reference manual

Delivering Agile Projects with Scrum

www.esi-intl.com
Module Overview

Overview of Scrum

This module describes the history of the agile project management movement, and the experiences and theories that led to the *Manifesto for Agile Software Development*; defines Scrum; and provides an overview of Scrum roles, ceremonies, and artifacts.

It also focuses on the Scrum framework and agile values and principles necessary to practice Scrum effectively.

Traditional Project Management and the Traditional Life Cycle

Traditional Project Management and the Traditional Life Cycle

When the term “traditional methods” is used, it refers to some prevailing practices that often have been presented as the “gold standard” in project management:

- Detailed, up-front specification of features, requirements, and user expectations
- Complete and comprehensive planning of the entire product and project before any development work is performed
- User or customer validation of the entire specification, with authorization to proceed and with a key goal of discouraging changes or modifications once accepted
- Creation of a complete “big-bang” project plan, cost estimate, and schedule for the development of the entire product in one iteration
- Development of the total system with little customer interaction during the development cycle. In software projects, this often means that elements are coded separately (modular design) and then integrated into a whole before presentation to the customer. This causes development teams to work in silos and generate a lot of waste when eventually interacting with one another.
- Presentation of the complete, finished product to the client for acceptance without any incremental exposure to the user before the product or prototype is completely built
Acceptance or rejection by the user of the entire finished product with virtually no involvement in building the product

This strict representation of the traditional style of project management is a bit of a fallacy, as iterative and incremental methods have long been a part of many projects, and changes—though tightly controlled and sometimes discouraged—often are an accepted part of even the sternest project methods. Nonetheless, this description of traditional project methods is accurate for the purpose of pointing out the differences between these traditional methods and the agile approaches reviewed here.

Many software developers, product designers, and project managers (PMs) have been trained in these methods, and have been taught that only this strict and structured approach can protect projects from undisciplined “hacking,” constant scope change, and the lack of project control. Organizations like the Project Management Institute (PMI)\(^1\) and its book, *A Guide to the Project Management Body of Knowledge (PMBOK\(^\circledast\) Guide)*, and the Software Engineering Institute (SEI) with its Capability Maturity Model\(^\circledast\) (CMM\(^\circledast\)) were early advocates of these disciplined processes, and they did much to inject a predictable, engineering-style approach to previously undisciplined and ad hoc project management methods.

An outgrowth of these methods was the traditional product development life cycle, often called the “waterfall” method. As depicted in the figure below, the waterfall method describes a sequential development process in which the tasks of development take place in a one-way manner and often incorporate phase gates or stage gates, which enforce the idea that one can’t move to the next sequential step until the current step is completed and validated. Additionally, once a step has been completed, it is very complicated to get back to it for additional changes. This leads to difficulties implementing any customer feedback or improvement ideas that the customer or project team might have.

\(^1\) PMI and PMBOK are registered marks of the Project Management Institute, Inc.
The Waterfall Method

This method was sponsored strongly by organizations such as the National Aeronautics and Space Administration (NASA) and the Department of Defense (DOD), which required contractors and employees to follow strictly defined waterfall methods in order to be eligible to contract with these entities (the DOD has since changed its predisposition and is now an advocate of iterative methods in certain projects). To be in compliance, this method also required the development of a set of lengthy, detailed documents, such as the following:

- Project concept document
- Feasibility study
- Requirements document
- Conceptual design
- Architectural design
- Detailed design (Work Breakdown Structure [WBS])
- Test plan and acceptance criteria
- Staffing plan
- Stakeholder engagement plan
- Communication plan
- Risk plan
- Procurement plan
These traditional concepts were developed in reaction to early experiences at product development. For example, in the software industry, programmers and developers would sometimes immediately begin to write code, rather than eliciting and documenting the requirements and expectations of users. These ad hoc methods often resulted in undiscovered defects or flaws in concept, which were costly and difficult to repair when discovered late in the project. Early software development efforts were also plagued by lack of documentation, leading to maintenance nightmares when original developers were no longer available.

These traditional methods often are referred to in the agile community as “predictive,” connoting the idea that the requirements, expectations, duration, activities, and outcomes of development projects can be predicted accurately before any actual development activity takes place. As previously noted, this idea of predictability has been challenged by both academic and practical experience. Listed below are some of the theories behind traditional models:

- The ability to predict all requirements up front
- The ability to estimate the efforts and resources required
- The ability to control or discourage change
- The value of extensive documentation
- Customers always knowing what they want
- The project team always knowing how to build it

### Traditional and Agile Project Constraints

In traditional project management, the project team is tasked with building detailed plans up front, based on a clear idea of scope. Based on that scope, a set of other plans are developed: schedule, budget, quality plan, communication plan, staffing plan, risk management plan, procurement plan, stakeholder planning, and so on.

Also in traditional project management, we use what we call a “triple constraint” consisting of scope, schedule, and budget management. The principle of the triple constraint is simple: You cannot change one of the components of the triple constraint triangle without altering at least one other. The team receives the scope and then calculates the schedule and the budget required to deliver that scope. With all these plans ready, the project team is responsible for implementing a master plan. Essentially, project execution in traditional project management means implementation of the plan. In addition, well-executed projects result in as little variations to the original plan as possible.

In agile projects, we start by admitting that the scope is not fully known. This less-defined scope is a positive because it allows the team to be part of creating the scope. This gives them the opportunity to add value beyond just executing project tasks. Unlike traditional project management, the budget or time are often the set constraints that guide the eventual scope of the project. This is referred to as the “inverted triple constraint.”
The Inverted Triple Constraint

The inverted triple constraint is particularly important in agile methods because it sets the stage for the agile approach. We begin without knowing exactly what we are building; however we trust in a skilled team that is able to deliver what customers want in a very efficient way. The team achieves this through a cycle of iterations, or sprints, where they continuously deliver results, ask for feedback, and allow some time to reflect on how to become more efficient in the next sprint.

The Agile Movement

Top Challenges and Drivers for Change

The climate of the business world and current project environments offer quite a few challenges that make it more and more difficult to use traditional methods to run projects. Among these challenges, some of the most important are:

- Changing business priorities
- “Siloed” work environments, which make handoffs very inefficient
- An increasing emphasis on time to market
- Missing or incomplete requirements
- Multitasking and a lack of focus from teams
- Quality and rework issues
- Unrealistic estimates and deadlines set by the wrong people
- A lack of empowerment, low morale, and reduced engagement
- Overproduction or overprocessing and an abundance of wasteful features
History of the Agile Movement

Agile philosophies are the culmination of years of research, theory, and discovery. They originally presented fresh ideas that have been consequential in the software community and have been adopted by the project management community to offer new, adaptive approaches to project management.

Agility is—

- The ability to move quickly and gracefully (an agile dancer)
- A quick, resourceful, and adaptable character (an agile mind)

The movement toward agility was actually the culmination of an incremental realization, among both practitioners and theorists, that the then-current methods of software development and project management had serious flaws. These limitations were impeding the development of innovative software applications and causing important projects to take longer, cost more, and to be delivered without the key features that users required.

The well-known Standish Group CHAOS Studies, familiar to PMs worldwide, illustrate that a majority of IT projects fail to comply with schedule and cost projections, and often fail to deliver the benefits predicted when they are approved. Additional research by entities such as the DOD confirms these findings. Of $35.7 billion spent by the DOD in 1995 for software, only 2 percent of the software was usable as delivered. The vast majority of the software, 75 percent, was either never used or was cancelled prior to delivery.

Other academic research challenged common project management methods in use, especially in an IT context. Experienced PMs are beginning to challenge many of the fundamental ideas of project management when applied to complex IT projects. Experience on these projects has shown the following assumptions are flawed:

- It is actually possible to plan a large project that guarantees success following a detailed plan.
- Changes can be prevented from occurring late in a large project.
- It makes sense to finalize decisions in a large project early in the process.

PMs began to promote the idea of uncertainty in IT projects with ideas such as the following:

- Requirements for new software systems will not be completely known until users have had an opportunity to use them.
- Continuous improvement within the same project is more important than lessons learned at the end of the project.
- Uncertainty is inevitable in the software development process.

Over time, there have been various IT systems development methods:

- **Waterfall model**: A sequential process; maintains a document trail
- **Rapid-prototyping model**: A disposable prototype; helps establish customer preference
- **Spiral model**: A series of prototypes; identifies major risks
- **Incremental or staged delivery model**: A system that is delivered to the customer in chunks
The **Evolutionary delivery model**: An iterative approach in which customers test an actual version of the software

These academic findings and observations from real-world efforts led many developers and PMs to question the assumptions of traditional methodologies, which include the following: Requirements can be specified completely in advance; requirements will not change during development; users can know what they want before they see it; and software development and project management are predictable, repeatable processes.

Recognizing problems with existing methods does not, however, solve the predicament of repairing, replacing, or augmenting those processes. Respected PMs began to recommend a set of practices that could begin to replace the traditional methods. These simple precepts have been cited as a central foundation of the movement toward agile approaches:

- The early release of evolving design and code
- The daily build of code and a fast turnaround on changes
- Deeply skilled teams

The DOD used iterative and incremental development in the 1950s, and Evo, an iterative project management approach, was developed by T. Gilb in the 1960s.

Developments in industry, especially the lean manufacturing systems pioneered by Japanese firms like Toyota, validated many of the ideas brewing in the software and project management communities. Toyota Production System (TPS) was the first system that introduced ideas like flow (later known as iteration), empowered and self-organizing employees, continuous improvement, single-piece or small-quantity production versus batch and queue. These concepts became later the foundation of agile.

The Standish Group finding that around 60 percent of features built into IT projects are rarely or never used was also noteworthy. Concepts like focusing on features the customer valued and specifically requested, and building in quality up front rather than “testing it in” later, resonated with these communities.

The development of agile methods accelerated in the 1990s, as Scrum was developed at Easel Corporation and XP evolved at Chrysler. In the mid-1990s, the Dynamic Systems Development Method (DSDM) was introduced and quickly adopted in Europe.

**The Agile Manifesto**

The *Manifesto for Agile Software Development* (also referred to as the *Agile Manifesto*) was the culmination of these new theories and approaches. Written by a group of advocates of iterative and incremental development methods, this simple statement is the foundation document of the agile movement. In combination with a set of 12 agile principles, it sets forth the underlying philosophical concepts of agile project management. The *Agile Manifesto* was written and released in 2001 and immediately generated controversy and debate. Designed to address many problems identified with software development methodologies in common use, the *Agile Manifesto* outlined a new approach; focused on innovative, exploratory projects; and was dedicated to concepts such as collaboration, prototyping, cross-functional teams, and responsiveness to change.
The signatories included Kent Beck, the proponent of XP, Alistair Cockburn, the developer of Crystal Methods and author of influential works on agile development, and Jim Highsmith, who has translated agile software concepts into an agile project management methodology.

<table>
<thead>
<tr>
<th>Manifesto for Agile Product Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>“We are uncovering better ways of developing products by doing it and helping others to do it. Through this work, we have come to value—</td>
</tr>
<tr>
<td>• &quot;Individuals and interactions over processes and tools</td>
</tr>
<tr>
<td>• &quot;Working products over comprehensive documentation</td>
</tr>
<tr>
<td>• &quot;Customer collaboration over contract negotiation</td>
</tr>
<tr>
<td>• &quot;Responding to change over following a plan</td>
</tr>
<tr>
<td>&quot;That is, while there is value in the items on the right, we value the items on the left more.”</td>
</tr>
</tbody>
</table>

Below is a list of these statements and what they mean:

- **Individuals and interactions over processes and tools**: Process and tools don’t add value for the user; only skilled people can do that. Processes and tools, when selected and applied judiciously, can offer guidelines and best practices, and can improve productivity, but agile practitioners understand that it’s the human interactions, within the team and with the users and customers of the desired product, that grant us the insight to deliver the right results. Many agile advocates look at the business process reengineering (BPR) movement of the late 1990s as the culmination of a trend toward valuing processes over people, as if the perfect process alone could enforce success, and the skills and characteristics of the people were secondary. Agile theory asserts that even the most perfect process can do no more than guide and support skilled, motivated, and driven individuals focused on delivering value. By shifting power to people and allowing them to make improvement suggestions, we achieve an end result far better than simply following a process.

