but another feature to integrate PayPal has already been implemented and proves to be better, then the credit card processing feature may drop from the product backlog. Again, the Product Owner is responsible for managing those changes, and better categorization can make that process easier.

Finally, the product backlog is viewable by everyone. The Product Owner is responsible for vetting new ideas against existing ideas, and determining whether they should be prioritized into the backlog.

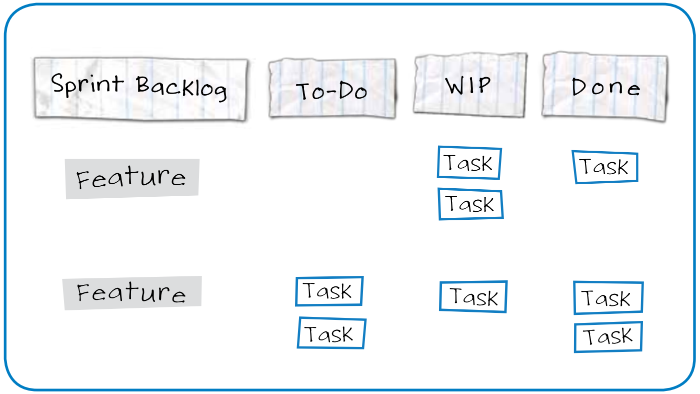
Populating the Sprint Backlog

The sprint backlog comprises the sized features that the team selects from the product backlog for a specific sprint. The team commits to deliver every feature in the sprint backlog within the current sprint. To help establish quick wins and build confidence, teams new to Scrum or other Agile methods should take on less work than they think they can handle in early sprints.



DANGER! It's important for the development team to understand the level of commitment they are making. Unlike Waterfall, where scope often gets removed to meet deadlines (leading to lack of trust with the customer), Agile teams value the commitments they make in a sprint to build trust with the customer.

Once the team knows what features will go into the sprint backlog, the development team decomposes the features into tasks. Tasks should be no more than 16 hours, or two man days, in duration. This helps the team understand the problem better and get a good idea of how much work is involved. The team should not assign or grab a task until it goes to work in progress. This ensures task ownership is not pre-determined and leaves tasks open for other team members to grab.



Backlogs in a Nutshell

Proper planning and organization is critical to delivering what your customers need, and that's where product and sprint backlogs come into play. Here's what we learned:

- The Product Owner is responsible for the product backlog, prioritizing customer requests, and ensuring the team is working on the most important features.
- The product backlog can include bugs, technical debt, and new feature requests, which can be organized into classification categories, such as functional area and theme.
- The Product Owner should then prioritize features in the product backlog, typically based on business value.
- The product backlog must be maintained on a regular basis to ensure the development team is always delivering the right product features.
- The team selects features from the product backlog that they will commit to delivering in the current sprint. During sprint planning, the selected features are broken down into their related tasks.

The Art and Science of Reliable Agile Estimating

Nothing to Fear but Fear Itself

Estimating. The mere thought of it strikes fear into the hearts of even the most stalwart project teams. The customer wants to know how long the project will take, but you don't want your team locked in to a commitment they may not be able to meet.

Relax. When it comes to estimating Agile projects, you have nothing to fear but fear itself.

In the beginning, you're going to be terrible at estimating. In fact, you're going to downright stink at it. And that's OK. Estimating is more of an art than a science and, like all art, it takes practice. It will take time for team estimates to converge naturally.

As you transition to Agile, your initial estimates will be all over the map. Even if you are an experienced estimator in other development methodologies, expect a learning curve. It generally takes several iterations, sometimes over the span of months, for the team to get good at Agile estimating. This is an important expectation to set with your team and with management. Fortunately, Agile allows you to fine tune your estimates as the project progresses.

When Does Estimating Occur?

Estimating on Agile projects occurs iteratively at the beginning of each sprint for features and throughout the sprint for tasks. If you're using Scrum, then the first day of the sprint is when the two-part sprint planning occurs. In other Agile methodologies the planning day is broken up into two parts: release planning and iteration (or sprint) planning. Although Ken Schwaber and Mike Beedle do not explicitly make the distinction in their *Agile Software Development with Scrum* book, Scrum's two-part sprint planning meeting follows essentially the same process.

DANGER! To avoid confusion and wasted time, do not start the first sprint with an empty product backlog. In general, the Scrum Master should work with the Product Owner to help populate the product backlog with features prior to the first planning meeting.

Who is Involved with Estimating?

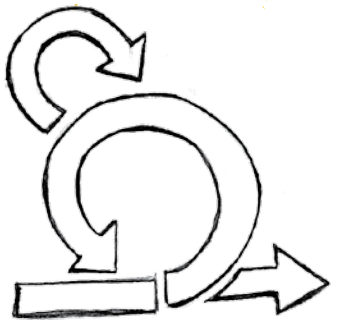
During release planning, the team sizes the features in the product backlog. Participants in the sizing effort include the core development team (developers, testers, tech writers, etc.), the Scrum Master, and the Product Owner or customer.

Wait, the customer? Yes, that's right. The customer representative—or Product Owner—needs to be involved to clarify the value the feature provides. The Product Owner can also provide the team with a good understanding of what they are estimating.

While the Product Owner and Scrum Master provide feature clarification and facilitation, only the development team should provide estimates because the team is accountable for doing the work. This is a departure from traditional methods, where team leads or development managers generally provide estimates. For a humorous take on this concept, check out the Vizdos and Clark cartoon about the chicken and the pig: www.implementingscrum.com/2006/09/11/the-classic-story-of-the-pig-and-chicken.

How Long Should it Take to Estimate?

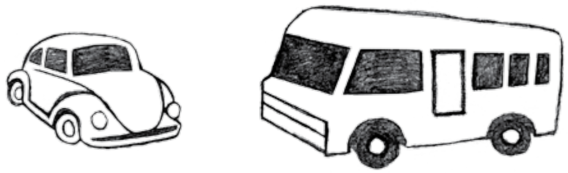
It depends on the team's familiarity with each other and the sizing techniques used. At first, it may take several hours to size only a few features. As the team becomes more experienced, the amount of time needed will decrease. Ultimately, teams should not spend more than a few minutes estimating each new feature. If the team is still struggling with estimating after a few sprints, address the issue in the sprint retrospective and consider hiring an Agile coach.



The Point System

Points are used for relative sizing of features, which is different from traditional methods where hours are usually used. Points can be used to size any type of feature or requirement artifact, such as a use case or user story. Point values can vary, but a common value range used is the Fibonacci sequence, which generates subsequent numbers in the sequence by adding the previous two numbers.

To illustrate this concept, assume we're comparing the following items:



Using the Fibonacci sequence numbers two, three, five, and eight, assume we assign the car a value of two. If we want to compare the size of the two vehicles, we might say the bus has a value of five, which means it is more than twice the size of the car (larger than three) but less than four times the size of the car (smaller than eight).

Points are also useful for determining team velocity. Velocity can be used to determine when the amount of work currently in the project will be done. For example, if there are 40 points in the project and the team shows an average velocity of five points per sprint, then the team will most likely be finished in eight sprints, assuming all else is equal.

Over time, a team develops a feel for the relative sizes of things. When a team first transitions to Agile methods, however, the initial estimates among the team members will not converge easily. Don't panic! This is normal and will get better with time.

DANGER! When sizing features using points remember that points are relative and specific to the skills of the team, which means it's impossible to compare the performance of two teams based on point values. So, a team that completes 40 points worth of features is not necessarily doing more work than a team that is doing 20 points.

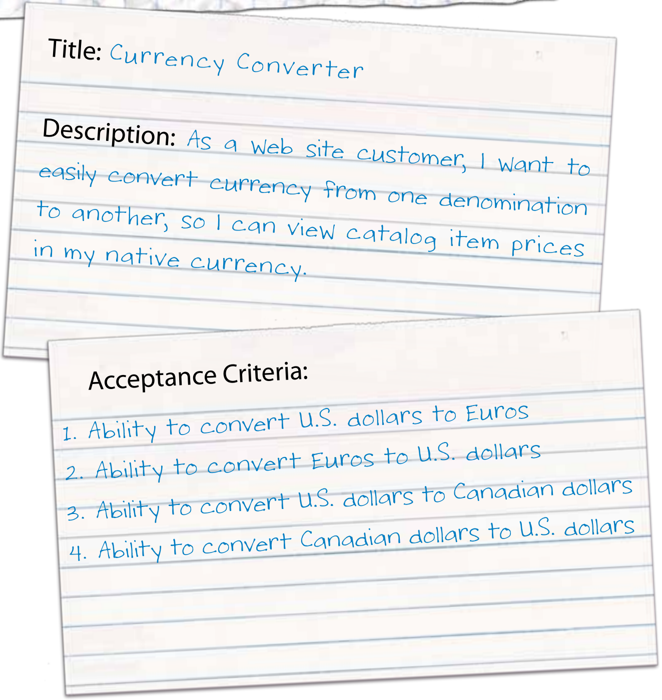
While points have the additional advantage of determining velocity, some teams prefer to use a qualitative measure, such as T-shirt sizes. Using this sizing method, teams size user stories based on small, medium, and large sizes.

User Stories

While there are many ways to write requirements for an Agile project, one method that is quickly becoming preferred is **user stories**. In fact, in Agile projects, the term "requirements" is rarely used, which is why we have used the word "feature" up until this point. Traditional projects that define all requirements up front often deliver features at the expense of value (what the business really needs). They may also deliver only a subset of features because the scope is reduced by the end of the project.

In contrast, an Agile project expresses features in terms of business value. One way to do this is through user stories, which are not specific to any Agile methodology. When adding user stories to the product backlog, the Product Owner should be asking, "What business value does this feature add?"

Sample User Story


Title: Currency Converter

Description: As a Web site customer, I want to easily convert currency from one denomination to another, so I can view catalog item prices in my native currency.

Acceptance Criteria:

1. Ability to convert U.S. dollars to Euros
2. Ability to convert Euros to U.S. dollars
3. Ability to convert U.S. dollars to Canadian dollars
4. Ability to convert Canadian dollars to U.S. dollars

DANGER! User stories are not meant to be detailed written documents. If you can't fit the description of a user story on one 3"x5" index card, rip it up and start again. As Mike Cohn explains in *Agile Estimating and Planning*, user stories are a "promise for a conversation." They are a placeholder for the conversation that needs to happen daily between the team and the Product Owner during the sprint.

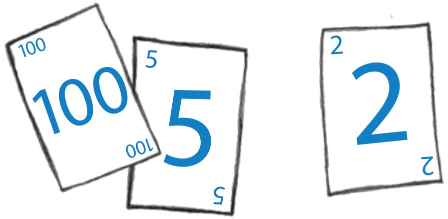
Software tools provide much more space than an index card, but this encourages bad story writing practices. This is one reason we recommend starting with note cards first to establish good habits, then moving to a tool, like TestTrack, for user story management.

Estimating with Story Points

EXAMPLE: *We project our currency converter user story on the screen in the team room, and everyone picks their story point estimates using the Fibonacci sequence. After everyone selects an estimate, we all reveal our story point estimates. We get the following numbers: 3, 2, 5, 5, and 8*

When estimating user stories with points, the term used is **story points**. As mentioned earlier, estimating user stories occurs iteratively throughout the project. During release planning, the Product Owner talks about new stories that have been added to the product backlog as well as the objective for the sprint or release. If any of the new stories are likely to end up in the next couple of sprints, the team should take the time to estimate as many stories as they can within the release planning time frame.

The team discusses each new story with the Product Owner until they have a good enough understanding of the user story to estimate its size. The size estimate will be relative to other user stories in the deck. Relative sizing reduces the pressure of having to be exact, which is one reason estimating in hours is discouraged. Estimating user stories in hours has the stigma of exactness or predictability.



One method for acquiring estimates is to play Planning Poker. The team uses a deck of special cards that have some number sequence, such as the Fibonacci sequence, with each team member receiving a set of cards from the deck. Team members pick a card from their set that reflects the size estimate they have chosen. When all players are ready to reveal their estimates, the cards are turned over and shown to the other players on the team. If there are outliers or a broad range of numbers, Planning Poker is great because it forces the team to openly discuss their differing opinions. For more information about Planning Poker, check out Mike Cohn's web site: www.mountaingoatsoftware.com/topics/planning-poker.

Learn how to configure story points in TestTrack: http://blogs.seapine.com/2010/06/configuring-story-points-in-testtrack/.

DANGER! Changing who is on the team may change how stories are estimated and the amount of points the team can deliver in a sprint. For example, Ray was transferred to another group and Steve was added to Ray's old team. It took a couple of sprints before Ray could get in sync with the team on sizing stories. Also, because Ray was not familiar with the code the team worked with, the team took on fewer story points in subsequent sprints until he was up to speed.

Handling Complex Stories

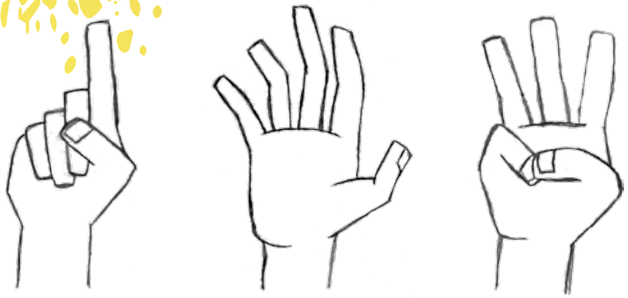
User stories with an unusually high number of dependencies or with many unknowns should be discussed minimally during planning. If dependencies are not well understood or there are a lot of unknowns around what the story should accomplish, then estimates will generally be larger to account for the additional complexity. In some cases, the team will not be able to estimate until more is known about the story. In these cases it's important for the Product Owner to keep enough stories in the product backlog for the team to discuss.

DANGER! User stories that are too big to estimate or complete in one sprint are often called epics. Epics are generally placeholders for stories that will be created from the epic. Product Owners can use epics for planning purposes, but should expect to break the epic down in to smaller stories before the team tries to estimate. After the epic is broken down, it is usually thrown away.

Reaching a Consensus

After you have everyone's story point estimates, you need to build a team consensus on what the final story point should be for the user story in question. Address any story point estimates that are radically higher or lower than the average, and ask the team members who gave the outlying estimates why they think the user story is easier or harder than other team members think it is. They may have misunderstood something or they might be aware of a problem no one else has thought of.

EXAMPLE: For our currency converter user story, the outlying estimates were 2 and 8. While discussing the outliers, some team members realize there is more complexity, while others realize there is less complexity. We resubmit our story point estimates, and the resulting numbers change to 3, 5, 3, 5, and 5. The team decides on a final story point estimate of 5.



At the beginning of the project, you should establish some method for determining how the team will move forward if there is not complete consensus. Collaboration requires working together as a team, but it does not necessarily mean there will be total agreement among the team on every decision. Before these situations arise, decide together how to move forward when you can't achieve a complete consensus.

Fist of Five: Fist of Five is a consensus building technique that allows team members to quickly place a vote ranging from zero to five. After conflicting story point estimates are discussed, team members hold up the number of fingers on one hand to indicate their level of agreement with the resulting decision. A fist indicates complete disagreement and five fingers indicate full support. Raising three fingers indicates that, although there are reservations, the team member is willing to move forward with the decision.