- **Working products over comprehensive documentation**: This is, as stated, one of the most frequently misinterpreted principles within the Agile Manifesto. The key idea it intends to convey is that a working product or prototype can grant users insight into the actual status of development efforts far more clearly and realistically than any status report or Gantt chart. This statement is based on the agile advocates’ experiences of product development, in which every software or product under development is always proclaimed by the developer to be “90 percent done” until it either is delivered or fails and is discarded. This principle insists that the only way to know the real status of an effort, and to gauge the users’ satisfaction and acceptance of the results, is to deliver something tangible that users can review and assess.

This preference of working software or product prototypes over documentation also serves to emphasize a key idea in agile development: Documentation is no substitute for collaboration and interaction. Status reports are distributed through e-mail; Gantt charts are posted on a wall in the project office; requirements documents are approved via an e-mail chain rather than face-to-face. These are all examples of the sort of interaction-free exchanges that agile methods are designed expressly to replace with vivid, intensive

1 www.agilemanifesto.org
collaboration. By showing tangible results in a collaborative environment, we can also help customers offer timely, valuable feedback that leads to successful products.

- **Customer collaboration over contract negotiation:** Projects based on contracts traditionally have implied that the entire scope and range of features required are documented before any development begins, and that the scope, schedule, resources, and often the price are fixed and unchanging throughout the life of the project. Sometimes, these contracts and scope documents are not even created by the teams building the product. To make matters worse, in many projects the contract becomes a substitute for collaboration with the client, and complex or innovative projects often become adversarial wrangles over terms of the contract. Agile methods aim to move focus from the contract and predefined planning documents to collaboration with the customer and other stakeholders, becoming a central element of the development effort. A key insight of this statement in the *Agile Manifesto* is the idea that it’s only customers who can define the value of the features the development teams deliver, and that they must take ownership of the output and be involved throughout the entire effort, not just at the beginning when scope documents are developed and at the end when acceptance is required. Ultimately, it is hard to capture all the value delivered into a contract; hence, the reduced focus on contracts over collaboration.

It’s important to note that this idea can create difficulties on both sides of the development effort. Project teams migrating from traditional to agile methods often are startled by the degree of client interaction required in an agile environment. They often can be troubled by the constant requirement to present their designs and prototypes for client review, and by the need to constantly incorporate changes and improvements as they work. Conversely, many clients who are trained to “throw projects over the wall” to developers—thus, having little contact with the project team between specification and acceptance time—often are unprepared for the intensive interaction expected in an agile engagement. The preparation of teams, and the setting of expectations regarding this new way of engaging, is a key success factor.

- **Responding to change over following a plan:** Developers and project teams are asked to participate in many types of projects. Some are production-style, low-uncertainty projects, such as installing a new server in an existing data center, for which there may be well-known, documented rules and procedures. Others may be innovative, high-uncertainty projects, like creating a new business based on a Web-based business model that is untried and untested. Agile theory contends that it’s a fallacy to believe that all types of projects, from production to exploration, can be accomplished using the same, anticipatory, plan-do type of project approach, and that exploratory, innovative projects require a more experimental, envision-explore approach. Another key element of this philosophy is the changing nature of business itself. As global competition heats up, and the pace of change speeds up due to the Internet and our instant communication culture, the requirements of a project can change dramatically from conception to execution. Project teams are no longer simply confronting a series of minor scope changes that can be handled with change control processes; they’re facing projects in which the entire set of business assumptions, technology, and infrastructure can change as the project is underway. Traditional measures of project success, such as compliance with an anticipatory plan and strict control of change requests, are a poor fit for this type of engagement.

Keep in mind that the longer the planning time frame, the less flexibility an agile team will
have. This does not mean that agile practitioners don't plan. It can be argued that given the iterative and reflective nature of agile projects, more planning takes place in total than in traditional project management. However, agile methodologies are very intentional regarding how much planning takes place and how detailed it is. Agile practitioners attempt to not limit their ability to react to change (or opportunities to generate it).

**Agile Principles**

The *Agile Manifesto* was published along with a set of 12 guiding principles, which help to clarify and highlight the underlying philosophies of agility:

<table>
<thead>
<tr>
<th>Principles Behind the <em>Agile Manifesto</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>• “Our highest priority is to satisfy the customer through early and continuous delivery of valuable [products].”</td>
</tr>
<tr>
<td>• &quot;Changing requirements should be welcomed, even late in development. Agile processes harness change for the customer’s competitive advantage.&quot;</td>
</tr>
<tr>
<td>• &quot;The delivery of working [products] should be frequent, from a couple of weeks to a couple of months, with a preference to the shorter timescale.&quot;</td>
</tr>
<tr>
<td>• &quot;Business people and developers must work together daily throughout the project.&quot;</td>
</tr>
<tr>
<td>• &quot;Projects should be built around motivated individuals. Give these individuals the environment and support they need, and trust them to get the job done.&quot;</td>
</tr>
<tr>
<td>• &quot;The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.&quot;</td>
</tr>
<tr>
<td>• &quot;Working [products] is the primary measure of progress.&quot;</td>
</tr>
<tr>
<td>• &quot;Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.&quot;</td>
</tr>
<tr>
<td>• &quot;Continuous attention to technical excellence and good design enhances agility.&quot;</td>
</tr>
<tr>
<td>• &quot;Simplicity—the art of maximizing the amount of work not done—is essential.&quot;</td>
</tr>
<tr>
<td>• &quot;The best architectures, requirements, and designs emerge from self-organizing teams.&quot;</td>
</tr>
<tr>
<td>• &quot;At regular intervals, the team reflects on how to become more effective, and then tunes and adjusts its behavior accordingly.&quot;</td>
</tr>
</tbody>
</table>

**What Is Scrum?**

**Scrum Values and Pillars**

The core of the Scrum approach is the belief that most systems development has the wrong philosophical basis...you can’t predict or definitely plan what you’ll deliver, when you’ll deliver it, and what the quality and cost will be. Development is not a defined process that can be
repeated time after time (like the implementation of a well-known system). Instead, it is an empirical process in which each project is fundamentally different; new discoveries must be made every time; and the process unfolds differently each time based on the findings of each unique project. In other words, this approach is based on learning from experiences and using these experiences as soon as possible.

It’s important to distinguish the process from the content of the project in thinking about Scrum. While the Scrum process can be consistent from project to project, the content to be developed—whether a new bit of software, a new system, or a new product—will be sufficiently innovative and unique so that the team will discover and integrate new ideas within each project as they work, create, and discover. Scrum can also be used for projects where the product is very clear and no product innovation is desired. In these particular cases, Scrum focuses on process innovation: how to deliver faster and faster each sprint. Process innovation is what makes a technologically great product affordable for its customers.

Unlike some agile methods that are designed with a programming-centric approach, Scrum has more of a project management focus. With the uncertainty of predicting the outcomes quoted above, Scrum concentrates on developing a supportive environment for creative development by closely examining the features delivered and constantly fine-tuning the product through increments as it is delivered iteratively. As in all agile methods, the supportive environment emphasizes collaboration, constant customer involvement, and the drive toward iteratively delivering real working features.

The philosophy of Scrum centers on a few simple ideas: Predicting the outcome of innovative, creative projects is folly; dedicated, talented teams have the initiative and creativity to build the requested features and functions if enabled to do so by a supportive environment; and the delivery of working functions and constant oversight of the content (rather than the process) makes the successful development of inventive products possible.

The Scrum methodology is underpinned by three “pillars:” transparency, inspection, and adaptation. There is a simple connection between all three. The premise we start from is that this method is based on continuous improvement. In order to improve, you must have fully transparent processes and actions so that when you perform any inspection, you understand what is most necessary to adapt. In other words, if there is no transparency, inspection will not reveal anything. Consequently, there won’t be any adaptation. More on the Scrum pillars follows:

- **Transparency**: All Scrum processes, when run correctly, offer full transparency. The product backlog is public, so anyone can see what the team is and will be working on. The daily stand-up (which will be discussed in more detail later) is a great example of transparency. Every team member must publicly explain what he or she has done, what will happen next, and what problems might block the progress. At the end of the sprint, the review and the retrospective (also to be discussed later) are both good transparency exercises with the customers and with the team.

- **Inspection**: Although Scrum doesn’t offer specific “monitoring & controlling” processes, there is a lot of inspection occurring. The difference between agile and traditional inspection is that instead of management traditionally inspecting the team’s performance to check for variation to plan, in agile, the team replaces management control with “peer pressure.” Every team member needs to understand that any activity needs some sort of inspection to
ensure it is done properly and that it delivers the expected result. Inspection must produce information and explanations when things go differently than expected.

- **Adaptation**: Once a difference has been noticed or an obstacle identified, the next step is to determine what actions should be taken to get to a desired status. Adaptation is about what can be done differently and what can be changed to achieve better results.

In addition to the pillars, Scrum has five values of its own, which are not to be confused with the four agile values from the Manifesto. The Scrum values are the following:

- **Commitment**: Scrum teams are empowered to make decisions; therefore, they are more committed to delivering great results. They contribute with everything they know and can do to deliver a product increment after each sprint.

- **Focus**: The more things you need to work on, the more defects you are likely to produce. In Scrum, the focus is on a limited amount of work at a given moment to ensure it has proper attention and consideration.

- **Openness**: Because Scrum is based on identifying ways to do things better, teams are always looking for ways to improve. In order to do that, they need to be very open and admit reality as it is.

- **Respect**: Scrum team members should help each other whenever necessary. Doing so is the basis of true agile work and requires a lot of respect for colleagues. The idea of continuous improvement demands respect, even for those team members who are inexperienced or have made mistakes. A team is a team for better or for worse, and respect for one another is what allows the team to improve.

- **Courage**: Team members must show courage to admit problems or deficiencies they might have. They understand that the sooner they face the problem, the sooner it will be solved. Additionally, it requires courage to embark on a project when the scope is unknown and certain factors are beyond your control.

### Scrum Artifacts

Scrum has four artifacts that are produced, managed, and adjusted throughout the life of the project. They are the following:

- The product backlog is the list of all the features that the project needs to deliver. It consists of work items, which is the general term referring to features, themes, epics, or user stories. The product backlog is always prioritized by business value and benefits for the customer. The higher the value, the more priority that a particular functionality gets and the sooner it is worked on. The product backlog includes both user-defined functionalities and technical requirements, and belongs to the product owner. However, all items in the product backlog need to be expressed from the end-user perspective.

- The sprint backlog is the part of the backlog being worked on in the next sprint. It consists of user stories that are estimated in story points and split into tasks during the sprint planning session. It is the output of sprint planning and belongs to the team.

- The product increment is the end result of the user stories that were completed (according to the definition of done) in the current sprint. The product increment is the team’s responsibility and needs to be accepted by the product owner. At the end of the sprint—after a demo in the sprint review—the product increment goes into production.
- Burn-down/burn-up charts are cumulative charts that display an evolution over a period of time. A good example is how a chart documents a team’s accumulated story points sprint by sprint.

**Scrum Framework**

The framework for the Scrum process centers on four ceremonies:

- **Sprint planning** has two parts. In part one, the product owner explains what he or she wants from the next sprint. This is known as the “WHAT,” and it refers to user stories being selected from the product backlog. In part two, the team, without any external influence from the product owner, splits the user stories into tasks. This is known as the "HOW." The result of the sprint planning is the sprint backlog.

- The **daily stand-up** is a meeting with a maximum duration of 15 minutes. Every team member responds to three questions:
  - What did I do since the last meeting/yesterday?
  - What will I do until the next meeting/today?
  - What obstacles are preventing me from doing my job?

  The daily stand-up helps improve transparency and problem-solving for Scrum teams.

- The **sprint review** is basically a demonstration of the product increment that was produced in the last sprint. The purpose of this meeting is to get feedback from the customer and other stakeholders to improve the product being built and to make sure it remains consistent with the customer’s desire.

- The **sprint retrospective** is the moment when the team reflects on how it can become more efficient and more productive. The question asked and answered here is, “How can we deliver more functionality in the next sprint(s)?”