Estimating Task Hours

During the last part of release planning, which is covered in the next chapter, the development team pulls the stories they agree to deliver in the current sprint from the product backlog. These stories become part of the sprint backlog for the current sprint.

After the team identifies the stories that will go in to the sprint backlog, they begin sprint planning. This is where the development team breaks down the stories from the sprint backlog into tasks, which are needed to implement the story. Examples of tasks include “build customer class,” “migrate customer billing data”, and “add customer name field to database”. Each task is assigned an estimate in hours. Like with estimating user stories, estimating task hours is a development team activity.

DANGER! The Product Owner should remain available for the team during planning. However, once sprint planning starts, the team should have enough information to identify tasks to get started on. Sitting with the team while they decompose stories into tasks may not be the best use of the Product Owner’s time.

Ideal Days

Task estimates are in ideal hours, based on an ideal day. The team should estimate how long it would take to complete a task assuming they had no interruptions, no meetings, and plenty of alertness and energy.

DANGER! Task hours should not be larger than 16 hours, or two working days, because smaller task estimates increase the overall understanding of the problem. For example, a testing task estimated at 80 hours for a two-week sprint does not sufficiently break down the types of testing that need to be done.

Avoid Analysis Paralysis

Whether estimating stories or task hours, it is important for the team to remember that there is a law of diminishing returns to the time spent estimating. In the case of user stories, if the team cannot decide between a point size of five and eight, they should take the eight, which is the more conservative number. In the case of tasks, the team should break down stories into tasks and estimate hours until they reach a natural stopping point. As the sprint progresses, tasks will be added and removed. Hours will also be updated regularly, so there is no need to sweat over getting it right the first time

Estimating in a Nutshell

Everyone has difficulty estimating in the beginning, but there's no reason to fear it. Here's what we learned:

- Estimating on Agile projects occurs iteratively, at the beginning of each sprint for features and throughout the sprint for tasks.
- Include everyone—especially the customer representative or Product Owner—in the initial, high-level estimate.
- Assign story points to indicate the level of difficulty for each user story.
- Allow for dependencies when estimating, and make sure each user story is broken into its component parts.
- Break down stories into tasks, and then estimate hours for each task.
- Build a consensus on story point estimates.

Mapping the Journey: Release and Sprint Planning

Do You Know the Seven Ps?

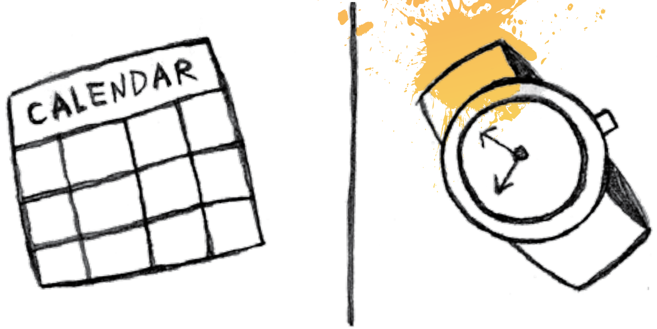
When training new recruits, drill sergeants in the British Army instill a healthy respect for the "Seven Ps." Forget the Seven Ps, British soldiers are told, and you are not likely to live long on the battlefield. Likewise, ignoring the Seven Ps when you're beginning a new software release may mean the demise of your project. So what are these Seven Ps?

Prior Planning and Preparation Prevents Pretty Poor Performance

(The British drill sergeants use a more vulgar "P" word than "pretty," but we've sanitized it for your protection.)

So far on your Agile Expedition, you've prepared by establishing your product backlog and completing your initial, high-level estimate in the form of story points. Now it's time to plan how you're going to tackle the project.

Release Planning vs. Sprint Planning



In an Agile environment, planning happens on two levels: **release planning** and **sprint planning**. Release planning happens on the first day of every sprint (or iteration) and focuses on the longer-term, strategic goals for the project.

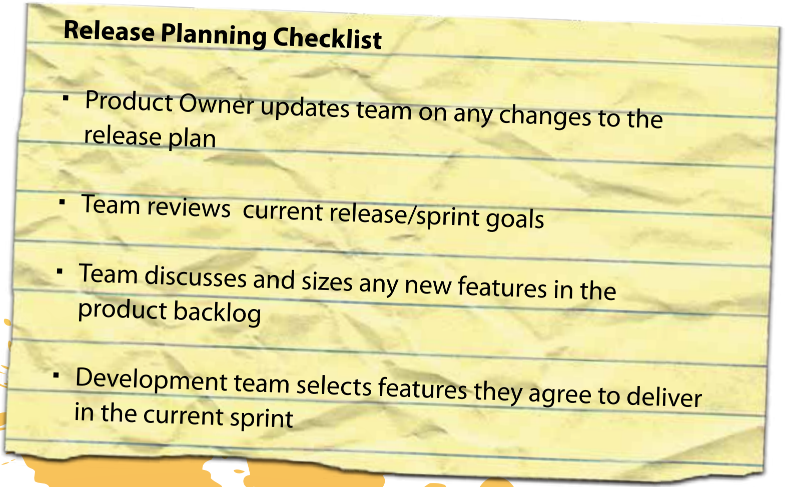
The release plan starts with the prioritized and estimated product backlog. It reflects the balance between business value and delivery capability. The release plan establishes the date for the release and the number and length of sprints in the release. Release planning, which also involves estimating new features, concludes with an understanding of what features will go into the sprint backlog for the current sprint.

Sprint planning also happens the first day of a sprint, but deals with the specifics of each sprint (or iteration, if you prefer). The output of sprint planning is the full sprint backlog.

By the time you finish your release and sprint planning, you'll have a roadmap of your product's releases. Let's take a closer look at these two key Agile activities.

Release Planning

After the Product Owner assembles their backlog of user stories, it's time to sit down with the team for a release planning session. During this meeting—usually up to four hours long—your team will review the project's strategic goals, commit to the goal for the current release, update the goals for future releases, provide story point estimates for stories in the product backlog, and then create a schedule of the sprints in the current release.



Release Planning Checklist

- Product Owner updates team on any changes to the release plan
- Team reviews current release/sprint goals
- Team discusses and sizes any new features in the product backlog
- Development team selects features they agree to deliver in the current sprint

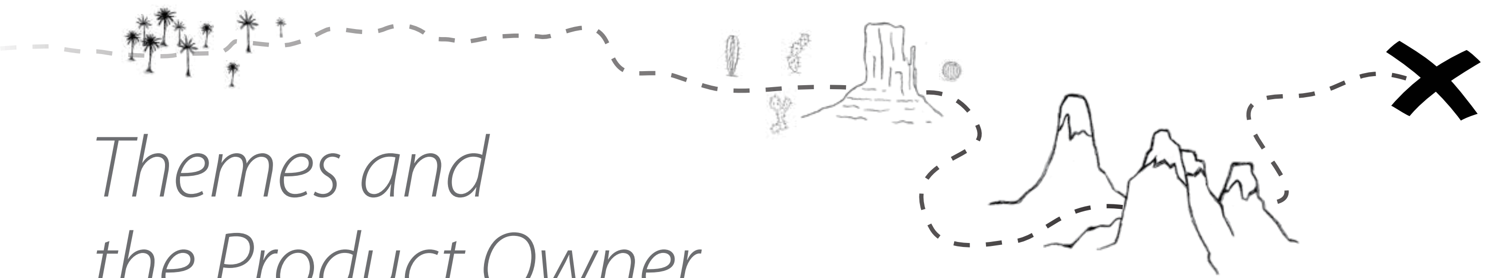
Use Themes to Help Plan Releases

When planning releases, grouping user stories by theme is helpful for a couple of reasons. First, working on user stories with related functionality usually goes faster than working on features that are scattered throughout the application. It's generally more efficient to focus on one functional area than it is for the team to divide its attention among multiple areas.

Why is it more efficient? When you change from one functional area to another, you switch to a new context. This switch requires an adjustment period, which slows down your (or your team's) productivity. By grouping work into functional areas, you minimize the amount of context switching, and therefore you get more done faster.

It's also more efficient because themes help organize the backlog so you can quickly find items with related functionality.

EXAMPLE: Our customers are struggling to find the data they need to complete their weekly reports in the system we're building for them. The solution involves a combination of user interface changes to existing screens and a new business rule for automatically calculating status. We create a report usability tag for items in the backlog related to fixing this issue, which makes it easy to then build a sprint to address the issue. The Product Owner prioritizes the stories in the report usability theme into the product backlog and the team pulls the stories they agree to complete in the sprint.



Themes and the Product Owner

The Product Owner should identify themes before the release planning meeting because themes will help them focus on what is really important and the value that will be delivered to the customer and business. From the Product Owner's point of view, themes typically reflect some aspect of business value—a benefit to the business or customer.

Possible themes could be:

- Improve administration
- Provide online payment options
- Meet compliance

Themes reflect the Product Owner's view and all stories in a theme will not be of equal importance. All stories do not need to be completed to deliver on the theme benefits.

Set Release Goals

The Product Owner should be prepared to discuss the release goals at the release planning meeting. These goals might be adjusted if the team has insights that would make for a better set of goals or if they're not confident they can deliver on the original goals.

It's important for the team to understand the customer and business needs driving the release goals, and that the Product Owner and management understand the challenges involved in meeting the release goals. At the end of this meeting, everyone should agree on the goals for the release.

To help with release planning, perform the following tasks:

- **Select a desired sprint length**. If the team is just starting the project, they will need to select a sprint length. The sprint length can be as short as one week and as long as six weeks. Consistency here is crucial because the team will set a natural rhythm or cadence for team operation and product releases for the remainder of the project. A consistent sprint length also helps stakeholders outside the team with planning.

  Consider the following when helping the team determine the initial sprint length:

  1. Team experience with, and attitude toward, Agile projects and practices.
  2. Team's ability to deliver based on team factors, such as skills and maturity, and organization factors, such as management support and environment availability.
  3. Any known dates that affect delivery of work, such as release dates.

DANGER! Keep the following Agile Manifesto Principle in mind: "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale." The team should always be looking for ways to deliver software faster without sacrificing quality. Sprint lengths on the longer side of the delivery spectrum, five to six weeks, have room for improvement.

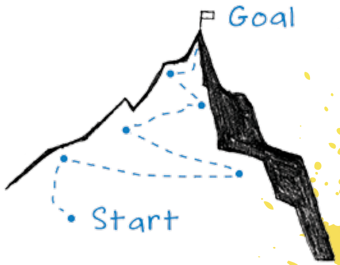
- **Set target release dates**. After the sprint length is determined, the team can set target release dates. Release dates may be driven by external factors, such as competitors, trade shows, or contractual requirements, or internal factors, such as the start of the fiscal year or establishing a consistent, repeating schedule of releasing every six months. For example, the team decides on a four week sprint. So, if the team decides to release every sprint, then they will be able to deliver code to production every four weeks. If the team decides to release every two sprints, then the team will deliver code to production every eight weeks. A release plan can be built around this schedule.

DANGER! While the team may decide to release every two sprints, the team must produce releaseable code at the end of every sprint. So, at the end of the first sprint in a two sprint release cycle, the team must have production-ready code.

- **Tag the release with a goal or goals**. Once the release target dates are determined, the Product Owner can tag release goals to each release. Release goals can be represented by themes or epics. These goals should not be misinterpreted as commitments by the team.

DANGER! Except for the current sprint, release goals attached to future release dates won't have implementable features associated with them. Remember, each team plans on a sprint by sprint basis, so features for a future sprint won't be known until the sprint is planned.

Agile teams establish a consistent release schedule, which creates a regular rhythm for the project. Everyone involved with the project, both inside and outside the team, knows when to expect a new release.



Calculating Velocity

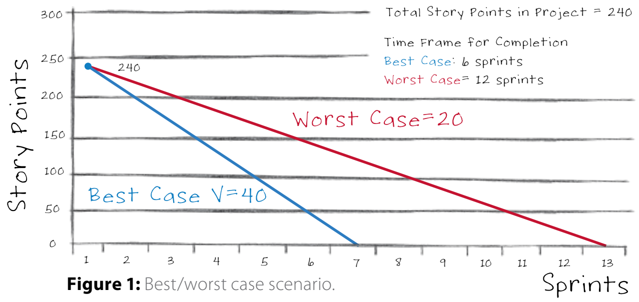


Figure 1: Best/worst case scenario.

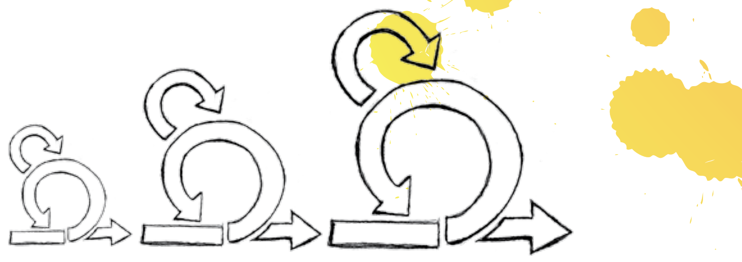
In the last chapter, we discussed estimating, story points, and velocity. As a reminder, **velocity** is the average feature or story points a team completes during a sprint. It simply helps you determine when your project will be completed.

After velocity is calculated, the team can plan how many sprints it will take to complete the project. The calculation is straightforward. For example, if the team's known velocity is 20 points per sprint, it should take them about 12 sprints to complete a project with 240 points in it.

For a project that is already in progress, velocity can be determined by looking at past sprints. However, if the team doesn't have an established velocity, a range should be used. The range, which is given as a best/worst case scenario for project completion, is determined by having the team estimate their maximum and minimum points per sprint for features that have already been estimated.

For example, the team may estimate 20 points per sprint as their minimum velocity and 40 points per sprint as their maximum velocity. This information can be used to estimate the best/worst case scenario as shown in Figure 1. As you can see, best case is 6 sprints and worst case is 12 sprints.

Sprinting Toward Change



Your initial release plan is really more of a rough draft. You've got enough of a blueprint that you can get started, but you'll need to continually revise and correct the release plan as you go. This is a key part of an Agile process.

One reason for this continual planning is that it will take a few sprints before you get an accurate idea of your team's true velocity. Also, sprints won't always deliver everything you plan for them to deliver, or they may deliver more than the plan calls for. And then there are staffing changes, customer changes, and other unforeseen problems that will impact your release plan.

The most important reason, however, is that business and customer needs change over time, sometimes rapidly. It's important to remain flexible about future release goals to ensure you are always delivering the highest value for the customer.

Zero In

Is your team new to Agile? You might want to plan for a **Sprint Zero**. Sprint Zero is a sprint that happens at the beginning of the project. It's used for things like getting organized, developing a backlog, setting story points, prioritizing user stories, and other technical and logistical issues. The Sprint Zero goal may be to simply build the backlog.

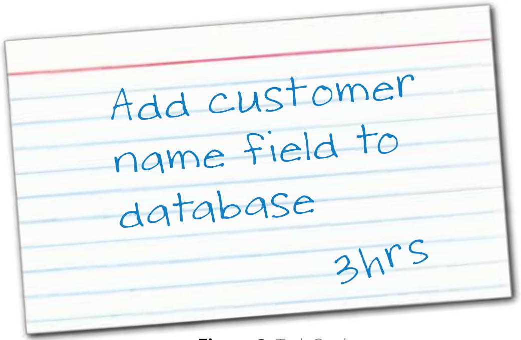


Figure 2: Task Card

Sprint Planning

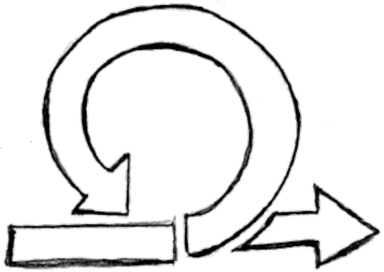
During sprint planning, the development team breaks down the features they selected for the current sprint in to tasks. For each task they write down the hours necessary to complete the task.

Notice there is no name on the task card in Figure 2. Tasks are not assigned until they move to “work in process.” Assigning tasks as they move from “to do” to “work in process” leaves the tasks open to anyone and encourages team ownership for the work in the sprint.

The development team should break down features until they reach a natural stopping point. Trying to identify every task is an exercise in futility. Tasks will be added, removed, and updated as the sprint progresses. While the time needed for sprint planning will depend on the project characteristics, as a rule of thumb teams should expect to spend up to four hours in sprint planning for a 30-day sprint.

DANGER! The Product Owner may not stay for the sprint planning session. Because the team discussed the sprint backlog features during release planning, the Product Owner usually does not need to stay while the development team breaks down the features in to their respective technical tasks.

With your release planning and sprint planning finished, you've got a roadmap for the release. Learn more about using TestTrack for release planning: http://blogs.seapine.com/2010/07how-to-use-release-planning-in-testtrack/.



Hardening Sprints

Where it makes sense, **hardening sprints** can be scheduled anytime in the release cycle. You do not add any new features in this sprint. Instead, hardening sprints are used to perform stabilization, bug fixes, and testing. Hardening sprints allow teams to focus on paying down the technical debt that has accumulated over the course of the project. The result is a more manageable and maintainable codebase.

DANGER! Hardening sprints are especially important for large or complex projects. Technical debt accumulates faster as lines of code and components that need to be integrated increase. Make sure you schedule time for hardening sprints on larger projects.

Got a Question? Spike it!

Chances are, your team doesn't know everything. Perhaps a backlog item exists and the team doesn't have enough information about it to be able to deliver an estimate. In such cases, you might need to plan for a **spike**.

A spike focuses on researching an issue or technology. The goal of the spike is to answer a specific business or IT question. Spikes can last from a few hours to a couple of days or more, depending on the problem. For large spikes, create a user story and estimate it at the same as other stories so the effort can be incorporated into a sprint.

DANGER! Don't allow spikes to go on indefinitely. There must be a limit on the spike's duration in order to focus the research effort and reduce the amount of time spent on tangents.

Release and Sprint Planning in a Nutshell

Release and sprint planning help keep your project on track. Here's what we learned:

- Release planning addresses long-term, strategic business goals.
- Sprint planning deals with the specifics of each sprint. The output of sprint planning is the full sprint backlog.
- Themes help organize the product backlog so you can quickly find items with related functionality.
- Velocity tells you how much work your team can complete per sprint and per release.
- A Sprint Zero can help a new team get ready for their first Agile project.
- Hardening sprints allow teams to focus on paying down the technical debt that has accumulated over the course of the project.
- Spikes are used to research and answer a specific business or IT question.

Marching Along: Daily Activities

Hitting the Trail

So far, we've covered the planning stages—building the product backlog, estimating effort and hours, and planning the release and sprints. Now that the planning is complete, what happens during the rest of the sprint? That's what you'll find out on this leg of the expedition. It's time to hit the trail and begin the work of building the product—one sprint at a time.

No matter how long sprints are, Agile teams feel there are some basic activities that are important to perform every day, such as:

- Working through stories, tasks, and testing
- Identifying and removing impediments
- Monitoring issues
- Updating progress
- Holding daily stand-up meetings
- Updating the plan based on progress information and new data

Working through Stories and Tasks

When you begin a new sprint, all the team members should have a list of the stories they've agreed to complete in that sprint. A well-functioning Agile team is adept at trading off tasks during the sprint.

For trade-offs to occur, however, team members must speak up when they start to run into trouble with a task, instead of waiting until there's little time left and the options for solving problems are limited. Conversely, if a team member finishes a task early or has expertise in an area that others lack, it's their responsibility to help other team members, even for tasks that are outside their usual responsibilities.

Two Heads Are Better than One

As you get into the work, you might find you need to partner with someone to get things done. For example, a developer might need to partner with another developer, a user interface designer, or a tester. Team members partner for as long as necessary to accomplish their objectives.

Pair programming, which is a common technique used by many Agile teams, involves two developers working together to design and program. Proponents of pair programming cite significant improvements in quality and higher productivity as benefits.

Pair programming also helps ensure consistency in code because there are always two people who can make sure standards are followed. For more information about pair programming, check out *Pair Programming Illuminated* by Laurie Williams and Robert Kessler.

Removing Roadblocks

At some point in the project—or, more likely, at several points—your team will run into roadblocks that will impede progress.

An **impediment** is anything that prevents a team member from performing work as efficiently as possible. It can range from a large issue (“I need data from another team to test this report”) to a small hassle (“I need a new mouse”).

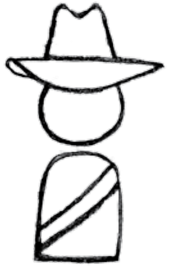
One of the Scrum Master’s responsibilities is to help remove impediments for the team. The Scrum Master can only be effective, however, if the team reports issues as soon as they are discovered. At the daily stand-up meeting (or daily Scrum), the Scrum Master records new impediments and reports on previous impediments that have been resolved.

Enter the Scrum Master

We’ve mentioned the Scrum Master in previous stops on the Agile Expedition, but we haven’t really explained what the Scrum Master does. The Scrum Master is responsible for ensuring Agile values and practices are followed, removing impediments, and ensuring the team stays productive. The Scrum Master fosters close cooperation among everyone involved with the project, and shields the team from external interferences.

DANGER! The Scrum Master does not work in a vacuum. Everyone on the team has some level of responsibility for helping to remove impediments. The team, including the Scrum Master and the Product Owner, should determine who is the best person for each situation.

Open Up



A key Agile practice is making sure the status of the team's progress is always visible to the team, management, and other stakeholders. Visibility builds management's trust and confidence in the team, which is necessary for the team to become self-organizing and self-empowered—two attributes shared by the most successful Agile teams.

To foster trust and openness, everyone working on the project must be comfortable discussing the true status of their tasks. This can be difficult for teams transitioning from more traditional development environments, where finger pointing and the blame game are unfortunately more common than we might like to admit.

The Scrum Master, who acts as a servant leader, is critical to a successful adoption of a more open environment. This can only happen if the Scrum Master "walks the talk" and models open behavior. The Scrum Master must facilitate problem solving by the team and avoid criticizing individual team members.

Handling Production Issues

Before going home at the end of the day, everyone on the team updates their tasks with the amount of work (expressed in hours) remaining. Task hours remaining in the sprint will fluctuate as tasks are added and removed, and hours for existing tasks are updated.

EXAMPLE: Barb took on the task of creating a new report, estimating her time at two hours. She assumed Joe had already created the tables for the report, but he had not. So Barb adds "create tables" as a task and adds it to the sprint backlog.

Update Your Progress

With Agile, code goes into production much sooner than it does with traditional development methodologies. Production code needs to be maintained even before the project is complete. How do you handle production defects? It depends on which team is responsible for maintenance support and the severity of the defects. If a separate team is responsible for maintenance support, the project team can focus strictly on project work.

However, if the project team is also responsible for maintenance support, make sure you allow time for emergency production issues when the sprint is planned. This ensures the team will have time to handle issues that require immediate assistance as they occur. For production issues that are not emergencies, the Product Owner will prioritize the defects back into the product backlog just like any other feature.

Establishing a triage process for production issues allows the team stay on top of bugs without disrupting progress. New and existing issues should be addressed in future sprints where proper planning and prioritization by the Product Owner can make sure the team is working on the most valuable features and tasks.

New Feature Requests

Bugs aren't the only things discovered in production. Customers often suggest new features or have suggestions for improving existing features. These types of issues are often reported via the same mechanism as defects. (For example, users often describe a feature request as a bug when the software doesn't work exactly how they would like.) For this reason, it's important for the Product Owner to have access to production issues.

DANGER! The Product Owner should always be planning ahead to ensure a continuous flow from sprint to sprint. The Product Owner should start looking forward to the next sprint before the end of the current sprint—planning new features, refining user stories, and updating backlog priorities, for example.

Daily Stand-up Meeting

The **daily stand-up meeting**, or **daily Scrum**, is a core part of Agile because it encourages communication and helps the team stay up to date on the sprint's progress.

The daily stand-up meeting earned its name because that's what it is—a short meeting where everyone remains standing to keep the meeting short and to the point. Daily stand-ups, which should be 15 minutes or less, are held five days a week at a time that works for all team members. The Product Owner, Scrum Master, and core development team all attend.

Stakeholders, managers, customers, and other interested parties are welcome to attend and listen, but this is not a status meeting for management. Only the team members, Scrum Master, and Product Owners speak during this meeting.

All team members share their brief answer to three questions:

1. What did you do yesterday?
2. What are you planning on doing today?
3. Is anything getting in your way?

DANGER! A common misperception is that the daily stand-up is for the Scrum Master. The daily stand-up is for the team and they should talk to each other instead of reporting to the Scrum Master. The primary role of the Scrum Master in the daily stand-up is to listen, provide guidance, and find ways to facilitate the team's progress.

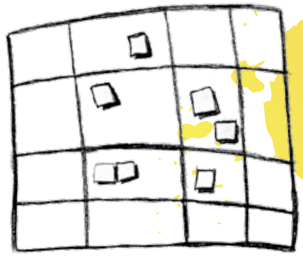
Keep It Short and Sweet

The daily stand-up is not the time for detailed design discussions. The need for a longer discussion might be identified during the meeting, but the actual discussion should take place outside of the meeting. That way, it can involve only the necessary participants and not tie up the entire team. The team members who need additional time to problem solve can schedule another time to get together.

DANGER! The team's daily stand-up might be in trouble if:

1. It lasts longer than 15 minutes
2. Participants sit down
3. Lengthy discussions ensue
4. You leave still not knowing if the sprint is on track

Making Progress Visible



As the team is reporting progress, it should be made easily visible and accessible to other project stakeholders. Transparency is highly regarded by Agile teams and, as such, communicating progress effectively is key. Burn down charts and task boards are two ways to communicate progress.

One of the responsibilities of the Scrum Master is to make sure that key progress indicators, are up to date and visible.

The **sprint Burn Down chart** communicates progress to stakeholders by showing the work remaining in the sprint. The daily stand-up, and not the burn down, is the best gauge of progress for teams.

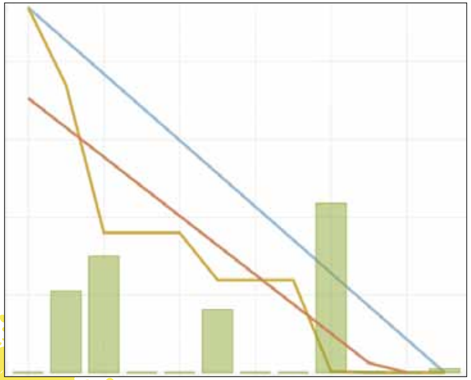
The **Task Board** shows who is handling a task and what state (Not Started, In Progress, or Done) the task is in.

These activities are good for keeping sprints on track, so let's take a closer look at each one.

Sprint Burn Down Chart

A sprint burn down chart is a graphical representation of work remaining in the sprint. The amount of work remaining in the sprint backlog is often shown on the vertical axis, with time along the horizontal axis. Burn down charts are useful for keeping track of the team's progress.

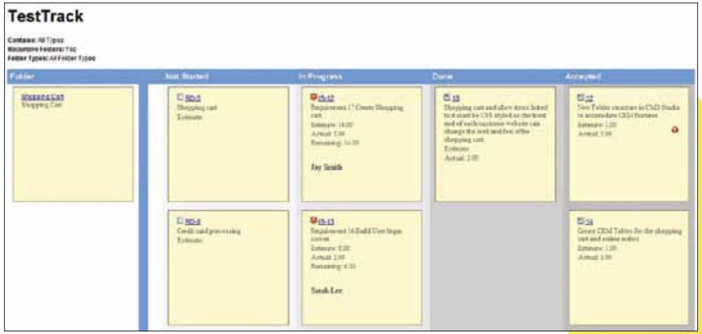
Ideally, the chart burns down to zero by the end of the sprint. If team members are reporting their remaining task hours honestly, the line should fluctuate up and down as it moves toward zero.



Task Board

The task board shows stakeholders who's working on what and keeps them updated on the status of all items. It is one of the most important information sources, or information radiators, that an Agile team has—perhaps the most important. The task board illustrates the progress an Agile team is making in achieving their sprint goals.

Many Agile experts recommend creating a physical task board located in an area where everyone on the team can see it often. Alternatively, you can use a software tool to automatically generate and update your task board. If you have access to a projector, you can display the task board on the wall. Besides saving space, this approach will also allow you to preserve the data in case you need to refer back to it after the sprint is complete.


TestTrack

Contains: All Types
Recursive Folders: Yes
Folder Types: All Folder Types

Folder	Not Started	In Progress	Done	Accepted
Shopping Cart Shopping Cart	SD-5 Shopping cart Estimate:	IS-12 Requirement 17 Create Shopping cart Estimate: 16.00 Actual: 2.00 Remaining: 14.00 Jay Smith	18 Shopping cart and allow items linked to it must be CSS styled so the front end of each customer website can change the look and feel of the shopping cart. Estimate: Actual: 2.00	12 New Tables structure in CMS Studio to accommodate CRM features Estimate: 1.00 Actual: 3.00
	SD-6 Credit card processing Estimate:	IS-13 Requirement 14 Build User login screen Estimate: 8.00 Actual: 2.00 Remaining: 6.00 Sarah Lee		16 Create CRM Tables for the shopping cart and online orders Estimate: 1.00 Actual: 1.00

When Things Aren't Going Well

Murphy's Law affects sprints just like it affects everything else in life. So when *whatever can go wrong does go wrong*, make sure you take the following steps:

Involve the Product Owner as soon as possible. The earlier you raise the red flag, the more time you'll have to get the problem resolved. No one likes surprises.

Involve the team. If there's a problem, the team should work together to figure out a solution. Maybe a developer helps out with testing because it doesn't do any good for him to continue programming if there's no bandwidth to test his new code.

Communicate. Make the sprint status available to everyone, especially stakeholders and management. Broadcasting issues usually helps to resolve them more quickly.

Agree to drop an item. While the team should complete everything they agreed to deliver in the sprint, if it becomes clear that an item won't be completed, work with the Product Owner to remove the least valuable item from the sprint.