In order to grasp Scrum, it's critical to understand the backlog concept, as illustrated in the figure below. The entire product, to be delivered when the project is complete, is defined by the features and functions captured in the product backlog. Those features are then prioritized by the product owner (assisted by the team) to determine which features will be included in the next sprint. The prioritization is always based on the business value for the customer.
Other Methods

Extreme Programming (XP)

Extreme Programming, or XP, has garnered a lot of interest among agile methods. XP drew attention because its audience of software developers were discovering many of its practices in real project work. Additionally, its initial successes, including the well-known Chrysler Compensation project, coincided with the Internet movement, with its need for an approach geared to speed-to-value and exploratory, innovative projects.

XP includes elements such as the following:

- **The planning game**: A recurrent workshop in which developers and customers interact in order to create and refine the “stories” that describe their project.

- **Small releases**: Every release should be as small as possible, containing the most valuable business requirements.

- **Metaphor**: The overall idea of the project; the broad goal told as a narrative or story to keep the technical jargon to a minimum and to build a collaborative vision between developers and customers.

- **Simplicity**: Simplicity is central to XP. The delivered product should be simple, delivering the needed features in the simplest manner without trying to speculate about the bells and whistles that might be useful sometime in the future. The use of methodology and technique should be simple too; XP developers avoid documentation other than stories and test cases, unless there’s a convincing demonstration of their value to the customer. Finally, XP practitioners strive for elegance and simplicity in their actual coding practices, which leads to refactoring.

- **Refactoring**: Refactoring is a key element of XP. It is the optimization of the internal code without changing its behavior; a response to one of the hazards of iterative design, which is
the danger that the separate iterations will be poorly integrated and internally incompatible; and a disciplined approach to rebuilding system internals to ensure simplicity, elegance, and compatibility.

- **Pair programming**: Pair programming takes the concept of software inspections and walk-throughs to the next level. Rather than periodic reviews, the key insight of pair programming is that two programmers can review and optimize each other's code and coding practices as they work toward the customer's goal. Simply put, two programmers are writing code using only one computer.

- **Testing**: The vital difference between XP testing practices and traditional practices is that XP insists that test cases for all features be developed up front, with the stories. This is also known as test-driven development (TDD). A failing test is referred to as “red,” so at the beginning of the development when there is no code, all tests are red. Code is written until tests become green (meaning they are passed). Any further development results in more functionality or quality than required and means overprocessing.

- **Continuous integration**: Going beyond the “daily build,” which is a common practice in many commercial software companies, XP practitioners prefer the continuous build, ensuring continuous compatibility and functionality as the product is created.

- **Sustainable development**: A reaction to the 70-hour-week “death march” project that many developers have experienced, the 40-hour-week standard (or similar standard by organization) that XP espouses is consistent with the agile philosophy that creative developers do their best work when they're committed, energized, clear, and focused.

- **Available customer**: XP calls for the customer to be completely integrated with the development team; available to review features, builds, and tests; and to review, assess, and optimize the product as it evolves.

XP's broad exposure and the debate it has engendered have given the project management and software development communities the chance to consider these ideas and to explore their applicability to their work.

**XP Process**
Dynamic Systems Development Method (DSDM)

Developed in 1994, the Dynamic Systems Development Method, or DSDM, is the oldest of the recognized agile frameworks. DSDM was developed as a structured approach to rapid application development (RAD), a popular methodology of the early 1990s. DSDM incorporated many of the fundamental concepts of iteration, incremental delivery, and customer collaboration that typify the agile methods that have become popular since the *Agile Manifesto* was published. DSDM is defined as a project delivery framework that aids the development and delivery of business solutions to tight timescales and fixed budgets. It is focused on the concept that customers can’t foresee all the detailed workings of their system or product in advance. The *DSDM Manual* informs us that “the current step need be completed only enough to move to the next step,” a nice distillation of the incremental, iterative approach common to agile methods.

DSDM is unique among methodologies in that it follows a consortium model, with members paying an annual fee (of about $2,000 a year) to remain members and retain access to the licensed use of the methodology. As such, it has the most developed training, support, templates and tools, and accreditation services of the agile methods (although Scrum, with its Scrum master series of certifications, is close). However, the consortium structure has also been a factor in DSDM’s slow adoption.

As depicted in the figure below, DSDM is an interlocking set of processes, each of which is iterative and incremental. While the model is complex and daunting, it is basically a three-step iterative model consisting of the Modeling, Design-Build, and Implementation phases. Once mastered, it has been proven to have some significant benefits in speed, team productivity, customer satisfaction, and product fitness for its visualized use. More detail on the individual processes is available at the DSDM consortium’s Web site.
DSDM Processes

DSDM, like many agile methods, promoted enforcement of “time-boxes” with preset start and end dates, and with a preselected team. Functionality may change, but the delivery date does not. This strict time management is a reaction to the ubiquitous problem of missed delivery dates for new software and products.

DSDM employs a strict method for prioritizing requirements (known as MoSCoW), which categorizes features as “must have, should have, could have, won’t have (low value or high value but hard to achieve).” Since functionality can change but time and team composition don’t, DSDM’s emphasis on feature priorities helps teams to focus on the features that deliver business value and relentlessly drive for efficiency and simplicity.

More relevant to our discussion of agile methods is the convergence between the stated principles of DSDM and those articulated in the Agile Manifesto.

In 2007, DSDM rebranded the framework under the name of DSDM Atern. Atern is the culmination of practitioners’ experiences drawn from a wide range of public-sector and private-sector projects spanning more than a decade. This unparalleled expertise has resulted in a robust but agile framework that can be successfully applied across a variety of organization and project contexts.
Below are the highlights of the eight DSDM Atern principles:

- **Focus on the business need**: Every decision taken during a project should be viewed in the light of the overriding project goal, which is to deliver what the business needs it to deliver, when it needs to be delivered.
- **Deliver on time**: Delivering products on time is a very desirable outcome for a project and is quite often the single most important success factor.
- **Collaborate**: Teams that work in a spirit of active cooperation and commitment will always outperform groups of individuals working only in loose association.
- **Never compromise quality**: The level of quality to be delivered should be agreed at the start. All work should be aimed at achieving that level of quality. If the business agrees that the features in the minimum usable subset have been provided adequately, then the solution should be acceptable.
- **Build incrementally from firm foundations**: In order to deliver real business benefit early, Atern advocates incremental delivery. This encourages stakeholder confidence and is a source of feedback for use in subsequent increments.
- **Develop iteratively**: In order to converge on an accurate business solution, Atern uses iterative development. It is very rare that anything is built perfectly the first time, and projects operate within a changing world.
- **Communicate continuously and clearly**: Poor communication is often cited as the biggest single course of project failure. Atern techniques are specifically designed to improve communication effectiveness for both teams and individuals.
- **Demonstrate control**: It is essential to be in control of a project at all times. An Atern team needs to be proactive when monitoring and controlling progress.

DSDM is a noteworthy example of a methodology that was developed prior to the release of the Agile Manifesto, but it has evolved significantly to be consistent with the ideas generated by the agile community.

**Crystal Methods**

Crystal was developed by Alistair Cockburn to get rid of the aberration called software engineering. He felt that engineering gives us questions like, “Is our model accurate?” instead of the more interesting questions such as, “Is our product meeting the customer’s needs?” and “Do we have our goals aligned as a team?” In line with Cockburn’s emphasis on these questions, Crystal is primarily focused on communication, with a special focus on interaction, community, people, and skills.

As noted, Crystal is a collection of methodologies based on two fundamental assumptions: Teams can streamline their process as they work and become a more integrated, optimized team; and projects are unique and dynamic, and require methods that are designed specifically for each effort. Another theme that has run through Cockburn’s work from his first articles to his current Crystal development work is the concept of product development as a game—a cooperative, interactive activity that should be designed in a way that stimulates the creativity of all participants.

As illustrated in the figure below, Crystal Methods recommend a specific methodology for each
project based on three critical factors: communication density (more people means more communication channels and increased complexity), system criticality, and project prioritization. The smaller the team, the more likely it can just retire into a conference room and build the product without a lot of status reporting, knowledge transfer, or written documentation. As the team size increases, the number of communication “artifacts” (like written notes) necessarily increases, and Crystal Methods enable that increase by presenting different methodologies and processes for different projects.

**Crystal Methods**

Cockburn has the advantage of years of research, both as an IBM researcher and an author, on what really works in software development. This pragmatic, ground-level view has driven him toward simplicity and minimalism in process. Cockburn feels team members understand and communicate their work dependencies. They can do this in lots of simple, low-tech, low-overhead ways. It is often unnecessary to introduce tool-intensive work products to manage it.

**Lean Development (LD)**

In 1989, Professor James Womack and consultant Dan Jones published “Lean Thinking,” a survey of the lean manufacturing techniques that helped to create the “Japanese miracle” of the late 1980s and early 1990s. With a focus on eliminating waste by creating a smooth “flow” of work on the factory floor and expecting workers to contribute high skill levels and an ownership mentality, they chronicled the ideas of lean manufacturing. These concepts helped Toyota, the exemplar of these techniques, vault over the traditional giants of the automotive industry. Lean manufacturing theories were highly influential in the creation of LD.

Bob Charette, while not a signatory of the *Agile Manifesto*, has developed a methodology that
has many commonalities with those mentioned so far. Similar though distinct ideas also have been put forward by Mary and Tom Poppendieck in their book *Lean Software Development*.

LD emphasizes four key success factors that clearly illustrate LD’s compatibility with other agile methods:

- Create visible customer value rapidly.
- Build change-tolerant software.
- Create only necessary functionality and no more.
- Be aggressive and stubborn, and challenge assumptions.

Like Scrum, LD is more of a project management environment than simply a software development one; it consists of three distinct phases: start-up, steady-state, and transition/renewal. Rather than the daily “Scrum,” it recommends a time-boxed “whirlpool” that, like all agile methods, includes the analysis, design, test, and build activities in each iteration.

LD is important not just for its conformance to the ideals of agile development but because the underlying philosophies of lean manufacturing have been accepted by business leaders worldwide. This makes the introduction of agile methods in a lean framework more easily accepted and presents a strategic framework that executives are likely to accept with less resistance.

![Lean Development Diagram](image)

Lean principles have become extremely useful for teams involved in any of the agile practices. They are the following:

- Eliminate waste.
- Amplify learning.
- Decide as late as possible.
- Deliver as fast as possible.
Build integrity in.
See the whole.

Lean’s obsession for eliminating waste has created an entire literature around value-driven delivery. Most notably, waste has been categorized into seven types:
- Transportation
- Inventory
- Motion
- Waiting
- Overprocessing
- Overproduction
- Defects

Interestingly, an eighth type of waste has been identified as the "unused potential" of the teams.

Unified Process (UP)
Unified Process, or UP, and its proprietary refinement as IBM’s Rational Unified Process (RUP), is an early example of an iterative and incremental software development process framework.

UP was first presented in 1999 in the book *The Unified Software Development Process* by Ivar Jacobson, Grady Booch, and James Rumbaugh. The authors are well respected in the software development community not only for this contribution, but also for their pioneering work in the development of Unified Modeling Language (UML) and the creation of standardized use-case formats.

Since RUP is a trade name owned by IBM after its acquisition of Rational Software in 2003, the term UP is used typically when referring to the generic process popularized in Jacobson’s book. Again, the key practices recommended in UP correspond nicely with the agile ideas that have so far been reviewed in this course:
- Develop in time-boxed iterations.
- Develop high-risk, high-value elements early and then reuse them.
- Deliver value to your customer.
- Accommodate change.
- Work together as one team.

In UP, there is a four-phase framework:
- **Inception:** This stage focuses on the development of the vision, risk list, and business case, and includes a workshop designed to derive the “top 10 percent” of the detailed requirements. The stress on only 10 percent of requirements is designed to ensure that only the key architectural elements are defined at this stage and that they form the foundation of the first iteration of the product.
- **Elaboration:** The core architectural elements of the product are developed and tested during this phase. This time-boxed, iterative phase includes, as with most agile methods, both the development of the product and the ongoing analysis, requirements definition, and
optimization necessary to satisfy the business vision. This stage ends when all agree that a solid foundation, on which the remainder of the system can be constructed, has been accomplished.

- **Construction**: This stage includes time-boxed iterative development of the remainder of the product, plus any acceptance testing, validation, and user documentation that the customer may require.