Terminate the sprint. In rare instances, the sprint can be terminated by management or even the team. Sprint terminations should only occur if the sprint goal is unattainable or has changed so significantly that the resulting output of the sprint will yield little or no business value.

Checklist: Tasks by Role

Here's a handy checklist of tasks, divided by role.

Product Owner

- Participates in sprint planning, the daily stand-up meeting, the sprint review, and the sprint retrospective
- Works ahead to add features to the product backlog, and organize and prioritize the backlog
- Provides the team with product development direction and is available for discussions and questions
- Accepts or rejects features completed in the sprint

Scrum Master

- Participates in sprint planning, the daily stand-up meeting, the sprint review, and the sprint retrospective
- Monitors and facilitates team progress
- Acts as a servant leader and not a traditional manager
- Looks for impediments and deals with them
- Helps with team's understanding and adoption of Agile practices

Development Team Member

- Participates in sprint planning, the daily stand-up meeting, the sprint review, and the sprint retrospective
- Works on tasks associated with features in sprint
- Monitors issue queues
- Responsible for delivering the features they committed to in the sprint.

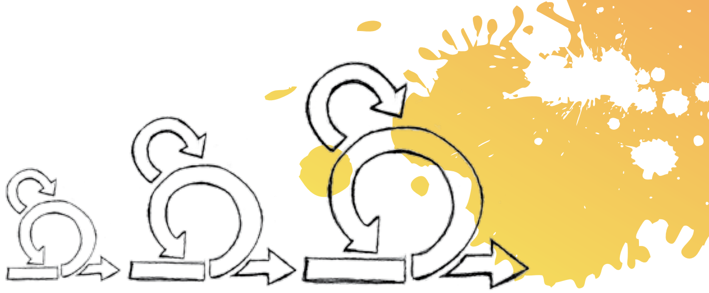
Daily Activities in a Nutshell

There are some basic Agile activities that are important to do every day. Here's what we learned:

- It's critical to honestly and openly communicate your status.
- Pair programming can improve quality and productivity.
- Among other things, the Scrum Master is responsible for moving impediments.
- The daily stand-up meeting allows the team to keep up to date on how the sprint is going and encourages communication.
- The burn down chart and task board are key progress indicators managed by the Scrum Master.
- Communicate problems and impediments as early as possible so the team has plenty of time to resolve them.

Automated Testing and Agile

Iterate with Confidence



During the last stop on this Agile Expedition, you got down to business with daily tasks and started developing the first iteration of your product. You're now probably ready to get into your next sprint. But what happens if you break what you built in the last sprint?

That's where **automated testing** comes in. Automated testing allows you to efficiently work incrementally with the confidence that each new sprint hasn't broken previous sprints.

Agile practitioners often include **Test-Driven Development (TDD)** as part of their toolkit. TDD, which focuses on writing tests for code before writing the code, can be an incredibly useful way to raise the quality of software.

Automated testing goes hand-in-hand with TDD. While TDD tests the code, automated testing typically makes sure the application functions properly as accessed through the user interface. In addition, you can also automate the testing of backend services, such as databases, to ensure an application is functioning correctly behind the scenes. For example, when a user account is added on a new user screen, automated tests can test and verify both that the screen functioned properly and that the database tables were updated with the information entered.

Automated testing usually focuses on customer acceptance testing. One of the goals of acceptance testing is to ensure the sum of the code parts are actually (at least) equal to the individual pieces. Unlike unit tests developed under a TDD model, which are tightly tied to the code itself, automated tests are one step removed and can be challenging to create, maintain, and extend in an Agile environment.

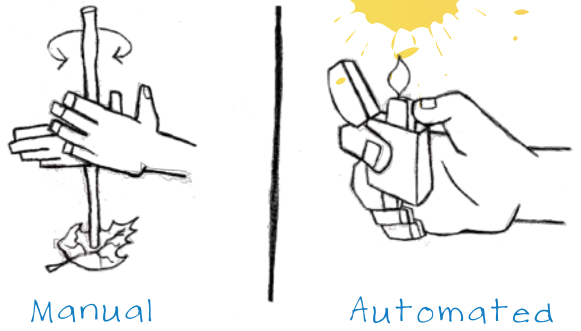
As your team transitions to Agile, it can seem like the functionality to test is constantly under development, and creating test scripts can feel like driving a car by looking only in the rear view mirror. The trick is creating the right kind of automated tests, in the right areas, to ensure that every sprint doesn't consist of one step forward and two steps back.

Is Automation Worth the Effort?

In the short term, automated testing can be challenging to implement in an Agile environment.

Some common challenges include:

- Manual testers don't have the skills to write scripts for test automation
- Software development environments are not equipped to handle automation
- Resistance from within the organization
- Financial costs associated with investment in tools, training, coaching, and hiring knowledgeable staff



So, with these challenges, why not just manually test each sprint? The answer lies in the long-term return on investment, or ROI.

The simple fact is automated test scripts run faster, don't get tired or bored, and don't suddenly miss test steps like people do. With a relatively small investment in tools and test scripts, your testers can focus on testing the new and complex parts of the application, while the automation tool keeps retesting the old stuff. Think of the ROI as 'time to run a test' times 'cost of a tester' times 'number of tests to run.'

For example, it takes 10 minutes on average to run a test, each tester costs $40 per hour, and you have 500 tests to run. Your manual testing cost is $3,333 per test cycle. In addition, it takes 83.3 hours to run all tests using one tester. That's over two man-weeks! An automated test tool can most likely run that same set of tests overnight across multiple computers, not only saving you money, but also saving you significant time. Plus, you can fully test after every nightly build, not just at the end of a sprint.

The Test Automation Manifesto

Several years ago, Gerard Meszaros, Shaun Smith, and Jennitta Andrea created the **Test Automation Manifesto**. (Download it here: http://xunitpatterns.com/~gerard/xpau2003-test-automation-manifesto-paper.pdf.) It's useful as a starting point when looking at automating tests in an Agile world.

The Test Automation Manifesto states that tests should have the following traits:

- **Concise:** Tests should be as simple as possible and no simpler.
- **Self-checking:** Tests report their own results.
- **Repeatable:** Tests can be run many times in a row without human intervention.
- **Robust:** Tests produce same result now and forever. They are not affected by changes in the external environment.
- **Sufficient:** Tests verify all the requirements of the software being tested.
- **Necessary:** Everything in each test contributes to the specification of desired behavior.
- **Clear:** Every statement is easy to understand.
- **Efficient:** Tests run in a reasonable amount of time.
- **Specific:** Each test failure points to a specific piece of broken functionality.
- **Independent:** Each test can be run by itself or in a suite with an arbitrary set of other tests in any order.
- **Maintainable:** Tests should be easy to understand, modify, and extend.
- **Traceable:** Tests should be traceable to and from the code they test, and to and from the requirements.

Several of these tenets can be quite challenging in the ever-changing Agile world. When examining what to automate (and what to apply these principles to), you might want to start with what not to automate. Edge cases and tests to explore functionality generally aren't good candidates. Focusing on core functionality maximizes your automated testing ROI.

When deciding what to automate, there are two parts of any development process to look at: continuous integration and regression testing.

Continuous integration means bringing your team's code together as often as possible, at least once per day, to ensure the software as a whole keeps working as changes are made. In its simplest form, continuous integration is used to make sure that all your code still compiles and links. When combined with automated testing, though, the value of continuous integration can dramatically increase.

Regression testing uncovers software errors by partially retesting a modified program to ensure that errors were not introduced in the process of fixing other problems. This area of testing often receives the least attention.

Continuous Integration

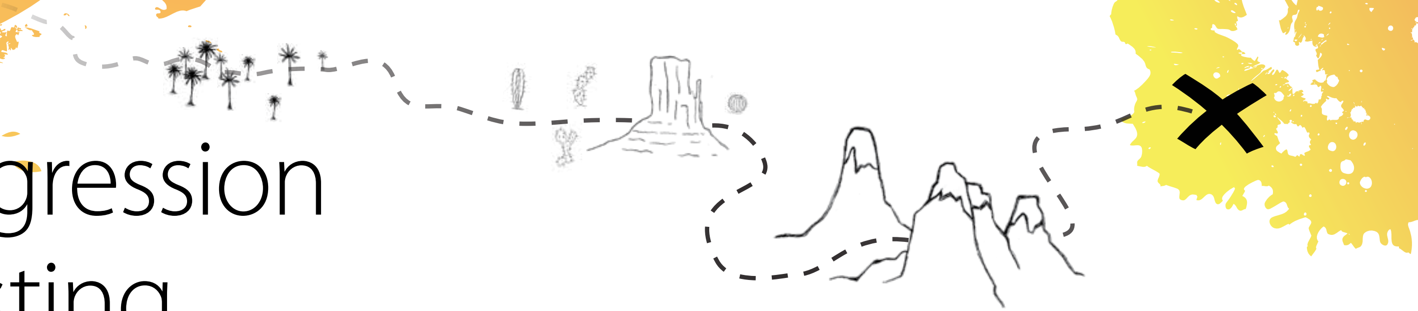
Most experienced Agile practitioners use continuous integration as part of their team plan. TDD can help here, and your compiler and linker will catch the basic "broken build" problems. However, including a good subset of your automated tests will help you find issues that have been unintentionally affected by the current sprint's updates.

When deciding which automated tests to include, your focus should be on broad automated tests, such as smoke tests, to ensure the base functionality of the application is intact. You want to look for tests that touch all the key application areas, so the testing you'll be doing in the sprint isn't delayed.

EXAMPLE: The application we're building requires users to log in before they can perform key actions. We include automated tests to make sure an administrative user can log in and view each main screen. However, we won't be adding the security functionality until a later sprint, so we don't include detailed security tests in the current sprint.

To learn more about automated smoke tests, check out this *Automated Smoke Testing* blog post: http://blogs.seapine.com/2009/06/automated-smoke-testing/.

Regression Testing



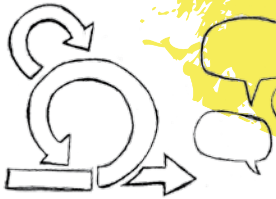
Regression testing is where traditional automated testing often comes into play. The challenge is how to integrate regression testing in a continually evolving environment.

Depending on the project, hardening sprints may need to be used as a way to compensate for a slow destabilizing of your application. In each sprint, you drift a bit further from your quality ideal as different parts of your system start diverging from each other.

Use hardening sprints to get everything back in line through refactoring. Fixing issues as they occur—rather than paying this technical debt toward the end of a release—is not only more efficient, but also builds in better quality.

The key is to focus your automated testing on stable or mature functionality. In those areas, going deep with your functional tests helps exercise the most code and minimize those late bugs in unchanged parts of the application.

Review at the End of Each Sprint



All good Agile practitioners understand the need for constant review and feedback, and it's no different with automated testing. At the end of each sprint, you should review three areas.

First, what new functionality should be automated and added to either your continuous integration or regression testing suite? If the new functionality is going to be extended in the next few sprints, add a set of shallow tests (probably in the continuous integration area). When you're ready, you can extend those scripts into a deeper set that can be used in your regression suite.

Second, which scripts should be rewritten or removed from your automation suite? You might want to rewrite or remove scripts that either test areas of functionality that are about to undergo large changes or consistently fail because of changes from the current sprint.

Finally, review areas that were not automated, but had significant defects associated with them. For example, you had no smoke or regression tests around failed passwords, but ad hoc testing discovered that area of the code was sensitive to change. Adding new automated tests to ensure that future changes don't break this sensitive area might be in order.

Plan for Automated Testing

This is one area where an Agile approach can pay real dividends for automation. Because the whole team is involved with sprint planning, team members who focus on automation have a chance to think about what needs to be automated.

In addition, while you're defining acceptance criteria, you should also be thinking about what you can (and cannot) verify in an automation test.

DANGER! If all your tests need to be rewritten after every sprint, you may need to examine your test automation approach. Are you automating the wrong areas of the application? Are your scripts using "fragile" methods, such as screen location, to find controls?

Engaging the Product Owners, developers, and other stakeholders on what can and will be automated also helps them understand how to define good acceptance plans. An acceptance plan of "screen should look clean and well-balanced" is difficult to verify, and even harder to automate.

Automated Testing in a Nutshell

Automated testing helps Agile developers iterate with confidence. Here's what we learned:

- TDD is a great way to improve software quality.
- Automation brings efficiency to testing and has a high long-term return on investment.
- Automated testing usually focuses on customer acceptance testing, but can also verify data behind the scenes.
- The Test Automation Manifesto is a good starting point for automated testing in an Agile world.
- Continuous integration ensures software keeps working as changes are made to the code.
- Regression testing helps you fix issues as they occur.
- Automated testing also requires constant review and feedback.

Are We There Yet?
Doneness Criteria

What's Done Is Done

We just explored automated testing's role in Agile development. Now, we'll be discussing a favorite word among developers: **done**.

So why devote an entire chapter to defining what done means? Isn't it obvious when something is done? Not if everyone's definition of done is different. That's why it's important for the team and other stakeholders to know where the goal line is.

Most traditional Waterfall projects end with an uncomfortable period of customer sign off, due to Waterfall's predictive planning approach. For example, on a year-long Waterfall project, done is determined a year in advance. This is one of the many problems with predictive planning methods.

With Agile, releasable features are developed in short sprints, so the team's definition of done is constantly reevaluated through the "inspect and adapt" process. There's less chance of disagreement at the end of a sprint because the definition of done is revisited at least every six weeks.

Who Says We're Done?

The definition of done is determined by the team during release planning—and that includes everyone who can be held accountable for delivery of the product at the end of the sprint.

Unless the team defines and agrees to specific doneness criteria, there will be conflicting opinions about what it means to be done. This is only natural, because members have unique perspectives related to their primary roles on the team.

For example, a Product Owner defines the acceptance criteria necessary for user story completion. Technical team members may define the more technical aspects of the development process that are or are not achievable during the sprint. External stakeholders who have influence may provide inputs and receive outputs from the sprint, but these are generally constraints on the team that prevent them from working toward a better definition of done.

“Done” Defined, Sort Of

Ideally, your team’s definition of done will be consistent throughout the project.

In organizations with a lot of waste, however, the definition may change over several sprints as the team becomes more efficient at producing software, and as the Scrum Master removes impediments.

With Agile, executable and production-ready code should be the minimum produced by the end of each sprint.

DANGER! There is no universally accepted definition of done for an Agile project. The uniqueness of projects, coupled with an organization’s culture, makes it nearly impossible to have a one-size-fits-all definition. This is why it is important for the team to establish doneness criteria.

What's the Definition?

You've convinced your team it's important to get together to define done. Now the question is, what is that definition?

If your team is new to Agile, working with each other, the product, or the organization's software development process, defining done will take some time. Schedule at least a couple of hours for the meeting.

When deciding what done means for a sprint, your team should understand the following:

1. What are the Product Owner's expectations?
2. What do other internal stakeholders require for getting to done?
3. What do external stakeholders require for getting to done?

You'll want to keep everyone involved, so try to make the meeting fun and interactive. Book a conference room and consider using sticky notes and a white board to map out the definition of done, like in the following example:

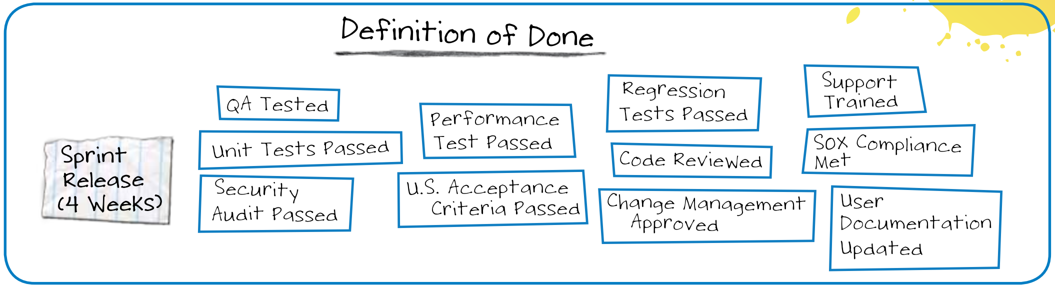


Figure 3: Release every sprint

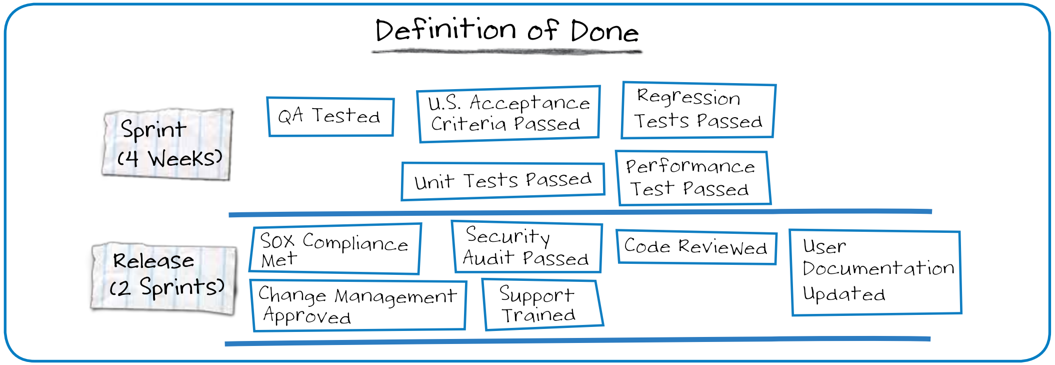


Figure 4: Sprint separate from release

Adjusting Sprint Length

If sprint length was determined prior to defining done, your team may need to adjust the length to get to their definition of done. If your team needs to extend the sprint length from two to four weeks to get to their definition of done, that's OK. If your team cannot get to done for the sprint or the release, you need to identify the blocks that are preventing them from getting to done.

DANGER! Defining done is not the same as user story decomposition, which is part of sprint planning. User story decomposition involves breaking down each story into its respective development tasks for each sprint. The definition of done may or may not change from sprint to sprint.

Communicating Done

After your team defines done, you need to make sure all stakeholders are aware of the definition. Put your definition of done on an information radiator, like a Scrum board or task board, and make sure it's visible by placing it in the hall, a team room, or somewhere else where it will be seen often by all stakeholders.

If the team is supporting their process with electronic tools, make sure everyone has access and knows how to view reports and get the latest status.

Remember, Agile relies heavily on trusting each other. Transparency about what done means is absolutely critical to building trust within your team and the organization.

DANGER! One benefit of Agile software development is that it identifies waste in an organization. The average sprint length is four weeks, with the range being plus or minus two weeks. This means sprints should be no longer than six weeks and should produce releasable code within that timeframe. If it takes more than six weeks to produce releasable code and get to done, you should inspect the development process for waste.

Eliminating Waste

Waste, which is defined as any activity that does not add value to the final product, should be eliminated. (This is different from removing slack. There should always be some slack in a system to make it perform optimally.)

A couple of methods for identifying and reducing waste include:

- **Value Stream Mapping**—a lean manufacturing technique used to analyze the flow of materials and information required to bring a product or service to a consumer. The goal of the technique is to make the company "lean," meaning free of wasted effort. For more information, visit http://en.wikipedia.org/wiki/Value_stream_mapping.
- **Innovation Games**®—helps companies improve business performance through collaborative and cooperative play. Learn how to eliminate waste by identifying how the product will actually be used. For more information, visit www.innovationgames.com.

After identifying a possible waste item, introduce a method, such as the "Five Whys," to understand the root cause. This method involves repeating the question "why?" up to five times to clarify the nature of the problem and reveal the solution. (You may have encountered the Five Whys if you have experience with Kaizen, lean manufacturing, or Six Sigma manufacturing methodologies.)

Suppose your white board looks like the board in Figure 5. In this example, "QA Tested" is impeded and prevents the team from getting to done in the four-week sprint. Remember, the team should produce production ready code at the end of each sprint. The team could extend the sprint to five or six weeks, but that won't solve the underlying problem.

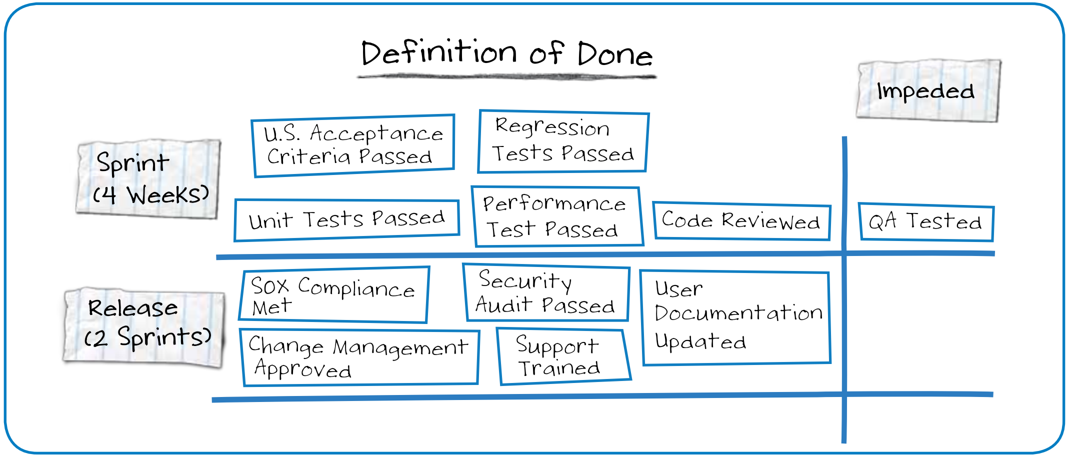


Figure 5: Production code cannot be produced in the sprint due to QA testing impediment

Using the Five Whys, the Scrum Master asks why as many as five times, to help understand what is causing the QA impediment:

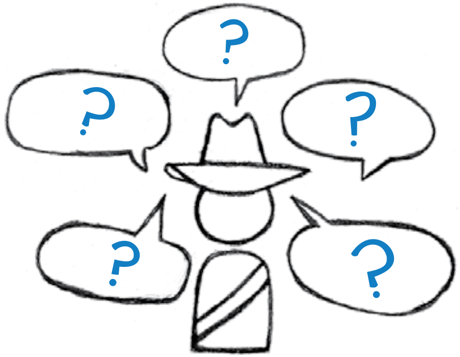
1. **Why can't QA test within one sprint?** We have to request the QA environment, which requires three weeks lead time.

2. **Why do we have to request the environment?** Because the QA environment is shared among all applications.

3. **Why is the QA environment shared?** Because there aren't enough QA team members to configure and prepare the environment.

4. **Why can't the dev team prepare the environment?** Because policy states that only QA team members can deploy code to the QA environment.

Ah ha! Now you know there is an organizational impediment to QA testing the code within the sprint. The Scrum Master, who is responsible for removing impediments, needs to work with the appropriate managers to better understand this policy and, hopefully, have it modified or removed.



Doneness in a Nutshell

Defining done helps Agile team members and other stakeholders understand where the finish line is for sprints and releases. Here's what we learned:

- The team's definition of done is constantly reevaluated during Agile projects.
- There is less chance for disagreement at the end of a sprint because the definition of done is revisited at least every six weeks.
- The team determines the definition of done during release planning.
- If sprint length was determined prior to defining done, the team may need to adjust it.
- Once done is defined, it should be communicated to all stakeholders.
- Eliminating waste can help you achieve a better definition of done.
- The Five Whys can help you understand the root cause of an impediment.

It's Showtime: The Sprint Review

What Comes After Done?

We trekked to Doneness Criteria on the last leg of our journey, but we certainly aren't "done" with this Agile Expedition. During this stop, we'll examine the **sprint review**.

The sprint review provides another point of inspection to make sure the team is delivering the right product. It also promotes communication among stakeholders. It's important to note that, for the Agile Expedition, we're defining stakeholder based on the Project Management Institute (PMI) definition, which is any person or organization that is positively or negatively affected by the outcome of a project.

At a minimum, stakeholders include the Scrum Master, Product Owner, and development team because they are responsible for the delivery of the sprint objectives and the project in general. Other stakeholders who may attend the sprint review include technical teams that provided support to the team during the sprint, such as infrastructure and architecture. The project sponsor, senior managers, and executives are also invited to attend, in addition to any external organizations that will be affected by the changes resulting from the project.

DANGER! Do not decide who should attend on the day of the sprint review. The Scrum Master and Product Owner plan who to invite to sprint reviews before the project starts, updating the list as the team iterates. Depending on the visibility of the project, some stakeholders may only interact with the project team during the sprint review. Even if no one attends the sprint review except the team, continue scheduling it for every sprint so it will always be an option for stakeholders.

Find Your Marching Cadence

One reason it's important to have a consistent sprint length is to help the team develop a natural rhythm or cadence. If the team is sprinting every four weeks, then stakeholders will come to expect that another sprint review will occur four weeks after the previous one.

As with the daily Scrum or stand-up, the sprint review should occur at the same time and in the same place each time. This provides consistency for the stakeholders, especially for senior managers and executives who may spend their days going from meeting to meeting.

Who Wants to Drive?

Prior to the sprint review, decide who will "drive" the meeting. The opportunity to demonstrate the product shouldn't fall on the same team member for every sprint. Sharing this responsibility gives everyone on the team a sense of ownership of the team's effort. It also allows team members to show off the team's work and receive recognition for it. Remember, the entire team contributed to the project, so don't let the Scrum Master or Product Owner get all the credit by doing every demo.

Another benefit to having a different team member drive each sprint review is that it helps reinforce the "servant leader" role of the Scrum Master. It can be difficult for teams used to working together under traditional roles to transition to the Agile leadership style. Sharing the demo responsibilities can help the team make the change from a top-down project hierarchy, where only the project manager or senior team members get face time with senior management.

DANGER! Remember that a Scrum Master acts as a "servant leader" and should not assign the sprint review responsibility to anyone. Let the team decide who will drive the sprint review. Make the decision fun by having the team draw straws or pull a name out of a hat.

Get on the Same Page

User stories accepted into a sprint are commitments the development team made to the Product Owner to complete the stories by the time the sprint finishes. Teams new to Agile may struggle with meeting these commitments for reasons varying from developers overestimating the work they could complete, the Product Owner not providing information in a timely manner, organization impediments getting in the way, and so on. It's important for the team to work out the message they will send to the stakeholders in the sprint review about any unfinished stories.

The team, including the Product Owner, should also be on the same page prior to the sprint review. There shouldn't be any surprises as far as the team is concerned, which means the Product Owner should have already accepted or rejected the user stories based on the acceptance criteria outlined for the stories. If there are incomplete user stories, the Product Owner will already be aware of this going into the sprint review.

DANGER! If some user stories were not completed in the sprint, the team should be upfront about what they know went wrong. Remember that Agile relies heavily on trust and the team must be transparent to gain that trust. If the team doesn't know what went wrong, let sprint review attendees know the team will discuss the sprint outcome in the retrospective. Make sure the retrospective results, including associated actions for future improvement, are shared with all the stakeholders.

The Sprint Review Steps

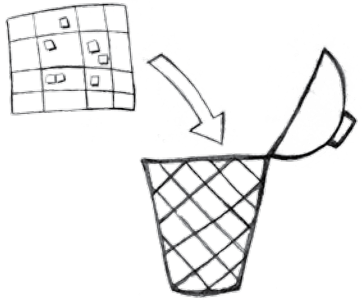
Everyone is on the same page, you've decided who will drive the meeting, and you've got all the stakeholders together. The sprint review is ready to begin.

Three things occur during the sprint review:

1. The Product Owner gives a brief overview of the objectives of the sprint or release.
2. A team member gives a brief overview of a selected story and demos the story in the software.
3. The team answers any questions about the implementation of the story and takes note of anything that will be valuable for the retrospective.

Steps two and three are repeated for each story in the sprint.

Be sure to block out adequate time to conduct the sprint review. As a general rule of thumb, plan for a meeting duration of four hours for a 30-day sprint. In practice, sprint review meeting length should be adjusted for the needs of the project, but it's best to keep the meeting length consistent for each sprint.



Completed Stories

Teams that use a physical task board to track stories usually rip up and throw away the cards for completed stories at the end of the sprint. However, this doesn't allow for any kind of archiving. Suppose you want to look back at how you handled a similar story in a previous project, or your organization was going through an IT audit. What would you do?

You could file the story cards for future reference, or take pictures of the board at the end of the sprint. An easier solution is to use a software tool to track the status of user stories and automatically generate your task board, burn down charts, and other reports. Using this kind of software tool ensures a story's history is stored in a database for future reference. If information about a user story is needed in the future, it can be recalled with just a few clicks.

This kind of electronic recordkeeping is especially beneficial for teams in heavily regulated industries. With a configurable and flexible software tool helping you track information, you will have the documentation you need to survive audits and provide accountability without having the tool get in the way of your Agile practice.

What Happens with Incomplete Stories?

Stories still in the sprint backlog at the end of the sprint are carried into release planning for the next sprint. After discussing the stories with the team, the Product Owner may decide not to carry a story into the next sprint, which is perfectly acceptable.

However, your team should understand these three things when deciding whether to continue developing an unfinished story:

1. The business value the story adds.
2. The technical ramifications of deciding to stop
3. The additional support needed to continue

DANGER! If your team uses story points to estimate user stories, the team does NOT get points for incomplete stories. Regardless of fault, the team gets points only for delivering completed stories. As opposed to traditional methods that value assigning work to the team, in Agile the team has the power to choose the work they will take on in a sprint. As a result, the team builds trust by delivering what they commit to.

Should You Re-Estimate?

Should you re-estimate an unfinished story when planning the next sprint? While there are different schools of thought, re-estimating stories that were started (but not completed) in a previous sprint may waste the team's time and cause confusion for stakeholders. If the story was not started in the previous sprint, it can easily be integrated back into the product backlog and re-sized against other stories in the backlog.