- **Transition**: The system transitions to production in this stage and includes any parallel runs, user training, knowledge transfer, data migration, or other production activities.

As illustrated in the figure below, the typical product development activities, such as requirements, analysis and design, and deployment, occur throughout the various phases of UP, reaching differing peaks of activity depending on their timing within the project.

**UP Phases**

Other key elements of UP include the following:

- **Use-case driven**: Not surprisingly, since the originators of UP were also founding figures of UML, UP recommends the use case as the mechanism for capturing requirements and defining iterations.

- **Architecture focused**: A key element of the requirements and iteration process is the idea that an architectural core be developed first, with functional extensions then added in later iterations.

- **Risk discipline**: Risk identification and management throughout each cycle is a key part of UP.

RUP, originally developed by Rational Software Inc. and acquired by IBM in 2003, is a licensed software version of the development methodology, with proprietary processes, templates, training aides, and tools.

While IBM’s product is proprietary and licensed, the UP defined in Jacobson and Booch’s book can be adopted by any team.

There are, of course, other variants that haven’t been explored here, including feature-driven development and Jim Highsmith’s adaptive software development. Some, like Scrum, are
focused primarily on the project management element of product development, while others, like XP, are software centric. The agile ideas outlined here, from the early theoretical findings to the Agile Manifesto, have not just created debate and discussion, but have led to the crafting of a variety of disciplined, complete methodologies that bring the theories into real-world practice.

The Scrum Team

Roles

Scrum only recognizes and accepts three roles in the process:

- **Product owner:** The product owner is responsible for delivering value to the customers. His or her responsibilities start with sharing the vision (sometimes even creating it) and continue across all levels of planning: roadmap, releases, and sprints. It is only inside the sprint that the product owner steps back and allows the team to focus on delivering the sprint backlog. Key tasks for this role include—
  - Collecting requirements from the stakeholders and putting them in the right format (epics, user stories)
  - Decomposing big chunks of work into smaller pieces
  - Prioritizing the backlog
  - Providing support for the team
  - Engaging with external stakeholders every time it is necessary
  - Ensuring there is something to demo to the customer at the end of each sprint

The product owner is the only role authorized to talk to external stakeholders. Everybody who wants to interact with the team must be directed to the product owner. If needed, team members can interact with other people but shouldn’t be disturbed by them.

- **Scrum master:** The Scrum master is responsible for the Scrum ceremonies, roles, values, and artifacts being used with fidelity. One of the key responsibilities of this role is to help the team become better at Scrum by facilitating its processes. This role is often referred to as a “servant leader,” which means the Scrum master must help the team with whatever it needs to get the job done. Note that this doesn’t mean that the Scrum master solves all of the team’s problems; rather, he or she will help the team solve their own problems. The Scrum master is also responsible for protecting the team from any interruptions.

- **The (development) team:** The team is mostly responsible for delivering a product increment after each sprint according to the acceptance criteria provided in the definition of done. It is also in charge of finding ways to improve its performance from one sprint to another. Unlike traditional project management methodologies where the project team is expected to follow the plan and only implement tasks mentioned in the plan, in Scrum, the team is responsible for helping to create the product and finding the best way to deliver it. An interesting way of looking at the team is to consider it responsible for reducing costs by delivering increasingly more functionalities in the same time interval (sprint).
Best Practices for Team Success

For a Scrum team to be successful, a few factors are important:

- **Size**: A Scrum team typically has **seven people** (plus/minus two). It needs to be small enough to synchronize efficiently and big enough to have all the necessary competencies required to deliver a relevant product increment.

- **Dedicated**: Although sometimes difficult, especially in functional organizations, it is preferable the team only has to focus on **one project** at a time. This is not as much about time management and multitasking (although those matter a lot!) as it is about energy, commitment, and focus.

- **Self-organizing**: The team needs to be **self-organizing**, which means that it doesn't need any external authority to manage it. The team can organize itself and make the best decisions in the interest of the project. The concept of self-organizing is based on the reality that in projects, it is the project people who are in the best position to decide and have the best information and skills to result in a favorable outcome.

- **Empowered**: In order to be able to make decisions, the team needs to be **empowered**. This means the team needs to have the power and authority to make the decisions it considers best for the project. In many situations, empowerment is connected to consequences. The team needs to understand that even if it makes mistakes, there won’t be any negative consequences as long as it acted in the project’s best interest. This is particularly important because, as agile is based on learning and improving, there will be some less-than-perfect decisions and even some mistakes.

- **Generalizing specialists**: To minimize waste and be more efficient while working, a team needs to consist of **generalizing specialists**. That means that every team member, although an expert in a field, will sometimes act as a generalist and help colleagues in need. That doesn’t mean that people will do each other’s jobs but, rather, that people are expected to jump in whenever necessary to help colleagues unblock a user story. This requires a shift in focus from individual tasks to team results.

- **Cross-functional**: A Scrum team is expected to deliver a fully functional product increment at the end of each sprint. For that, it needs to be **cross-functional**, which means that it will contain all the skills and competencies necessary to deliver the product increment. Simply put, as a general rule, the team doesn’t require external help to do its job. Everything is done with people from the team. The team needs to understand that it doesn’t depend on anybody else to deliver results. Typical activities include—
  - Discussing requirements with the customer and product owner
  - Design
  - Architecture
  - Graphics
  - Software development
  - Testing
  - Deployment
Module Summary

While the Agile Manifesto is relatively new, the agile movement’s beginnings are related to Toyota’s new approach to manufacturing in the 1950s. Agile has become a set of methodologies that focus on the customer, delivering great quality and embracing change. The most important document of agile is the Agile Manifesto (2001), which contains 4 values and 12 principles to guide practitioners. In addition to values and principles, agile also has some methodologies: It is simple to use and lightweight in process, and it offers PMs and teams a set of tools to better do their jobs. The most relevant agile methodologies are—

- Scrum
- Extreme
- Lean

Scrum is a lightweight agile methodology that relies on a team to deliver a product increment after each sprint. It consists of:

- Three pillars
  - Transparency
  - Inspection
  - Adaptation
- Five values
  - Focus
  - Commitment
  - Openness
  - Courage
  - Respect
- Four ceremonies
  - Sprint planning
  - Daily stand-up
  - Sprint review
  - Sprint retrospective
- Three roles
  - Product owner
  - Scrum master
  - The team
Module Overview

From a Vision to a Roadmap

Vision is a very high-level expression of why the project is undertaken. It sounds more like a promise than a plan. The product owner, together with the sponsor and sometimes with other stakeholders, is in charge of turning this vision into something more tangible and real than just a phrase. The output of this process is the roadmap, a collection of high-level functionalities to be delivered by the project and prioritized in time according to the benefits they will bring to the customer.

The roadmap serves as a view of what key features need to be included in the product and how the product will evolve in time. Usually a roadmap covers several months or even years. We say that in agile, we plan at five-time horizons. These are—

- Vision
- Roadmap
- Release
- Sprint
- Day

A roadmap, then, is still very general but gives guidance as to what priorities each feature has. In agile, we start the most important functionalities and deliver them first. A product might not have all the nice accessories from the beginning but is final (“done”) and ready to go into production. Essentially, the product is good enough to be used by the end user.

One important aspect of the roadmap is evolution. How will the product evolve? What other new features will be added? Have we already added what was the most important? The roadmap needs to make very clear what the most important aspects the team needs to focus on are and what features are more optional. These may not offer a lot of value but still need to be done. Since the roadmap is in fact a collection of releases, it also provides some information about the deadlines and milestones that need to be respected. They need to be captured at this level because they will drive later planning, including how the roadmap is split into releases and sprints.
From Vision to Product Backlog

For those familiar with traditional styles of project management, the concepts of subprojects and project phases are well-known. For example, a data center implementation project typically will consist of a facilities subproject, a subproject to install the racks and cabling, and another subproject, perhaps with a different PM and team, to bring in the computer hardware and set it up in the racks. While the build-out of the facilities and the racks won’t deliver much in the way of features or functions to the user community, there’s probably a consensus that these tasks are necessary precursors to the delivery of the data center. A data center build-out is probably not the best example of an agile-appropriate project, but it illustrates the idea of subprojects that fit into the greater project objective of delivering business value to the customer.

In the agile world, the corresponding concept of interim phases that collectively leads to the delivery of the total project is the “release” concept. Two ideas are central to understanding the difference between the subproject, or phase model, of traditional project methods and the release model applied in agile projects. First, as reviewed above, traditional projects will deliver phases or subprojects that, while necessary, do not directly deliver user business value on their own. Agile projects deliver some meaningful business value in each release. In industrial design projects, the business value may be a prototype that users can review rather than a real working end product, but the concept remains.

The second central concept is the close and constant collaboration that occurs throughout the development effort. Developers, PMs, and customers don’t wait until the delivery of milestones or the completion of phases before they come together to evaluate the project’s direction. They do so constantly because the user or customer is integrated into the development team, not an outside entity that is kept informed through status reports or project dashboards.

Each project is segmented into five levels of effort: the vision, roadmap, release, sprint, and daily levels.

The envisioning process provides the overall purpose and expectation of the whole product. Once that’s been established and agreed upon, the collaborative developer/customer team performs the release planning process in which the goal is to answer two simple questions:

- When do we want to see the release?
- What is the priority of each feature (epic or user story)?

In regards to the first question, recall that one of the key ideas in agile methods is the time-box, the predefined unit of time in which we agree to deliver some set of valuable functionality. As previously discussed, this core discipline of determining a strict time period for each effort is important to ensure that agile projects remain on track, deliver value while the business opportunity is still current, and enforce the process of prioritizing features so that the customer sees real progress (rather than just seeing percentages of completion on a status report). This approach enables the development team to set reasonable expectations about what can be
delivered in the initial release and to make the difficult decisions about which features will be included (and excluded) in each release.

Unlike waterfall projects—where the triple constraint fixes the scope, and time and cost are variable—in agile projects, time is typically fixed (usually expressed as a number of sprints) and the idea is to generate as much functionality (scope) as possible.

It’s obvious that having some idea of the time, cost, and resources required to deliver a particular feature is needed in order to do release planning. Planning and estimation techniques for agile projects with Scrum will be examined in greater depth in upcoming modules, but at this stage, all of these elements are considered by the team in the creation of a release plan.

This cascading series of iterative efforts, with multiple releases containing multiple iterations within that are all finally coming together to deliver the entire work product, is the fundamental project management approach of the agile framework.

The Product Backlog and User Stories

The Product Backlog and User Stories

Within each agile development effort is an incremental series of releases, and within each release is a series of sprints. Within each sprint, the product owner and development team create a sprint plan and perform the actual development work of coding, engineering, or designing the product that will be presented to the customer at the end of the designated sprint length. During the planning of the sprints, the team will address technical, internal-facing issues that may not be directly pertinent to the customer’s perception of product value. The release plan is typically the customer-facing, business-value-focused plan; the sprint plan is the place for the development team to determine the tactics by which it will achieve the customer’s expectations for this release. This is not to say that collaboration and consultation ends while the team plans and executes the sprints; in reality, the development team frequently must consult with the customer to clarify technical, priority, or feature-related questions during sprint planning.

Two fundamental concepts underlie the iterative project approach that has been discussed in this course: short time-boxes and feature-based development. Sprints are smaller, time-boxed units of work that are planned and executed in an incremental, iterative manner. Time-boxed development enforces efficiency and discipline, and trains developers to be realistic about what they can accomplish within a sprint. Feature-based development, in contrast with activity-based project planning, helps to maintain focus on the customer’s perception of value rather than on internal technical language. In activity-based planning, a task like “create the domain database server” or “analyze the British Thermal Unit (BTU) requirements” may be planned. While these may be tasks that must get done, they are not expressed in a manner that resonates with customers or helps them to assess the progress toward their objectives. Feature-based development encourages the discussion of features in a language that has meaning to the customer.
User stories document the features that customers expect in their product. Those features, such as "a user can search for flights by departure and destination cities," are the input to the sprint planning process. All the features that have been collected are now accumulated into a features list and must be prioritized and allocated to the releases and sprints, during which they will be developed and delivered. In Scrum, this features list is referred to as a product backlog. It represents the catalog of commitments that the developers have made to the customer. The team has agreed that it will attempt to implement the features documented in the backlog and will divide its development into sprints and releases that will make sense to stakeholders, and enable the customer to view and optimize the product as it is developed.

Before a discussion can take place about the process of creating a prioritized features list from a collection of user stories, a few concepts are worth noting. First, it's important to remember that not only is the product being incrementally optimized, but the process of delivering the product is also being optimized, thus leading to a gain in efficiency. This is a key element of agile thinking: The belief that individuals get better at developing by developing together as a team. Many agile approaches emphasize the importance of reviews during a sprint and retrospectives at the end of a sprint. Every development team is a unique collection of individuals, skills, experiences, and perspectives, and agile approaches emphasize the importance of consciously integrating those attributes to build a more efficient team.

Secondly, agile teams should always assume that the features list they're working from today will differ from the one they're using tomorrow. Agile teams accept the idea that the customer can always request new features, discover that old ones are not as useful as expected, and decide that certain features are higher or lower in priority. Developers must resist getting attached emotionally to the features they've accumulated in their backlog; technically "cool" or "elegant" features that technicians love often get dumped by the customer, emphasizing again that it is the customer who defines value.

**Elements of Release Planning**

**Elements of Release Planning**

When planning the release, a few elements are essential. First is the vision. The product owner needs to make sure there is a vision that can be shared with the team and that the vision has been spread across a timeline as a roadmap. Of course, by this time, there needs to be a product owner and a relatively firm list of critical stakeholders.
The Roadmap

Since the roadmap is about time, it is very important that it gets prioritized as soon as possible. Prioritizing the roadmap ensures delivery of the most important features at the beginning. As the project moves from roadmap to release planning, the Scrum master and the team begin their work.

The Release Plan

When it comes to release planning, there are a number of essential questions to ask and answer:

- How do you want to deliver your product to the customer?
- How often?
- How consistent does each delivery need to be?
- How much feedback is needed after each delivery?
- How available is the customer to support these deliveries?
- How much unknown is in the project?
- How experienced is the team?

One of the major inputs to the release plan is the backlog of features that will be delivered as a product backlog composed of user stories. A full user story life cycle includes the following:

- The idea
- The epic that contains the idea
- The user story that explains the benefit of the idea
- The acceptance criteria that confirms the benefit was delivered
- The tasks that will implement the idea (tasks are the team deciding how to deliver the user story)

The concept of user stories will be discussed in more detail later in the guide.

After the release plan is prepared, it should tell when every feature will be delivered and why. “When” is a matter of which sprint, as a release is a collection of sprints, and “why” refers to the
benefit that each particular feature brings to the customer.

Additionally, some other important aspects need to be agreed on. First, how long should the planned backlog be? How many sprints should we plan ahead? It is important to strike a balance between planning and forward momentum. In other words, it can be useful to agree on certain things as early as possible; however, recall that the longer we plan, the less reactive to change we can be. Planning in short steps (like just one sprint ahead) provides more flexibility and allows teams to incorporate changes more easily.

The second major additional consideration is how detailed our planning should be. Just because we have all the elements for a detailed plan doesn’t mean we should construct it. Again, we should plan just as much as we need to execute and keep future work plans general enough to accommodate any change or feedback that might arise from our current work.

Sprint Zero

On many projects, the team is allowed a sprint zero before starting development. There are a number of reasons to include a sprint zero, including—

- Getting used to the project
- Understanding Scrum as a methodology
- Setting up the environment

This sprint is dedicated to making sure the next sprints have all they need to be run successfully. Sprint zero also helps to validate the vision and ensure that everybody understands and shares it. Agreement must also be reached on what the definition of done is for the project as a whole, the release, and the user stories. This means that the product owner has defined "done" according to the business needs of the customer. The team must also understand the definition and validate its feasibility from the technical perspective.

At this point, the team should be formed and ready to start working. There are some final considerations to be addressed in sprint zero:

- Verify that the team is cross-functional and can deliver the product increment on its own without asking for help outside the team.
- Assess the team’s familiarity with Scrum as a method. If some team members are not familiar, then a short intro or training might be in order. At the very least, the Scrum master and product owner should have some meaningful experience, as team members will expect them to drive key aspects of the process and provide guidance in certain situations.
- Determine how the organization will or can support the team. In a typical functional organization, it might be difficult to have people full-time on the project and committed when they have other tasks or projects to attend to. Functional managers may feel uncomfortable with their people involved in a project with no management control. These issues need to be
addressed as soon as possible before concluding sprint zero. Having a strong sponsor is always a good solution.

- The product owner should perform a stakeholder check to ensure relevant stakeholders are involved and aware of the agile approach being taken with this project. It is important that stakeholders get involved from the beginning because Scrum relies on heavy involvement from the customers to create and deliver the product. They need to be aware of how the backlog is built and why we don’t use detailed, up-front specifications. Customers also need to be comfortable with the team making some important decisions. Customers will also be asked to offer proper support at the end of the sprint in the sprint review. Again, it should be made clear that the whole purpose of the demo is to get feedback and not necessarily accept the product.

- Gather all the tools, processes, equipment, and resources the team needs to get started. This activity ranges from buying new computers to installing the software and the Kanban board. Teams won’t be productive without the right tools.

Sprint Zero Checklist

In summary, the checklist below provides a number of the most important considerations for sprint zero:

- An agreed-upon vision and release plan
- A clear and prioritized product backlog
- A determined sprint length
- Team membership, processes, and functioning
- Architectural or design matters
- Assessed risks

Module Summary

Module Summary

A backlog is a list of working items that will be developed by the team in sprints. Typically, a product backlog consists of features, themes, epics, and user stories. You get to the backlog after setting the vision and turning it into a roadmap of high-level functionalities to be delivered. This roadmap is later split into releases with important milestones and deadlines. Every release is a collection of sprints that deliver items from the backlog.

We use a five-level planning horizon when we look at our project:

- The vision
- The roadmap
- The release
The sprint

Before actually starting the project, it is recommended to do a sprint zero. This is necessary to make sure that everybody is on the same page and that the team has everything they need to start working.
Prioritizing the Backlog

One important agile principle is that we delight our customers through early and continuous deliveries. Not only are product increments delivered early, but the most important features are delivered in the beginning of development, and then less important and optional features come last. A common phrase in agile circles is that in agile, you "start with the cake." There are several reasons for this approach. The most important is that we want to help our customers realize business benefits from the project as soon as possible. Instead of waiting until the end of the project, we assist them in monetizing the most important features as early as possible.

This value-driven approach is key to delivering agile projects. It involves the team helping to decide what value is, validating it with the customer, and continuously improving upon the value delivered. If done right, by the time the project schedule is halfway complete, the project has already delivered up to 70–80 percent of its value.

Prioritization Techniques

MoSCoW

The MoSCoW technique consists of splitting the backlog as follows:

- **Must have:** Things that are core to the system
- **Should have:** Things that are essential to the system. Together with "must have," it provides the minimum for a system to work
- **Could have:** Things that are nice to have but don’t bring a lot of value
- **Won’t have:** Things that we postpone because either they don’t provide value or they do, but they’re impossible or too expensive to do
In time, features have the tendency to migrate from "won’t have" to "could have" and eventually to "should have/must have."

**Kano**

Developed in the 1980s by Professor Noriaki Kano, this model looks at products and features on two axes: need and satisfaction. From the intersection of these axes, we discover that some attributes are delighters (great satisfaction and some need fulfilled), some are performance attributes (great satisfaction while fully satisfying the need), and some are basic (not that great satisfaction but will solve the need).
Sprint Timing

Sprint Calendar

A full sprint calendar consists of the following sequence:

- Sprint planning
- Development
- Sprint review
- Sprint retrospective

There shouldn’t be any pauses between these ceremonies unless the team needs to perform a spike that will impact the future sprint. A spike is a time-boxed period of time given to the team to do some research and development, and to come back with the best possible solution to an encountered obstacle. Otherwise, these ceremonies will flow one after the other and at the end, after the sprint retrospective, the team will immediately begin a new cycle with sprint planning.

Sprint Duration

A common question in Scrum is how long the sprint should be. A common guideline is two-to-four weeks, but it should be the product owner and the team who decides. A sprint should be long enough to create relevant value for the customer at the end. Remember, whatever is done in a sprint is potentially shippable and can go into production. On the other hand, the sprint needs to be short enough to allow many opportunities for feedback from the customer. Shorter sprint durations lead to more opportunities for sprint reviews and retrospectives, which provide feedback on product development and the opportunity to reflect on improving team functioning.

Some additional aspects that should be considered when deciding on sprint duration include—

- The overall project length
- The expected frequency and the impact of anticipated changes
- The overall risk in the project
- Team maturity
User Stories

The 3Cs

User stories focus on the features that users expect to be available when they use the finished product. A key element of these stories is that they concentrate on features that users value and interact with directly. While internal components of the final product are important to developers, users typically don’t care about the internal technical details of the product under development. They want it to work as expected and to handle their transaction volume, but unless they are technicians themselves, they aren’t interested in the granular technical specifications. While the technical details may be critical to the performance of the product and typically would be documented in a traditional requirements process, they are not part of the users’ direct experience with the product and are not part of the user stories technique.

Much less formal than the specifications documented in a traditional requirements process, user stories typically are captured in a conversation with an expected user of the system and describes in a short, unceremonious manner what the new product will do. For a travel Web site, for example, a typical story might tell us that, “A user can search for flights by departure and destination cities.”

Short scenarios of user expectations are just part of the user stories’ process. User stories also include two additional elements:

- Notes from further discussions about the story that help to clarify the expectations and intent of the story
- Validation tests that will confirm to the user that the story, when delivered, does what it is expected to do

Ron Jeffries, one of the signers of the Agile Manifesto, has called the user stories’ information “Card, Conversation, Confirmation” (as seen in the figure below).¹

The card captures the user’s description of the product feature expected; the conversation allows for elaboration of the story so the developer can understand the context and details; and the confirmation sets up the test cases that the customer will use to ensure that the product does what is expected.

Rather than being captured in a highly structured template or in a requirements software product, user stories usually are written on a note card. These cards typically are handwritten index cards and are often referred to as story cards. Once the user is satisfied that the essence of his or her expectation has been captured, these story cards will become the features that will be used to prioritize, plan, and estimate the work that will go into the iterations, releases, and products being built.

As previously mentioned, one of the key tenets of agile methods is that collaboration and conversation occur constantly throughout the project, rather than just at the beginning and end of the project. The agile requirements cycle illustrates this theory perfectly; unlike predictive methods, in which the requirements are developed and accepted by the customer once at the beginning, the agile requirements process depends on constant collaboration and conversation throughout the engagement.

This is the conversation element referred to earlier. Simply asking the users to describe their expectations, and documenting them in a requirements template, or even on user story cards, is not sufficient in the agile environment. The agile approach to understanding the details of the story is not to write an increasingly complex story of requirement; it is instead to work closely with the customer continuously throughout the project to refine the story and to validate that what is being built suits the expected purpose. Conversation is essential to understanding the value for the customer and, most importantly, why the customer wants this particular story.

This is also true of the confirmation component of the user stories’ process. How do agile developers know when they’ve created the right product to deliver the features communicated in the user story cards? That’s the question that the confirmation component seeks to answer. By
asking the user exactly what the conditions in which he or she expects the feature to perform, an implicit agreement with the customer is created. If the product being developed performs under the circumstances captured in the confirmation section of the user story card, then the development agreement has been fulfilled. It is extremely important for the development team to understand when exactly the story is done. Failing to capture this definition will cause the team to deliver either less than the customer expected or more, which is labeled as waste (overproduction or overprocessing).

For example, confirmation notes on the travel site mentioned earlier might specify that the travel search will be tested with—

- International cities
- Empty entries in the “destination” and “departure” fields
- A four-letter airport-locator field

Product owners and developers work with the user to ensure that the boundary conditions of a feature are considered and described in the confirmation part of the user story card so that the customer can be assured that the developed product does what is expected.