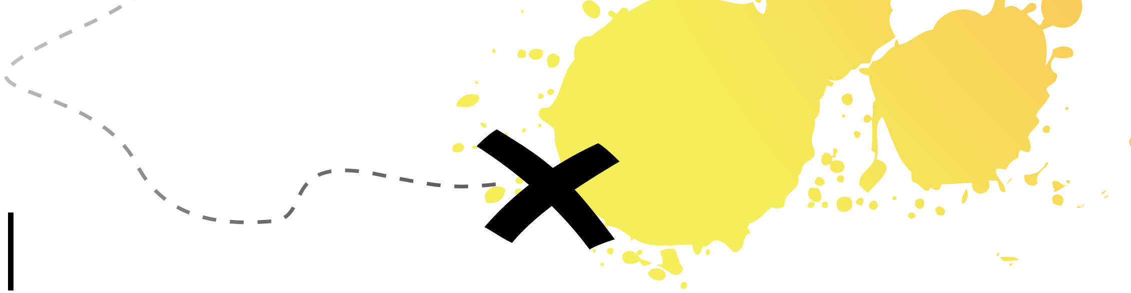
Sprint Review in a Nutshell

The sprint review provides another point of inspection to make sure the team is delivering the right product. Here's what we learned:

- A stakeholder is any person or organization that is positively or negatively affected by the outcome of a project.
- Sprint length should remain consistent so your team can develop its cadence.
- Sprint reviews should occur at the same time and in the same location to prevent confusion.
- A different team member should drive each sprint review.
- Prior to the sprint review, the team should know what will be covered during the meeting so there are no surprises.
- The team should agree about the message they are going to send to the stakeholders about any unfinished user stories.
- The length of the sprint review should be adjusted to suit the project.
- Software tools can automatically archive completed user stories and other data about the sprint for future reference.
- Incomplete stories can be taken into the release planning for the next sprint.

Look Back in Agile:
The Sprint Retrospective

The End of the Trail



In the words of a popular GPS navigation system, “You have reached your destination.” You’re at the end of the trail, standing on the big X on the map. The sprint is complete, the product has been demonstrated in the sprint review, and everyone can relax, right?

Not exactly.

Even though the sprint review takes place on the last day of each sprint, the sprint is not closed until the team has held their sprint retrospective. (While we could call it the “lessons learned” or “postmortem,” the term most often associated with Agile in general, and Scrum in particular, is “retrospective.”)

The sprint retrospective is usually held after the sprint review, allowing the team to reflect on the entire sprint, including the review. However, sometimes scheduling conflicts make it necessary to have the retrospective before the sprint review.

DANGER! Regardless of which way your team decides to order the retrospective and sprint review, they should occur on the same day to keep the team’s cadence. Completing both on the same day allows the team to mentally close the sprint and focus on what’s coming next.

Why a Sprint Retrospective?

Traditionally, most projects have included some sort of retrospective at the end of a project. While well intentioned, reviewing lessons learned at the end of a project can have several disadvantages:

1. **It's too late to change anything**. The project has been completed, for better or worse. All you can do is capture the lessons learned, write the wrap-up report, and file it away. If you're lucky, future project teams might benefit from your findings, but it's too late to help your project.

2. **There are large memory gaps**. Team members have a hard time remembering what they did last week, let alone what they did over the past year or more. Waiting until the end of a project often results in large gaps between what the team remembers and what actually happened.

3. **Energy levels are low**. Everyone knows the disadvantages already described, and this drains team members of their enthusiasm for a retrospective conducted at the end of the project.

By contrast, retrospectives conducted at the end of every sprint give the team an opportunity to reconnect and improve the way the project is delivered.

The Last Inspection

For Agile teams, the retrospective is the last in a long line of inspections that occur during a sprint. Unlike the other inspections, the retrospective provides the best opportunity for the team to identify ways to adapt and improve their Agile processes going forward. Following are the main advantages of the retrospective:

1. **There's still time to make a difference**. What you discover in the sprint retrospective will be used in future sprints, making a positive impact on the project team, the customer, and product quality.
2. **You have almost total recall**. Team members only need to remember things that happened over a period of weeks instead of months or years, which means the gap between what they remember and what actually happened will be small.
3. **Energy levels are high**. Teams are engaged and energized because they know the actions coming out of the retrospective will be implemented. In other words, they know the time is well spent and not wasted on a fruitless exercise.

Who’s Involved?

At a minimum, the retrospective should include the core team (the Product Owner, development team, and Scrum Master) and the supporting teams—people who were involved in fulfilling sprint objectives, but not assigned to the entire project.

Optional participants include other stakeholders, such as managers.

DANGER! Many teams new to Agile only include the delivery team in the retrospective. Project stakeholders should be given the option to attend because they are often consumers of the information that comes out of the sprint, such as metrics and reports. For example, if there is a problem with the way the burn down chart is structured, stakeholders will want to use the retrospective to discuss the issue and identify the actions necessary to change the chart.

How Long Should the Retrospective Last?

As a rule of thumb, a retrospective for a 30-day sprint should last about four hours. However, the retrospective should be tailored to the needs of the project and the experience of the team. For teams new to Agile, the sprint retrospective will initially take longer but will gradually take less time as they get used to the activity. Alternatively, as they start to realize the benefits of the retrospective, the time may actually increase because the team will be more engaged in subsequent retrospectives.

DANGER! The more participants there are in the sprint retrospective, the more time the team will need to gather and process the information generated during the meeting. This can cause problems if there are time constraints—an inefficient retrospective that does not produce actionable results will yield little value. You don't want your team to feel like the sprint retrospective was a waste of time. If time is an issue, consider splitting up-in to groups and then coming back together to share ideas.

Facilitating the Retrospective

If your team is new to Agile, have the Scrum Master facilitate the retrospective because they typically have the most retrospective experience. As the team becomes familiar with the methods used to conduct a sprint retrospective, other team members should have the opportunity to facilitate the retrospective. This encourages buy-in from the team, which results in a more productive retrospective.

Conducting the Retrospective

In *Agile Retrospectives: Making Good Teams Great*, Esther Derby and Diana Larsen identify several steps the facilitator should follow when conducting a retrospective. The basic steps are:

1. Set the stage
2. Gather data
3. Generate insights
4. Decide what to do
5. Close the retrospective

Retrospectives should be fun and informative, and they don't always need to take a lot of time to put together. Often, just having candy in the room to keep the blood sugar up, a white board, markers, some sticky notes, and the ability to facilitate a good conversation is enough to get the job done and make it worthwhile for everyone.

For example, take a look at Figure 6: Retrospective results.

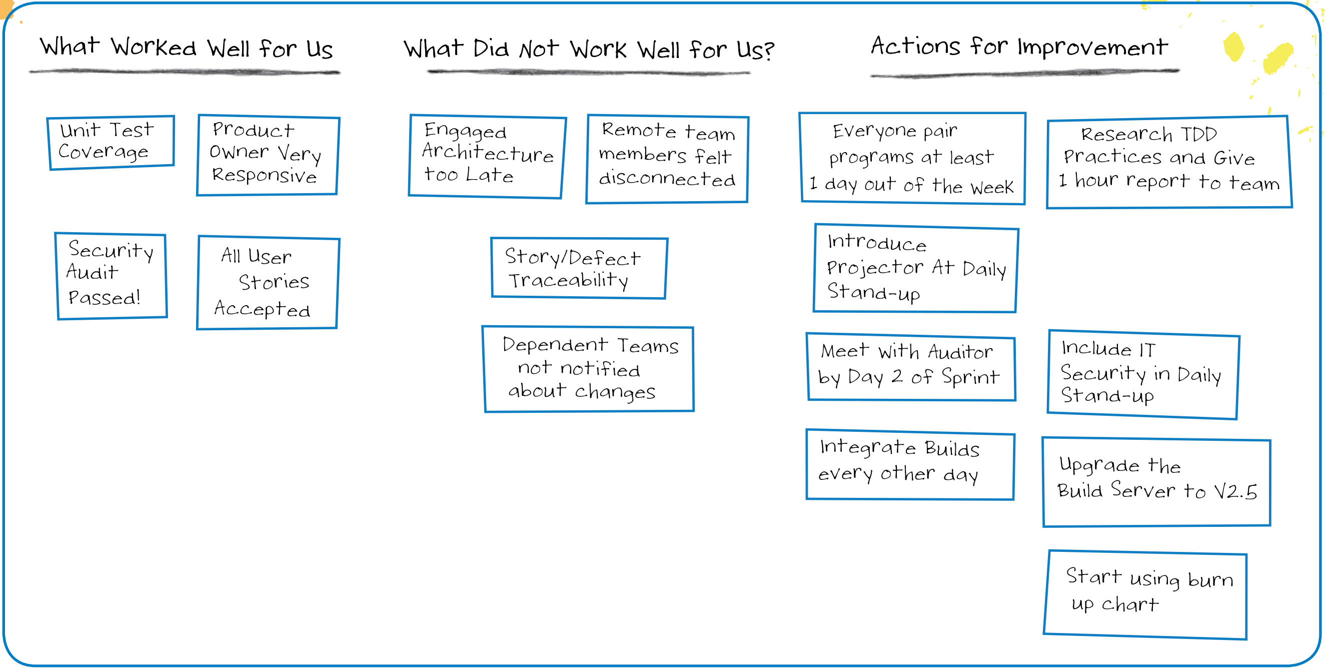

What Worked Well for Us
Unit Test Coverage
Product Owner Very Responsive
Security Audit Passed!
All User Stories Accepted
What Did Not Work Well for Us?
Engaged Architecture too Late
Remote team members felt disconnected
Story/Defect Traceability
Dependent Teams not notified about changes
Actions for Improvement
Everyone pair programs at least 1 day out of the Week
Research TDD Practices and Give 1 hour report to team
Introduce Projector At Daily Stand-up
Meet With Auditor by Day 2 of Sprint
Include IT Security in Daily Stand-up
Integrate Builds every other day
Upgrade the Build Server to V2.5
Start using burn up chart

Figure 6: Retrospective results

While sticky notes are used in Figure 6, a white board could easily be used to track the following:

1. What worked well in the sprint? List the processes, interactions, and events that the team found helpful and would like to continue.

2. What didn't work well in the sprint? List the delays, impediments, and broken processes that the team would like to either improve or discontinue.

3. What actions can we take to improve our process going forward? List the actions that volunteers from the team (including the Scrum Master and Product Owner) agree to see through to completion in future sprints. In rare cases, actions may be picked up by the team as a whole.

DANGER! It's not enough to only capture what worked well and what didn't. Actions for improvement must be captured too. It's the only way to make sure improvements are rolled into future sprints and releases. Remember Agile teams don't just inspect—they also adapt!

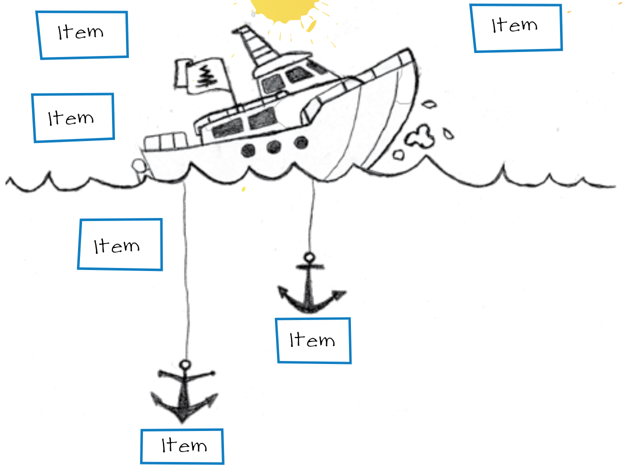
Plan to Play

Games can also be used to foster communication and help identify issues during a retrospective, but they usually require advanced planning. One such game is Speed Boat from Innovation Games. (To learn more about playing Speed Boat, visit http://innovationgames.com/speed-boat.)

In this game, the speed boat represents the project and the items represent activities in the project. The placement of the sticky notes also has meaning:

1. Above the waterline-these sticky notes represent wind, which are items that are propelling the project forward. In other words, they are the wind in the sails. (We realize that speed boats don't have sails, but go with it.) The farther ahead of the sail, the stronger the item.

2. Below the waterline-these sticky notes represent anchors, which are the impediments and other items holding the project back. The deeper the anchor, the heavier the item.



Notice that actions have not been identified. While items representing wind might lead to future actions, anchors are generally where the team should focus their time. Lead the team in a discussion about the specific actions that can be generated from these anchors.

Note: In the variation of Speed Boat played in the video, the participants concerned themselves only with the anchors to focus on the impediments that need to be fixed. Because retrospectives should also capture the things that went well, however, it is better to use the version in our example.

DANGER! Depending on your corporate culture, the word "game" should be used cautiously. While these games produce a useful outcome for the organization, the word "game" can derail a well-intentioned and valid activity. Make sure senior management and stakeholders understand the benefits of the game beforehand, or use a different word to describe the activity.

Carry It Forward

The team decides which improvements and actions they will carry forward into future sprints based on the lessons learned in the retrospective. It's good to open subsequent retrospectives with a review of how the actions from the previous retrospective played out.

If actions require more than an hour of time to complete, the team member who volunteered for the action should carry it with them into sprint planning (which should be the next day) and then add it to the task board in the next sprint.

DANGER! Don't be discouraged if actions that were carried forward from a previous retrospective aren't successful. The actions are often experiments to see if a certain process will work better than an existing process, and all experiments are not a success. A team may also identify an item as something that was done well in one sprint, yet mark the same item as something that was done poorly in a subsequent sprint. It's OK when this happens, but if the same item seesaws regularly, dig a little deeper to figure out the cause of this fluctuation.

Retrospective Challenges

Again, retrospectives should be fun. Teams new to Agile will often struggle in the beginning when it comes to participating in retrospectives. The reasons for this struggle are varied, but a few of the possible culprits include the following:

- **Fear of blaming**. If a story wasn't completed in the sprint or something happened in a team member's area that delayed the rest of the team, there may be a fear that the retrospective will result in blaming. To combat this, make sure you focus the team's attention on the group goals and on continuous improvement as a team. Work with team members who might cause problems before the retrospective to understand their issues.
- **Fear of wasting time**. If a team member thinks the retrospective is a waste of time, try to find out why. Maybe they didn't benefit from the actions (or lack of actions) coming out of the previous retrospective, or the team member might be overloaded with other projects and is frustrated at the thought of spending time looking backward.

- **Fear of speaking up**. Team members who are unsure of what to say or are shy will refrain from speaking. Handle this by introducing a speaking token that gets passed from team member to team member, but make it OK for team members to skip their turn if they have nothing to say. If a team member is unusually quiet over several sprints, speak with them individually to find out why.