The key differentiators between the front-loaded requirements definition process described earlier and the feature-based, story-focused requirements process are that user stories—

- Emphasize communication, collaboration, and understanding
- Are created with the expectation that they will be modified and improved progressively as the project moves from iteration to iteration
- Encourage agile practitioners to avoid getting bogged down in predictive details that may change in importance or relevance as the project progresses
- Are feature oriented rather than task oriented, allowing agile practitioners to focus on business value rather than activities
- Allow developers to avoid mixing business features with technical implementation details
- Are captured at a level of detail that enables agile developers to plan for their placement within iterations and releases, and to estimate the effort required to deliver them are general enough to allow the team to come up with ideas for improvement

**User Story Formula**

INVEST is a good acronym to help identify the attributes of good user stories. Well-formed user stories should be—

- **Independent**: Each story should stand alone, avoiding the creation of complex dependencies with other stories that might make estimating and planning difficult. More importantly, the customer should be able to stop the project at any point and still get valuable, independent deliverables.
- **Negotiable**: Since the story cards created are always the basis of further conversation and collaboration with the users rather than a “carved-in-stone” final requirements document, it must be assumed that there will be ongoing negotiation about the priority, form, and function that the stories represent. What should be avoided is getting too deep into the details of a story and thinking that how it will turn out before development begins can be
predicted. The team is encouraged to keep asking “why,” not only to better understand the product, but also to challenge the initial thinking of the product.

- **Valuable to the user**: As previously noted with the atomic specification that is too finely specified, focus should remain on the features that are of value to the user, not on the technical details that are only meaningful to developers and technicians. Value for the customer is used to prioritize a user story in the product backlog. The more value it brings, the higher it is in the backlog.

- **Estimable**: Is the story discrete and clear enough that a development team can review it and come up with at least a rough-order-of-magnitude (ROM) estimate of the effort and resource required to deliver it? A user story is estimable when it can be assigned a number of story points.

- **Small**: As previously noted, the epic that is too large or complex to guide the work should be avoided. For clarity, estimating, and planning purposes, user stories must not turn into epics. Typically, a user story takes between one and three days to complete.

- **Testable**: The story must be explicit enough about the feature expected so that it can pass a simple yes-or-no test. Does it allow the user to pay with a credit card, or doesn’t it? Does it provide assistance to users who don’t know their three-letter airport code, or doesn’t it? This criterion weeds out vague or nonfunctional stories. “The system must be user friendly” may be an admirable goal, but it’s not a story that lends itself to an unambiguous yes-or-no test. This is confirmation—the final C of the 3Cs described above.

The story collection process, like the agile approach itself, is an iterative, incremental process. The envisioning workshop that creates the overall guiding vision leads to the story workshop that develops the feature-based stories developers will use to create the product. The stories, in turn, lead to further conversation and collaboration that define the details to be delivered and the tests that will validate that delivery.

User stories are always written respecting the following template:

- As a (user type) I want (this functionality) so than I get (this benefit).

---

**Using the Scrum of Scrums for Large Groups**

**Using the Scrum of Scrums for Large Groups**

A common concern is how a project can be effectively scaled using Scrum. Essentially, projects can be scaled up by using a Scrum of Scrums. Typically, this is done by multiplying the number of teams and can be used most effectively when a consistent backlog and deadlines exist, which mean that one single team will not be able to do the job.

Every new team that is created needs to have the same attributes as a team in a single-team project. Remember that teams should be self-organizing, empowered to make decisions, cross-functional, consist of generalizing specialists, and generally be composed of not more than nine
persons and no less than five. Every team will usually have its own Scrum master, but it is possible for a Scrum master to work with several teams. However, there will be only one product owner. This is very important, as it is mandatory to have a single point of contact and one authority when it comes to decisions about the backlog.

In some situations, the product owner has a team (the product owner team) that helps with all aspects of the backlog, including—

- Roadmapping
- Release planning
- Sprint planning
- Support for the team
- Feedback

When the backlog gets bigger, it is highly recommended to create such a team. Otherwise, most product owners will find it difficult to maintain the time, energy, or focus to deal with all aspects of product ownership. The product owner can become too focused on the sprints and neglect the higher level aspects of the project, like the roadmap and release planning, or vice versa.

The product owner also decides how synchronized the teams should be, which usually depends on the frequency and importance of the reviews. If teams start at the same time, then product owners should be aware that all ceremonies will take place almost at the same time. It might be difficult for the product owner to attend that many meetings in a short time frame. An alternative is to delay teams by one week, for example, which would cause a review every week given a two-week sprint duration. Again, this might require too much involvement from the customer. Ultimately, it is the product owner who is best positioned to choose the best option for the customer and team.

### Breaking Down Large Stories

#### Compound and Complex Stories

Sometimes, the team realizes some of the stories are too big to be estimated and tracked correctly. A user story needs to take somewhere between one and three days to complete, which means a certain granularity is required to be able to successfully manage the project work. Alternately, stories shouldn’t be too small because then they fail to deliver any relevant value and become insignificant. These are rather referred to as tasks—small activities that focus on the functionality and neglect the business benefit.

Big stories are known as epics, and there are two types: compound and complex.

- **Compound** user stories are user stories that contain several smaller, independent user stories inside them. These can be easily split because the user stories, being independent, are easy to spot. Typical examples here are different processes from a work flow.
• **Complex** user stories, on the other hand, are more difficult to split. In these situations, it is recommended to use techniques that reduce complexity. Instead of one big, complex functionality, value is delivered in small increments from the most simple to the most complex in iterations. Beginning with the most simple stories is not only easier to understand and implement, but also allows the customer and team to reflect on how useful the next level of complexity really is. It becomes possible to stop and deliver a feature that is good enough for the customer and avoid overprocessing or overproduction.

**Story-Splitting Techniques**

Some techniques for splitting large stories are—

- Process-based slicing
- Business-rule slicing
- Create, read, update, delete (CRUD)
- User or platform slicing
- Acceptance test slicing

What is important about any of these techniques is that the slicing should always be vertical. This means that any story resulting from the slicing should have a full functionality of its own, including all layers of product creation: architecture and design, graphics, development, testing, deployment, and so on.

**The Scrum Work Breakdown Process**

**The Scrum Work Breakdown Process**

The approach of continuously adding details while getting closer to a particular functionality is called progressive elaboration. We dismiss the details of the functionality scheduled to be developed later and add those details only when we are sure the functionality will be completed without any more changes. This allows changes to all functionalities that are scheduled to be developed in future sprints to be added to the backlog.

**Moving from Theme to Epic to Story**

In a previous discussion of the backlog, we introduced the concept of a “work item,” a generalized term referring to themes, epics, or user stories. The levels of work items used in a Scrum project (from bigger and more general to smaller and more specific) include—

- **Theme:** This is a very general expression of a certain functionality or idea that needs to be developed.
- **Epic:** Adding a few details to the theme will turn it into an epic.
- **User story:** Clear functionality, benefit, and users. This is the most detailed level that the product owner operates on.
Task: This is how the team chooses to implement a user story.

The product owner doesn’t control this breakdown process; rather, it is the team who decides. Examples of how to move from a theme to a story follow below:

**Work Breakdown Example 1**
Acceptance Criteria

Acceptance criteria are part of the definition of done of each user story. They are agreed upon with the team and reflect the level of work the product owner considers sufficient for a user story. Not respecting these criteria by under-delivering on them will cause dissatisfaction from the customer, while delivering more will cause the team to over-deliver, and spend time and money without real value.

This is the final C (confirmation) of the 3Cs of a user story and the T (testable) of INVEST. The criteria for passing or failing need to be very clear and nondebatable. Therefore, the definition of done needs to include both functional and nonfunctional requirements. While functional requirements are typically provided to the team by the product owner, the team needs to support the product owner in defining good acceptance criteria for the technical parts of the nonfunctional requirements.

The team needs to have the definition of done clarified at the beginning of the sprint because it becomes very expensive to deal with rework, defects, and under-delivery later. Additionally,
acceptance criteria have a great impact on estimating the stories, making a firm definition of done more critical.

Nonuser Story Considerations

Nonuser Story Considerations

Nonfunctional user stories, or nonuser stories, don’t offer immediate value or benefit to the customer but are mandatory for the system to work. Some examples include—

- Infrastructure
- Architecture
- Technology foundation
- Performance

Since these stories don’t bring immediate value to the customer but are necessary, they raise many questions regarding why they should be done, what happens if they are not done, or how much work should be put into them. The key to answering these questions is the acceptance criteria behind the stories. Unclear acceptance criteria may cause the team to spend a lot of time up front building a very advanced architecture that doesn’t have to be complicated. The traditional approach to architecture and design is that everything should be built from the beginning. The agile approach is to build everything, including architecture and foundation, in a very incremental way. Modern technologies allow for a more step-by-step approach, making it unnecessary to invest significant time at the beginning of the project.

Another important consideration is the effort put into these stories. Considering that the customer is not specifically asking for them (but expect them nonetheless), the best approach is to find ways to achieve results with a minimum of effort. Save effort (and sprints) for the stories that bring more value to the customers. Otherwise, we risk delivering low-level, basic functionalities while failing to deliver the big things the customer was expecting. In lean thinking, these stories are often called “necessary waste,” meaning they don’t bring value (waste) but are necessary to the system.

Sometimes, a spike can be considered a nonfunctional story if the issue is not blocking progress and allows work to continue on the sprint.
The Release Planning Meeting

The release planning meeting is the first real planning based on the product backlog, which means that dates, sprints, deadlines, and milestones will be considered. The output of these meetings is a timeline mapping major functionalities (expressed as themes, epics, or user stories) to a time frame that represents the project and its milestones.

A few things need to be clarified in the release planning meeting before actually starting the sprints, including—

- Definitions of done for the entire project, releases, and user stories
- Sprint length and number of sprints in the project
- Initial velocity based on previous experiences (to be discussed later)
- Hardening or prerelease sprints

Module Summary

Agile planning is based on the concept of progressive elaboration, which means not planning too far into the future and not adding too many details until the actual functionalities we need to include become clear enough to prevent major changes.

The product backlog is prioritized by business value for the customer; we always deliver the greatest value first. Some prioritization techniques include—

- MoSCoW (must have, should have, could have, won't have)
- Kano (need and satisfaction are considered when analyzing products or features)

If the backlog is too big for one team, then a Scrum of Scrums might be a solution. Recall this using multiple teams, each respecting the attributes of a Scrum team. We can use several Scrum masters to work with these teams, but there should be only one product owner.

In order for a user story to be correct, it needs to respect a few rules:

- 3Cs: card, conversation, confirmation
- INVEST (independent, negotiable, valued, estimable, small, testable)
• User story formula (As a USER TYPE, I want THIS FUNCTIONALITY in order to get THIS BENEFIT)

Sometimes, user stories are too big (complex or compound) and require splitting. Splitting techniques include process based, business rule based, CRUD, and acceptance criteria.

Acceptance criteria are details added to a user story to decide when the story is done. They can be functional and nonfunctional (technical) needs that are agreed upon between the product owner and the team.
Sprint Planning and Backlog Grooming

A common misconception remains that agile practices and practitioners don’t plan; however, the truth is that agile projects plan even more than traditional waterfall projects. Agile projects don’t put all the planning up front because they rely on relevant project experience that can be used for planning.

The only formal planning process, or ceremony, of Scrum is sprint planning. The product owner may also perform other types of planning not part of the official Scrum framework. He or she can decide how to conduct that or who to involve. An example of this planning is backlog grooming.

The Importance of Capacity in Sprint Planning

A valuable input to the sprint planning meeting is the team’s capacity, a subjective measure of the team’s ability to deliver. Capacity has a direct impact on a team’s velocity because the more capacity a team has, the more functionality it can build. Although that sounds like common sense, it is valid only until the team reaches eight-to-nine people. Experience shows that adding more people to a team of nine people not only will not increase production, but it will actually reduce it. The additional complexity of working with more than nine people affects its productivity.

Elements influencing team capacity include—

- The number of people
- Availability (including holidays and vacations)
- The skill level
- Levels and types of interruptions and multitasking

Once a team’s capacity has been decided, it is very important that the team maintains this capacity in order to maintain a sustainable pace. Only maintaining constant capacity allows us
to see if a team’s productivity increases from sprint to sprint. Fluctuations in capacity make it difficult to track improvements and plan subsequent sprints with any accuracy.

Sprint Planning Cycle

Sprint Planning Cycle

Sprint planning is a team exercise. Agile teams work together with the product owner and mutually determine the features to deliver during a sprint. This typically happens in a facilitated work session with a whiteboard, some flip charts, and a stack of index cards or Post-it® Notes. The team determines who will take responsibility for each feature, and it uncovers the risks, constraints, and assumptions for this iteration of the product. The team then estimates the resources and effort required to deliver.

In a sprint planning session, the team reviews the user stories it collected to glean the features that the customer expects. This is a dynamic list, with stories recorded on Post-it® Notes or index cards that can be easily moved around and reallocated from sprint to sprint. As previously stated, the user story cards document both the features expected by the customer and the tests that will be run to validate their quality. In sprint planning, the technical details required to implement the features are discussed, and this is the time to dig beneath the user-visible features and analyze the technical underpinnings, such as network connectivity or bandwidth considerations, that are necessary to support the features expected.
Some agile proponents advocate including riskier elements in the early sprints as a sort of feasibility test to determine if these elements are achievable. This is called “fail fast.” If critical, but speculative, components of the design are not viable, it should be determined as early as possible. Other agile proponents promote the development of an architectural foundation first, sticking to the basic core of the product and then adding on additional features once the base components are proven and accepted. Still others propose starting with the features that have the most consensuses and are least likely to change dramatically to create a solid foundation on which to experiment. Obviously, budget, resource, and schedule constraints have a large influence here as well. The prioritization and selection of features can swiftly become an exercise in negotiation skills, as different features often have advocates who want to promote their early selection. As with traditional project methods, agile teams need to resist undue pressure to include specific features or overcommit to features in a sprint.

Estimating Story Size

Relative and Absolute Estimating

Absolute estimating uses ideal time (days, hours), which can only be applied to tasks because they are precise and assigned to specific individuals. At the user story level, on the other hand, we require relative estimating (that is, estimating stories in relation to one another) because user stories—

- Are not very exact
- Belong to the team
- Are a vertical slice of functionality that require different skills (and different people) to be done
- Allow the team some freedom to find better ways of implementing the story

For these reasons, estimating user stories require a scale that everyone can relate to and understand regardless of who is doing the tasks from the user story.

Relative estimating starts with setting a reference—something that everybody understands in terms of how much work is required to complete it. Later, everything else will be estimated based on a multiple of this reference.

While the precision of this approach is questionable, it is a very effective way of realizing how much work lies ahead. Precision is not as important here because the team requires only an idea of how much work there is to be done and a reference point to find ways to obtain the same results with less work. In fact, the team is encouraged to come up with ideas to deliver the results in a more timely and effective manner than planned.

The Purpose of Relative Story Sizing

Repetitive efforts, with little variation from project to project, have a distinct advantage. Since these efforts don’t vary much, once a team has done one or two, it has a pretty good idea what
it will take to do the next one. Innovative, experimental projects don’t have this advantage. Since agile teams are often inventing the product as they develop, they need to understand how to estimate in an uncertain environment.

Also essential is the ability to estimate by features rather than by activities. Remember that in agile projects, requirements, scoping, and planning are performed on an iterative basis. Rather than trying to estimate task durations for an entire project, only looking as far as the next sprint or two is needed. In the backlog prioritization and allocation process previously reviewed, the team analyzes in depth the technical components of each feature scheduled for the current sprint. This detailed review gives the agile team a tremendous edge; they’ve already worked together to understand what it will take to deliver each feature, how features fit together, and who is best suited to develop them. This exercise gives the team a big head start in estimating the effort required to deliver.

The time-box concept that is common to agile approaches has a major influence on our estimating techniques. Because each iteration is time-boxed and many agile projects drive toward a defined deadline, the variable is features, not schedule. Working toward predefined time-boxes and end dates enforces the discipline of focusing on essential features and of committing to a reasonable number and size of features per sprint. Because the time-box is not elastic, agile teams need to be quite judicious about the number of features they decide to include in a sprint. In some projects, it’s the budget that is inelastic, and this obviously will have a significant impact on the features chosen and prioritized as the team progresses.

To make the necessary decisions about which features to include in each sprint and the number of sprints expected, agile teams will typically develop a ROM estimate.

**Factors for Sizing Stories**

The size of a user story is influenced by several factors. Some of the most important are the following:

- Volume of work
- Complexity of the user story
- Risks involved
- Uncertainty
- Previous experience
- Acceptance criteria

The size of a use story is a compound indicator including all the above. The team needs to decide how much each of these will weight in the total estimating and how it can address them to deliver faster.

**Using Story Points**

Story points are the preferred way of estimating user stories for several reasons. A relative story point scale is much easier to agree upon by all team members. Unlike absolute estimates, which declare that “it will take me about 16.5 hours to install the server,” Estimates of story
points are based on relative size. Is user story A twice as big as user story B, or half the size? The absolute duration of each feature, or story point, is not relevant to this exercise. A feature that is assigned a 4 should be four times bigger than another feature assigned a 1. This is an experience-based estimating approach, as teams obviously need to have some familiarity with the required effort for different features in order to assign a number of story points to them. Teams using this approach typically find one of the smallest features in their backlog and assign it a 1 (or a small value), then assign relative values to other stories in relation to that one.

**Planning Poker®**

One estimating technique common in agile teams is Planning Poker®. This technique includes all members of the development team and uses a prepared deck of cards with numbers representing possible estimates written on them. As an example, a deck of cards with the numbers 1, 3, 5, 8, 13, and 20 points on them will be handed out to each team member. The product owner will read the user story under consideration out loud and will allocate a short question-and-answer period. In keeping with the agile theme, the time allotted for this exercise should be brief enough that team members do not get bogged down in trying to derive a perfect estimate.

Once questions are answered, each team member displays the card that represents his or her best estimate for the feature in question. It’s common for these to vary widely; in fact, that’s an advantage, as it allows the teams to explore the varying assumptions that drove differing team
members to their conclusion. It is this discussion period that drives the real value of this exercise. Team members must justify their estimates to the entire team, and this exposes many risks, concerns, and technical complexities that would have been undiscovered if the teams simply applied an estimated duration to each feature. This allows fresh views to be shared with the team and is very important in identifying new ways of doing things.

The process is repeated until the team can converge on a shared estimate. It’s not necessary to repeat the process ad infinitum until everyone comes up with the exact same number; the point is to derive an estimate that everyone can be comfortable with in a reasonable amount of time.

Planning Poker® is based on the Wide-Band Delphi estimation in which during the estimation session, the team will come to a consensus on the effort required. Wide-Band Delphi analysis, like Planning Poker®, is most valuable when it is accompanied by explanation and dialogue, but it often becomes just a series of guesses that are captured and calculated rather than a participatory team exercise. The informal, game-like features of Planning Poker® ensure that the value of discussion and collaboration is realized.

Numerous studies have shown that the core elements of Planning Poker® lead to better estimates. The technique of gathering estimates from experts with actual experience in implementation, discussing and justifying those estimates with a team of collaborators, and averaging the estimates all have been shown to improve accuracy. Planning Poker® is a fun estimating technique with this added advantage: It works.

Velocity

Using Team Velocity

In order for these estimating approaches to make sense, the agile development concept of velocity needs to be understood. Velocity is a measure of the team’s ability to get work done based on the number of story points it has been able to deliver in the past. A team that delivered 12 story points last sprint is assumed to be able to deliver 12 in the next one. While this is clearly not a scientific approach, in the dynamic world of agile development, it offers enough information to decide how many sprints to expect and how many features can prudently be committed to within each sprint.

So, just to reiterate for clarity:

- Features are defined by users or customers and consist of the customer’s nontechnical description of the functionality that the product will provide.
- Story points are the development team’s estimation of the relative size of each user story, rating one user story—which will be relatively easy and fast to deliver—with 1 story point and another user story—which it estimates will take four times as long to deliver—as 4 story points.
Velocity is the number of story points that a development team can achieve in a given amount of time. If a team delivered 21 story points last sprint, it is assumed it can deliver 21 the next sprint, until it discovers that the number is not accurate—velocity is a dynamic number that changes as more is learned about the team’s abilities and speed. Additionally, once the team gains experience, velocity is expected to grow.

**Estimating Initial Velocity**

Obviously, for newly formed teams that have never worked together, deriving velocity will be difficult, as there’s no history to go on. As the team progresses toward the ultimate product delivery, its velocity becomes clearer and clearer, and the ability to estimate using story points becomes more refined. Can the team really develop 12 story points consistently from sprint to sprint, or was that a fluke? Must the team commit to features more cautiously in upcoming sprints? Or, conversely, is experience indicating that the team can, in fact, deliver 15 or 20 story points now that it has a few sprints under its belt and has formed an efficient working unit? Agile proponents insist that by working together and building an understanding of each others’ abilities and attributes, teams become incrementally more efficient and can tackle larger numbers of story points as they go.

**Velocity and Points**

One tool that many agile teams use to track progress is the “burn-down chart,” illustrated in the figure below. The chart begins in the upper left of the chart, before any development work is completed, and as features are completed or “burned down,” the team tracks that progress with a simple line chart. In this example, the product owner is tracking a release, which contains a total of 200 story points that the team must “burn through.” A burn-down chart also can track the progress of a sprint. Instead of sprints on the y-axis, it would be days. As user stories are built and tested, the number of story points remaining is plotted and a line connecting each point is drawn. By keeping the burn-down chart current, product owners can offer their customers a quick, visual representation of the functionality that has been delivered and is left to be delivered, and provide clues regarding the team’s productivity. During the sprint, the team will self-organize to deliver the sprint objective so even if it uses a burn-down chart, this chart is for the team’s usage only. If shared with people outside the project, it could lead to interruptions and influences during the sprint. The sprint is the time-boxed period of time given to the team to implement the functionality from the user stories it committed to.
Velocity and the Release Plan

If a team knows the velocity, or number of story points it can deliver, and it knows the aggregate number of story points for the entire backlog, it can make an educated guess about the number of sprints it needs to develop the product, assuming that the product backlog is detailed enough to be estimated. And, if the team has agreed to a time-box for each sprint, it can now map these time-boxed sprints against the calendar and give its customers a rough idea of the schedule for delivering its product. Again, it’s critical to caution against assuming that this, or any other, estimation method can take the uncertainty out of product development. In fact, as agile team members know, changes and optimizations will occur throughout the project.

The Importance of Realistic Expectations

To better understand why realistic expectations are important, we must refer back to the Scrum pillars: transparency, inspection, and adaptation. It is very important to discover the reality, admit it, and do something to change it. Transparency will help teams identify how they can be more accurate when estimating. Uncovering situations like lack of experience, optimism, details, or motivation are sometimes the result of such explorations.

It is totally acceptable for the team to be wrong. It naturally and normally takes a team a few sprints to achieve its true velocity. Scrum is based on empirical processes and relies on the creation of experience to draw from. After a while, the team’s velocity will reach a plateau, which reflects its real velocity under normal conditions. What might happen though is that the product owner might need more and ask for more points. In that situation, the team will need to react
and find ways to deliver even more (technical excellence, innovation, creativity, and so on),
causing the velocity to grow even more.

**Advanced Estimating with Business Value**

**Advanced Estimating with Business Value**

The term “value” has become so used in today’s business language that it is hard to define what
it means exactly. However, when we refer to “value for the customer,” we generally mean that
we are delivering one or several of the following to them:

- Money: Helps make more money or saves some money
- Time: Saves time
- Satisfaction: Satisfies the customer, stakeholder, or shareholder
- Compliance to regulations

A very important technique for estimating value is the “value buckets” method. The first step
involves defining the value buckets, which are sometimes corporate priorities like customer
satisfaction or market share. Next, all the user stories from the backlog will be mapped into
these buckets. The result is very visual and shows how much effort is put into each bucket. If a
bucket is really important, then it should contain a lot of user stories. If it is not important and
doesn’t have enough stories, then it probably needs to be removed from consideration
completely. Estimating with value buckets is a great exercise to realize what your project is
actually delivering and if it is different from what was initially planned.