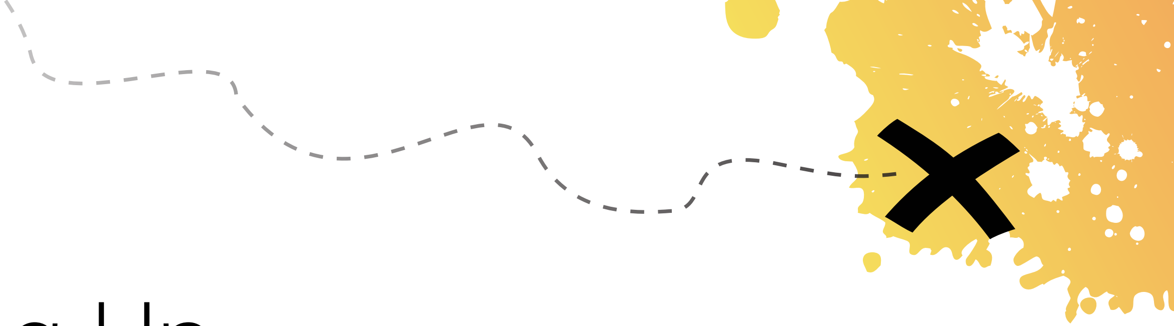
DANGER! Don't put quiet members on the spot in the retrospective. Singling out quieter team members may only intensify their unwillingness to talk, and may come off as an attempt to lay blame. Instead, talk to the person outside of the retrospective, one on one.

The Sprint Retrospective in a Nutshell

The sprint retrospective is held after the sprint review, allowing the team to reflect on the entire sprint, including the review. Here's what we learned:

- Even though the sprint review occurs on the last day, the sprint is not closed until the team has held their retrospective.
- Retrospectives conducted at the end of every sprint provide an opportunity for the team to reconnect and improve the way the project is delivered.
- The retrospective is the last in a long line of inspections that occur during a sprint, and it provides the best opportunity to identify ways to adapt the Agile process going forward.
- At a minimum, the Product Owner, development team, and Scrum Master should participate in the retrospective. Supporting teams and other stakeholders may also be included.
- Retrospectives are usually four hours long for a 30-day sprint, but should be tailored to the needs of the project and team.
- The Scrum Master typically facilitates the retrospective in the beginning, but other team members are welcome to take on the role of facilitator.
- Games can help make the retrospective meeting be more productive, interactive, and fun.

Measuring Up: Progress Metrics



Measuring Up

We've covered the basics of what it takes to become more Agile. But, we haven't talked about metrics yet.

Metrics are important for all projects, no matter how they are delivered. In addition to helping you measure and communicate team progress, which is especially important in Agile development, metrics also provide insight into areas for improvement.

On this stop of the Agile Expedition, we'll lay the foundation for a good set of metrics for measuring progress on an Agile project.

DANGER! The metrics needed over the life of a project may change in response to the needs of the project. Do not start executing the project until the team understands the initial metrics required by stakeholders.

Be Prepared

Before your Agile team starts executing their first sprint, it is important to understand what metrics will be collected, why they are being collected, and who will be reading the resulting reports.

Understanding the data and your stakeholders' needs will help you provide clarity when reporting your team's status, especially to senior management. Also, depending on the project, the person responsible for generating the reports (the project manager, Scrum Master, etc.) may be required to provide different views for different stakeholders. A software tool can help by automatically pulling this information together into the various views.

DANGER! There is no one-size-fits-all approach to metrics, and the use of metrics will vary from project to project. Be aware that the metrics we cover do not make up a comprehensive list.

Hard Metrics

Projects involve two types of metrics: hard and soft. **Hard metrics** address the mechanics of Agile projects and are reported in the burn down chart, the burn up chart, the defect report, and build failures.

The Burn Down Chart

Arguably, the most useful report for tracking progress on an Agile project is the **burn down chart**. Burn down charts track work (points, ideal days, hours, etc.) remaining against either sprints in the project or days in a sprint. So, burn downs can be tracked at the project and sprint level.

At the project level, the X-axis represents the number of sprints and the Y-axis represents the work remaining (which can be represented as story points, ideal days, or whatever the team chooses).

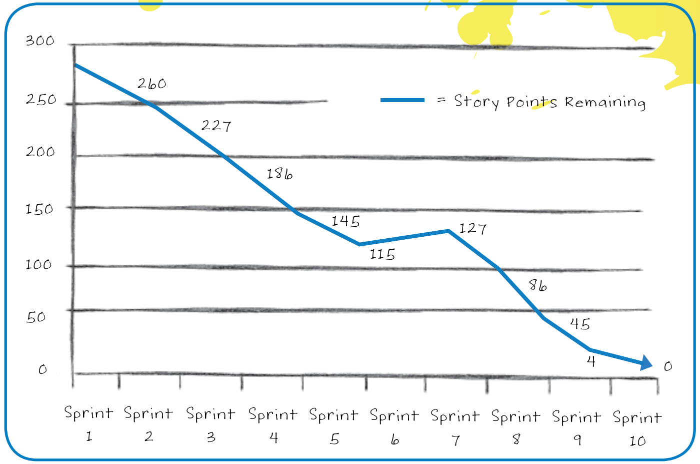


Figure 7: Project-level burn down chart using story points

At the sprint level, the X-axis represents the number of days in the sprint and the Y-axis represents the remaining effort (in hours) in the sprint.

At the project level (see Figure 7), burn down charts are good for showing work remaining in the project, determining team velocity, and estimating how many sprints it will take to complete the project. Because average velocity can only be measured after a few sprints, it is important not to set unrealistic expectations for the team at the beginning of the project. After the team has established an average velocity, it can be used to estimate how many sprints it will take to finish the project's current backlog of work.

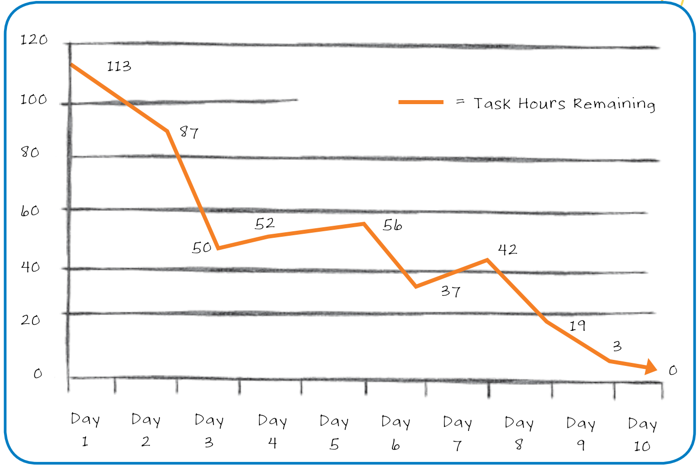


Figure 8: Sprint-level burn down chart using hours

Example: In Figure 7, the team's velocity is around 41 story points per sprint. Assuming all else is equal and the number of points does not change, the team will finish all 260 story points in about seven sprints.

At the sprint level (see Figure 8), burn down charts are good for showing work remaining in the sprint. Work is normally shown as hours remaining. Sprint burn downs are a good way to keep a pulse on the sprint progress at a daily level.

DANGER! As illustrated in Figures 7 and 8, hours remaining will not trend smoothly down to zero. This is the nature of burn downs. As work is added and removed, the trend line will roller-coaster to the bottom. The team's feedback during the daily Scrum should be the primary indicator of progress. Always trust what the team says.

The weakness of the burn down chart is that it doesn't always show the amount of work completed in the sprint or the change in the amount of work in the project. Someone looking only at a burn down chart may not see how much work was completed or how much work was added or removed to the project. Without this understanding, stakeholders may get confused and unnecessarily alarmed, as illustrated in the example below.

Example: In Figure 8, when comparing Sprints 4, 5, and 6, it appears the team did only about 30 points worth of work in Sprint 4 (measured between Sprints 4 and 5). In Sprint 5, it appears that no work was done, but work was added (measured between Sprints 5 and 6). This roller coaster trend continues through the project's remaining sprints. If the project stakeholders saw only this burn down chart and no other reports, they might worry that the project was slowing down without understanding the reasons why.

DANGER! If stakeholders want to know when the team will complete the work in the project backlog before team velocity is known, the person who is reporting the status must set the appropriate expectations by helping management understand the importance of empirically deriving future projections. Stress the importance of projecting the number of story points the team can deliver in future sprints by looking at what it has delivered in past sprints. If you must, work with the team to provide a best case/worst case range, but do not give an exact date or number of sprints.

The Burn Up Chart

Burn up charts are typically shown at the project level using story points. In a burn up chart, the X-axis represents the sprint while the Y-axis represents the story points completed and the total story points in the project.

Burn up charts are good for showing the total amount of work (usually in story points) in the project, the number of story points completed in each sprint, and for determining team velocity, which can be used to estimate the number of sprints it will take to finish the project's current backlog of work.

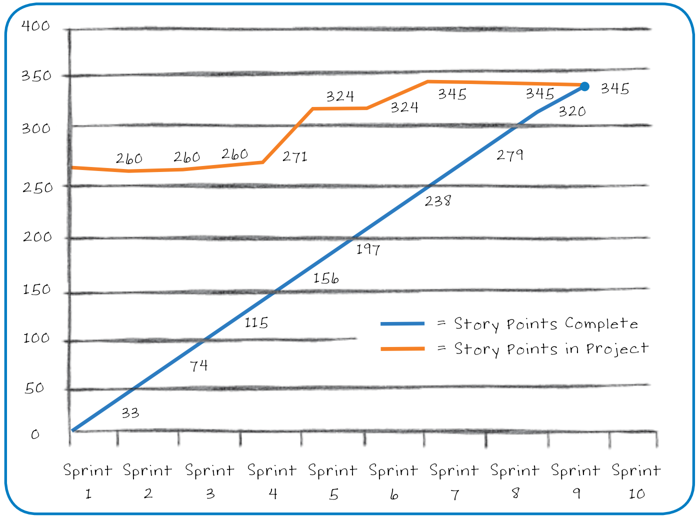


Figure 9: A burn up chart with story points completed and total story points in project

Example: In Figure 9, the 'Story Points in Project' line represents the total story points in the project. Any time user stories are added or removed from the project, they are shown in the burn up chart. The 'Story Points Complete' line shows the completed user stories. The project finishes where the two lines intersect.

The weakness of the burn up chart is that the number of story points remaining is not immediately evident. To get story points remaining, you must subtract the most recent number in the 'Story Points Complete' line from the most recent number in the 'Story Points in Project' line.

Example: In Figure 9, if the team subtracted 33 from 260 in Sprint 1, they would get 227 story points remaining in the project.

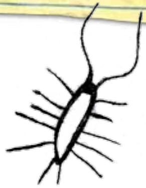
DANGER! While story points remaining in the project can be calculated using a burn up, consider your audience when reporting this information. Managers will generally want the numbers presented to them in a more simplified format. It's often better to provide both the burn up and burn down charts to give them the full project picture. Consider aggregating the data from both charts into a single format that is easily scannable.

The Defect Report

So far, we have talked about measuring project and sprint progress, which is good for figuring out how fast you're delivering. But burn up and burn down charts do not indicate product quality. One of the benefits of adopting Agile practices is building better quality into products. As a result, stakeholders may want to know how many defects were discovered in the current sprint and how the number and type of defects compares with previous sprints. Software testing tools can assist with generating defect reports on the fly.

If your organization is transitioning to Agile methods from Waterfall, and you need to show proof of the benefits, it would be useful to compare the total defect rate for a product over several projects.

Example: Were fewer defects found in this year's Agile project than last year's Waterfall project? Breaking it down further can yield defect comparisons between categories (e.g,. Critical, Average, Minor) and across different types of testing (e.g., Unit, Integration, User Acceptance).



Build Failures

Using build failures as a metric can help teams understand how many builds they are breaking within a sprint and across sprints. Build failures are a reflection of the code quality going into the build and can help the team make decisions regarding their engineering practices. Also, when done in combination with continuous integration, understanding build failures can help the team identify who is submitting buggy code to the repository.

Soft Metrics

You now know more about the hard metrics that address the mechanics of Agile projects. What about soft metrics, that track things like **customer satisfaction** and **team morale?** Too often, soft metrics are not reported to management. This is unfortunate because half of the Agile Manifesto emphasizes the importance of people. Specifically, it emphasizes the values of “Individuals and interactions over comprehensive documentation” and “Customer collaboration over contract negotiation.”

Customer Satisfaction

Customer satisfaction simply measures how satisfied the customer is with the product. This is not unique to Agile, but it is highly emphasized on Agile projects. Customer satisfaction is emphasized so highly, in fact, that the product owner (who represents the customer) is accessible to the development team daily, throughout the entire project. While the product owner is an important reflection of the overall happiness of the customer, teams should also reach out to end users to assess their happiness with the product. The Sprint Review is an excellent time to measure customer satisfaction.

Team Morale

While customer satisfaction is important, equally important is the satisfaction of the team. This includes the Product Owner, Scrum Master, and development team. An Agile team will not perform optimally if they are not happy, and management should be made aware of morale issues so they can step in and help if needed.

There are many ways to uncover team morale and a few are outlined in *Agile Retrospectives: Making Good Teams Great* by Esther Derby and Diana Larson. One way to determine morale is through the use of a morale thermometer, which is a chart that shows how the team is feeling at a moment in time. The retrospective is a good time to take the team's morale temperature.

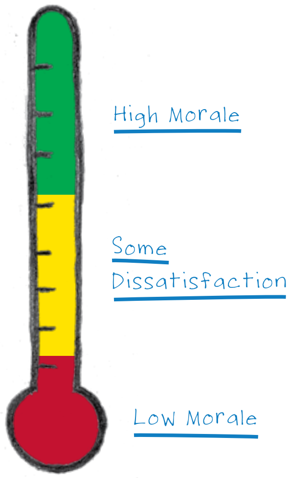
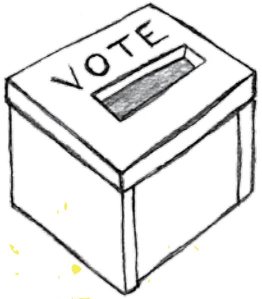


Figure 10: Morale thermometer: red means morale is low, yellow means there's some dissatisfaction, and green means morale is high

In Figure 10, the team's morale is shown in varying levels. The diagram is counterintuitive because the higher the temperature, the higher the morale. In other words, you want a high team temperature. The numbers along the thermometer indicate the number of team members who voted for each color.

If a sprint resulted in extremely low morale, the team will want to spend some time in the retrospective exploring what caused so much dissatisfaction.

DANGER! If team members aren't comfortable sharing their feelings, make voting anonymous to ensure an accurate temperature reading. While it's always good to share responsibilities in an Agile project, this is one time where it might make sense for one person (possibly even someone from outside the team) to take the temperature.



Metrics that Don’t Matter

What conversation about metrics would be complete without touching on the metrics that don’t matter?

As organizations transition from Waterfall to Agile methods or try to become more Agile, many of the old metrics should be left behind.

Percent Complete

Percent complete is the percentage of overall work for a feature or task that has been done. For example, if a task is estimated to take eight hours and has four hours remaining, then the task is 50 percent complete. Fairly obvious, right?

In less creative industries, percent complete might accurately reflect the amount of effort needed to complete a task, but this is not the case in the software industry. In fact, the last 20 percent is often the most difficult part or requires the most effort.

For Agile projects, only two percentages matter: zero and 100 percent. A feature or task is either done or not. So, if someone says they are 80 percent done with a task, an Agile team member will translate that into zero percent done. Reporting the percent complete puts undo overhead on the team and takes focus away from what really matters, which is getting to “done”.

Tracking Actual Feature or Task Hours

DANGER! Management will often want to see percent complete because "that's the way we've always done it." In this case, your best bet is to educate management on the benefits of not tracking percent complete. Learn how to get away from percent complete: http://blogs.seapine.com/2010/11/how-to-get-away-from-percent-complete/.

Most organizations have a time tracking tool that team members use when they are working on multiple projects. For Agile projects, tracking actual hours at the project level makes sense when team members are on multiple projects that are under multiple cost centers. This goes for both internal and external projects.

On the other hand, for internal projects, tracking actual hours at the feature and task level is a wasted activity. Many project and people managers wrongly believe that forcing a team to track actual hours at the feature or task level will make the team better estimators. With software development, however, the level of accuracy that can be attained with estimating is limited at best because two features are rarely exactly alike.

Agile team members also work on multiple tasks (including email, meetings, etc.) throughout the day. When actual hours are recorded, they are really more like estimated actual hours. As with percent complete, tracking actual hours at the feature and task level puts undo overhead on the team.

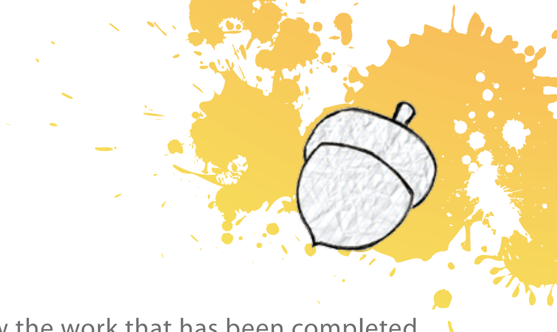
If your organization is developing an application for an external customer or an external vendor is developing an application for your organization, details around tracking actual hours will come down to the specifics of the contract. If the payment is received on a feature-by-feature basis, then it will be necessary to track hours at the feature level. This should be the exception and not the rule.

Metrics Checklist

Here's a quick checklist of commonly used metrics and the ways to obtain them.

Metric	Burn Down Chart	Burn Up Chart	Defect Report	Build Report	Sprint Review	Product Owner Feedback	Morale Thermometer
Work remaining	✓	✓ (calculated)					
Velocity	✓	✓					
Work completed		✓					
Total Work		✓					
Defects			✓				
Broken Builds				✓			
Customer Satisfaction					✓	✓	
Team Morale							✓

Metrics in a Nutshell



Metrics are important to all projects, Agile or not. Here's what we learned:

- Before starting an Agile project, know what metrics you need to collect and who will be reviewing them.
- Burn down charts track information at the project level and at the sprint level.
- Burn downs charts are good for determining team velocity and projecting how many sprints it will take to complete the project.
- Burn up charts are typically shown at the project level using story points.
- Burn up charts show the work that has been completed and changes in the workload.
- The defect report shows the number of defects found in the entire project.
- The defect report is useful for revealing whether the number of defects is increasing or decreasing compared with previous sprints, releases, or projects.
- Customer satisfaction and team morale are also important metrics to track for Agile projects.
- Tracking percent complete and actual feature/task hours are not useful metrics.

Mixing Methodologies

Don’t “Do” Agile

If you have ever talked shop with a fellow software developer about methods, you might have heard something like, “We don’t do Agile.” Well, of course not! Agile is a set of values, not a process or method. Organizations do not do Agile—they either are Agile, or they aren’t.

Every organization takes a different approach to being Agile and, as a result, multiple methods have emerged under the Agile umbrella. However, they all stay true to the Agile Manifesto.

The method we have explored the most during the Agile Expedition is Scrum. On this stop, you’ll learn about some of the other Agile methodologies and engineering practices that support Agile processes, to help you gain a better understanding of which methodologies and practices provide the most benefit for your organization and projects.

Agile Delivery Methodologies

Agile delivery methodologies are defined processes for delivering software in an agile manner. They all value close collaboration with the customer and incrementally delivering quality software based on prioritized business value and in the shortest timeframe possible.

DANGER! This is not an exhaustive discussion of Agile delivery methodologies. It should go without saying that you'll need to do more research to discover the best fit for your situation.

Scrum

Scrum, which was formalized by Ken Schwaber and Jeff Sutherland in the mid 1990s, is one of the more popular Agile delivery methods. During this Agile Expedition, we have referred to many Scrum terms, including sprint, Product Owner, and Scrum Master. One of Scrum's strengths is that it is a well-defined and extensively documented delivery methodology. Because of that, we won't define Scrum in detail here. Instead, check out the recommended resources to learn more.

Recommended Resources

- *Agile Project Management with Scrum* by Ken Schwaber
- *Agile Software Development with Scrum* by Ken Schwaber and Mike Beedle
- Scrum.org (www.scrum.org)
- Scrum Alliance (www.scrumalliance.org)

Extreme Programming

Extreme Programming (XP) was created by Kent Beck and is similar to Scrum in that it emphasizes short, iterative, and incremental development cycles, short feedback loops, close customer collaboration, and work prioritized by highest business value.

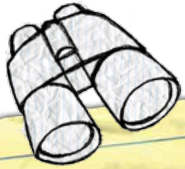
XP puts an extra emphasis on engineering practices that help improve code or product quality, such as:

- Test-Driven Development
- Pair Programming
- Continuous Integration
- Refactoring

These engineering practices also provide a perfect complement to Scrum's more widely known and adopted delivery methods.

XP operates on the following values:

1. Simplicity
2. Communication
3. Feedback
4. Respect
5. Courage



Recommended Resources

- *Extreme Programming Explained: Embrace Change* by Kent Beck and Cynthia Andres
- *Planning Extreme Programming* by Kent Beck and Martin Fowler

Feature-Driven Development

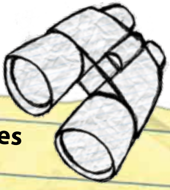
Feature-Driven Development (FDD) was created by Jeff De Luca in the late 1990s and, like XP and Scrum, is focused on delivering customer value by identifying and delivering the features with the highest business value first, in an iterative and incremental fashion. FDD is a model-driven Agile process that puts emphasis on first identifying the problem domain using Unified Modeling Language (UML®), then digging into feature development on an iterative and incremental basis. Because FDD is model driven, it has additional roles such as Class Owner to support the modeling function.

FDD emphasizes the following activities as part of its process:

1. Develop an overall model
2. Build a features list
3. Plan by feature
4. Design by feature
5. Build by feature

"At Bayt.com, we had opted to use FDD. While I believe that SCRUM, as all other agile methodologies is excellent in supporting human-oriented software development environment, I continue to believe that it lacks well defined control points (milestones) that are required to track the progress of features implementation. It is clearly focused more on the Project Management side rather than the Software Development side"

- Ala' Abuhijleh, Development Manager at Bayt.com



Recommended Resources

- *Java Modeling in Color with UML* by Peter Coad, Erick Lefebvre, and Jeff De Luca
- *A Practical Guide to Feature Driven Development* by Stephen Palmer and Mac Felsing
- Feature Driven Development (www.featuredrivendevelopment.com/)

Dynamic Systems Development Method

Dynamic Systems Development Method (DSDM) was created and is formally maintained by the DSDM Consortium and, like other Agile processes, focuses on close customer collaboration and delivering the features with the highest business value first, in an iterative and incremental fashion.

Like FDD, DSDM has more roles than Scrum or XP. However, DSDM is unique in that it identifies a pre-project and post-project phase. Scrum, XP, and FDD all assume project funding has been approved and do not provide guidance for pre- or post-project work. Project managers will appreciate this extra guidance because all projects must go through the initial funding process, as well as fulfill their service level agreements (SLAs) in the post-project period.

The three phases of DSDM are:

1. Pre-project
2. The project lifecycle
3. Post-project

The project lifecycle phase includes the following stages:

1. Feasibility
2. Functional model iteration
3. Design and build iteration
4. Implementation

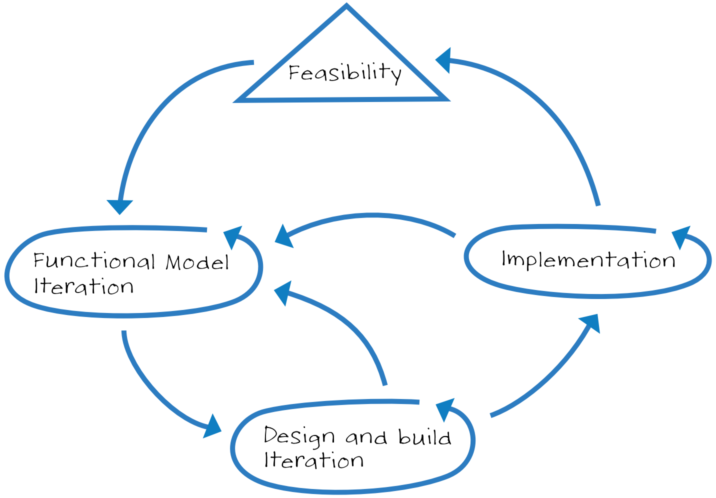


Figure 13: DSDM Project Lifecycle Stages

Recommended Resources

- *DSDM: Business Focused Development* by Jennifer Stapleton
- DSDM Consortium (www.dsdm.org)

Kanban

Kanban, which roughly means "signboard" in Japanese, is a manufacturing technique created by Taichi Ohno and popularized by Toyota more than 50 years ago. Kanban was one of the earliest forms of lean manufacturing and, in the past 10 years or so, "lean" thinking has made its way into software development in the form of Agile methods.

Kanban uses what is known as "pull" (or demand-generated supply) to meet demand rather than a "push" method, which relies on demand forecasting. Tolerances are set for work in process (WIP), so there is never more WIP in the queue than the team can handle and what is needed to meet demand. Kanban systems are useful for organizations where short timeboxes associated with other Agile methodologies are not required.

To learn more, check out this *Kanban by Example* blog post: http://blogs.seapine.com/2011/01/kanban-by-example/.

Recommended Resources

- *Implementing Lean Software Development: From Concept to Cash* by Mary and Tom Poppendieck
- *Kanban* by David Anderson
- *Lean Software Development: An Agile Toolkit* by Mary and Tom Poppendieck
- *Scrumban* by Corey Ladas
- *The Principles of Product Development Flow: Second Generation Lean Product Development* by Donald G. Reinertsen

Engineering Practices

Engineering practices involve activities that complement Agile delivery methods with best practices generally associated with Agile projects.

All of these practices should reduce the business and technical risk associated with software development, which includes the cost of development and maintenance. These practices can be used individually or all at the same time.

DANGER! It is important to implement at least one of these practices to lessen the likelihood of what Ward Cunningham, one of the pioneers of XP, terms **technical debt**, which is a decrease in code maintainability and an increase in development cost over time due to the shipment of "not-quite-right code."

Test-Driven Development

Test-Driven Development (TDD) is the practice of writing a unit test before writing any code. This can be done relatively quickly, with the developer writing the test, then writing the code, and then running the test in small increments. TDD ensures the code is consistently refactored for a better design.

TDD has the following distinct benefits:

- It contributes to better overall system design by reducing code duplication and other anomalies.

- It forces programmers to think about end results first, which increases the likelihood that the code will meet customer needs.

- It increases test coverage for the system under development, which uncovers more defects and reduces what Kent Beck, author of *Test Driven Development by Example*, calls defect density.

In his book, Beck identifies the following TDD mantra:

1. Red (Fail) – First, write a little test that doesn't work, and perhaps doesn't even compile.
2. Green (Pass) – Make the test work quickly, committing whatever sins necessary in the process.
3. Refactor – Eliminate all of the duplication created in merely getting the test to work.

Pair Programming

Pair Programming is the practice of pairing two developers to work on one module of code. The typical setup is two developers at one workstation. One developer writes the code, while the other watches and thinks further how to break down the problem and reengineer the solution. When one developer reaches a block, the keyboard is passed to the other. This passing back and forth happens as often as is necessary.

Pair programming has the following distinct benefits:

- It reduces the chance of defects in code written for tough software engineering problems.
- It allows one developer to think more abstractly about the solution while the other writes the code, leading to better design.
- It shares the knowledge of the code between two developers, decreasing the number of modular experts and increasing the number of generalists, which reduces bottlenecking among team members.
- It increases the chances that team members will develop common and improved programming methods.

DANGER! Pair programming is not an all-or-nothing proposition. Your team may chose to pair only one day a week, or only on the problems they think are the most difficult. Teams should have the flexibility to pair when they think it's necessary to build the best product for the customer and create the most maintainable codebase.

Continuous Integration

Continuous Integration is the practice of integrating code daily, usually with automation, and running automated tests against the code. Developers check their tested unit code into the code repository each day. Based on a differing number of methods (e.g., polling, specific times, etc.), the build server integrates all the code and runs automated new and regression tests against the build. The development team is notified of build failures and successes through various methods, such as email and RSS feed.

Continuous integration has the following distinct benefits:

- It allows the team to track down problem code faster, in the event the build breaks. As a result, code quality is improved.
- It provides feedback to the development team almost immediately, depending on when the build is run.
- If desired, it allows successful builds to be deployed to a production environment immediately.

Refactoring

Refactoring is the practice of taking small chunks of code (i.e., at the unit level) and frequently rebuilding it so that it is as simple as possible, yet still delivers the expected value. Martin Fowler popularized this practice in his book *Refactoring: Improving the Design of Existing Code*.

Refactoring has the following distinct benefits:

- It maintains the flexibility of the architecture, allowing it to evolve in a structured manner over the entire life of the product, which usually translates into lower maintenance costs.
- It helps software engineers understand their code better, which leads to better design.
- It helps reduce unnecessary and duplicate code.

That’s All, Folks!

We’ve come to the end of Seapine’s Agile Expedition. On this journey through the heart of Agile, we’ve taken you from building a backlog to conducting a sprint retrospective and beyond.

Although we’ve covered the basics, we have by no means fully explored the territory. Our goal was to give you a better understanding of Agile development, a few of the common pitfalls to avoid, and how it can benefit your organization and your customers.

When you’re ready for a more in-depth exploration, Seapine offers a range of services and solutions that can help make your business more agile. Our Agile Services team provides coaching, assessment, training, and delivery solutions. To learn more, please visit www.seapine.com/agileservices.html.

About the Authors

Jeff Amfahr, Director of Product Management, is responsible for establishing the strategic direction of Seapine products. Jeff has more than 20 years of experience designing and delivering software products for a variety of organizations, from small start-ups to large multinational companies.

Alan Bustamante, Senior Agile Consultant, leads Seapine Software's Agile Services practice. Alan has been in the software business for over 10 years, four of which have been working with Agile teams. He is passionate about building better software through the use of Agile methods, and is very active in the Agile community. Alan holds multiple industry certifications including Project Management Professional (PMP), Certified Scrum Professional (CSP), and IBM Rational Unified Process (RUP) Solution Designer.

Paula Rome, Senior TestTrack Product Manager, focuses on Seapine Software's TestTrack product family. For over 20 years, Paula has been creating quality-critical software systems for a wide range of industries spanning from healthcare to satellites. She has worked with large and small teams and a variety of methodologies and cultures, which has given her an appreciation for practical approaches to developing software products that make customers happy.