**Revisiting Sprint Length**

**Revisiting Sprint Length**

Agile teams must consider how far in advance they want to plan. Should they stay focused on
only the current sprint or should they plan for a number of sprints, or even an entire release?
The longer the planning period, the less flexibility to adjust to changes. The shorter the period,
the more able teams are to incorporate feedback. As noted earlier, some teams try to include
the riskiest elements in the early sprints to test their feasibility. In these sorts of projects in which
there is high uncertainty regarding the viability of key elements, and, therefore, the entire effort,
it doesn’t make much sense to spend time planning ahead until the basic feasibility questions
are answered.

For projects in which the ultimate viability is less in question, planning a sprint or two ahead
allows the customer to visualize a more complete picture of how the effort might unfold, allows
for more insight into dependencies, and offers opportunities for moving features between sprints
in order to allocate work more realistically. Finally, in some environments, especially those early in the migration from traditional to agile methods, a complete release calendar can ease the customer’s stress of uncertainty regarding the length and effort of the entire project. Release-length planning is fine as long as it doesn’t undermine the iterative nature of the agile approach and doesn’t mislead the customer into thinking that we can accurately predict work far out into the future.

No matter how far into the future sprints are planned, the same uncertainties, risks, and contingencies apply, and customers are still free to change their conception of the product as the project progresses. Product owners must maintain a fine balance between planning far enough into future sprints to sustain a holistic vision of the entire program while avoiding the error of predicting or estimating events that are still undefined and conjectural.

**Backlog Grooming**

**Conducting Backlog Grooming**

After reviewing all the details of planning for, and delivering, a sprint, it’s useful to step back and take a broader view. While a team is concentrating on planning and delivering a sprint, the ultimate goal of a releasable product as the culmination of its efforts may seem abstract and remote. In software development, for example, talented developers will get so deeply immersed in the technical challenge at hand that the ultimate business function can recede. The product/release/sprint structure of incremental project development gives teams an interim, a major, and a final goal toward which to drive. By returning to release planning after each sprint in order to incorporate the changes and experiences captured, the team gets a chance to return to the bigger picture and to the customer-oriented language of the release level.

Each return to release-level planning gives the team a chance to check the evolution of its thinking on the project at hand. Do the technical elements still seem feasible? Has the team learned anything about its velocity that should prompt it to revisit its sprint or release plans? Can the team still deliver a working product or prototype at the conclusion of this release? Has the team found risks, constraints, or errors that need to be analyzed? What has the reality of development revealed that must be taken into account for the project to succeed? Agile offers the advantage of a built-in review and course correction as the project progresses—it would be a waste not to use the opportunity to review our strategy and tactics.

The heart of the release plan is the total features list, or backlog. This catalog of our commitments is the central artifact of the release. It is not, however, a strategy. Release planning is the forum in which teams should raise their sights to the more strategic level and be sure they understand what this release will show to the customer, and how that product fits into the development picture for the final product. Teams should discuss the success criteria for the release and be sure that there’s still consensus on the value of both the release and the product.

This is not to say that the end of each sprint should be a complete revisiting of the project’s
strategy. In most cases, there will be an unspoken agreement within the team that everyone understands and accepts the value of the project. In agile projects, however, openness to change and improvement means that there is a real possibility that the original conception of the project has mutated. The team and the PM must be aware of this possibility and be sensitive to any clues that this has occurred.

The obvious advantage to incremental, iterative project methods is the opportunity to honestly check the status, progress, commitment, and productivity, and to fine-tune both the product and the process as the project progresses. The product/release/sprint framework presents agile teams and PMs with a structured, logical approach to innovation and inventiveness that can adapt while it creates. Teams that use the iterative approach to bring knowledge, experience, and reality into their project plans will reap the benefits of agile project management.

Backlog Grooming Considerations

Backlog grooming is important because this is how we ensure we are still on track with what the customer expects from the project. Backlog grooming sessions are product owner driven and focused on—

- Adapting the backlog based on the latest feedback from the customer
- Making sure it is detailed enough for the team to work on in the next sprint or sprints

Grooming is performed both by adding more stories to the backlog as a result of feedback from the customer and also by reprioritizing the backlog so it still reflects both the reality of the project and the customer’s priorities. The backlog should be considered as a living document that is constantly updated to remain aligned with new requirements and expectations from the customer.

The result of a backlog grooming session is an updated product backlog and a detailed set of stories to be undertaken in the next sprint. The identified stories will now be discussed in the next sprint planning session and will be given to the team for implementation.

Module Summary

Sprint planning is the ceremony where the product owner explains to the team what (the WHAT) should be implemented in the next sprint, and the team decides how they will do it (the HOW). The output of this meeting is the sprint backlog.

Stories that go into the sprint are estimated using story points, an estimating technique based on relative estimating. Each user story is estimated in story points, which is a measure of work, complexity, risk, and previous experience. The number of points a team delivers in a sprint is called velocity. This is the team’s rate of burning down the product backlog. The team needs to constantly find new ways to increase its velocity while maintaining a sustainable pace.
The product owner is responsible to keep the backlog up-to-date by adding new stories and reprioritizing existing ones. This is done in the backlog grooming session where the product owner prepares the backlog for future sprints.
Module Overview

The Daily Stand-Up

Agile teams are empowered and self-organize to deliver the best possible results. Ideally, this means there is no external influence, especially during the sprint, to monitor or control them. In spite of this lack of external control, however, all activities still require some form of monitoring to make sure the team is making progress toward its goals.

The Scrum framework acknowledges this need and uses Scrum ceremonies as a means of monitoring work. The daily Scrum is the most efficient method because of its granularity and transparency. As the name implies, the meeting occurs daily, which creates task-level visibility and addresses all user stories that are open. The format of the Scrum allows each team member to describe his or her progress, immediate plans, and issues that might block continued progress. And, all in under 15 minutes!

The Daily Scrum

What Is a Daily Scrum?

In most agile approaches, teams use some variant of a daily meeting to track the daily issues, risks, and challenges that arise. These sessions often are called “stand-up” meetings because agile proponents believe that by standing rather than sitting around a conference table, teams are encouraged to keep these meetings short, concentrated, and effective, which helps to avoid the plague of irrelevant and endless meetings that can distract from forward progress. In the Scrum method, this session is the Scrum, named for the player’s huddle in rugby.

Why Do a Daily Scrum?

This meeting is the forum for discussing the tasks that slipped or were completed early and the unexpected barriers or challenges that arose in the previous day of development, and for refining plans on a daily basis depending on the realities of this particular effort and team. Features and tasks can be swapped by team members as they discover their own suitability for the tasks they’ve accepted. Schedules can be refined as the team’s velocity becomes clearer. This daily team meeting often is referred to as the heartbeat of the agile methodology. The
meeting serves as a reporting tool to inform colleagues about progress and also as a planning tool for the very near future (next day). It is essential for teams to remain synchronized and limit the inefficiencies that arise from working as a team.

**How to Do a Daily Scrum**

In the daily Scrum, the team stands together and, in round-robin fashion, answers three simple questions:

- What have you done since the last Scrum?
- What will you do until the next Scrum?
- What issues, risks, or barriers have arisen that could distract you from accomplishing the sprint goals?

It is important to note that the Scrum is a team meeting meant to provide value primarily to the team. The product owner’s presence is optional and only if the team feels comfortable with it. The Scrum master is invisible and only intervenes if things go off course. Otherwise, the Scrum master only observes and take notes. If there are any issues, then he or she will follow up to make sure they are resolved.

Other agile methods have their variants of the daily meeting, but all are focused on planning and communication at the granular, daily level. Some standard characteristics of a daily meeting are: Each meeting is 15–20 minutes; everyone stands in a circle; each meeting occurs at the same location; the order of presentation is defined; team members share their status/obstacles; and all team members are invited, but only people involved in the iteration speak at the meeting.

**Module Summary**

The daily Scrum is a (maximum) 15-minute meeting in which every member of the team answers three questions:

- What have you done since the last Scrum?
- What will you do until the next Scrum?
- What problems distract you from accomplishing the sprint goals?
Sprint Review and Retrospective

Module Overview

Sprint Review and Retrospective

Scrum is based on the concept of continuous improvement, not only with regards to the product, but also to the team and its functioning. The team needs to improve its efficiency and productivity while delivering value for the customer. In other words, the Scrum process not only focuses on delivering value to the customer through product innovations, but also through innovations, improvements, and efficiencies in the process.

There are two major ceremonies that provide opportunities to address continuous improvement. The sprint review is concerned with receiving feedback on and improving the product. The team achieves this objective by performing a demo of the product completed in the previous sprint. It is an important point that the major focus of this ceremony is not to attain product acceptance; rather, it is to get more work by focusing on ways the product can be improved. As a result of the review, the team generates the change requests needed to evolve the product.

The sprint retrospective is a ceremony specially focused on the improvement of the team's performance. Relying on the Scrum values of commitment, focus, openness, courage, and respect, the team should attempt to identify all potential problems with its performance and address them. Even on a highly functioning team, there are opportunities to innovate in an attempt to achieve even more effective outcomes and deliver greater value.

The Sprint Review

What Is a Sprint Review?

The mechanism for integrating change into an agile project under development is the sprint review, performed at the end of every sprint.
Why Do a Sprint Review?

The purpose of the sprint review is to get feedback on what has been achieved and generate ideas that will be captured in the product backlog, and later implemented according to the priority they receive.

Borrowing from the rugby analogy that runs through the Scrum methodology, a sprint signifies an energetic, high-speed race to the goal line, a definition that fits the agile iteration as well as it does the athletic analogy. Developers are investing all of their energy into delivering on the commitments made to the customer, and Scrum masters are focused on ensuring collaboration, encouraging creativity and efficiency, and removing any barriers that would impede progress. The result of all this concentrated effort, hopefully, will be a quality product that exceeds customer expectations. It is also true that the likely result will be an exhausted agile team that needs a pause to refresh.

This pause in the action allows agile teams to reset and view their interim creation from a more objective vantage point. It enables development teams to collaborate with their customers to review their creation; ensure that the value expected is being delivered; and encourage the creative injection of new ideas, approaches, and solutions in order to create a more perfect outcome. These reviews should be scheduled as part of the iterative project cycle and should include reviews of both the content and the process. This means analyzing the result created and the process used to achieve the result in order to optimize both.

How to Do a Sprint Review

Obviously, the customer review of the product’s functionality is key. By using customer focus groups, in which selected members of the development team and the additional members of the customer team meet in a facilitated session to analyze and evaluate the interim deliverables, agile enterprises can generate new ideas and develop strong collaborative bonds at the same
time. These work sessions typically include a functional walk-through of the product and grant development teams an opportunity to assess the customer’s reaction to their work.

### Functional Walk-Through

Elements of the product, from the subjective look-and-feel through quantifiable performance metrics, are reviewed and discussed by the entire assessment team, and any recommended changes, objections, or new ideas are recorded for further discussion. In keeping with the agile philosophy, these reviews focus on the actual, delivered functionality, not on the manuals, documents, or predictions of future deliverables. Detailed discussions and problem solving regarding changes and new features often are addressed in a second, separate session; trying to analyze and solve these questions in one session can lead to technical digressions over feasibility and to emotional detours, as proponents argue over their fondest ideas. The integration of recommended changes must be managed and controlled in order to sustain the balance between agility and discipline, and these changes and new features must be subjected to a technical impact review to understand their feasibility and effect on the entire effort.
Recommend Product and Process Changes

This is one of the reasons that many agile teams also include a technical review in their adaptive review process. A technical review is important to ensure that technical quality and robustness is not being sacrificed to expediency or poor practices. While experienced agile teams often perform technical reviews as they progress, a formal pause to consider the technical status between iterations is good practice, as it allows developers to step back from their pride of ownership and see their technical work holistically, as part of the delivered product. It also allows them to absorb the comments and, perhaps, criticisms that arose in the customer review process and to think through their implications.

This is the forum for the team to discuss the technical feasibility and difficulty of incorporating the changes requested by their sponsors and to begin thinking about their potential impact to the time, scope, and cost to the product effort. The outcome of this session is typically a list of changes documented from the customer focus group, along with the technical team’s assessment of the impact and feasibility of these changes, for a presentation to the customer’s chief product manager or sponsor. The product owner and development team will then typically determine the changes that will be integrated into the next iteration, which will fit somewhere within the release and which may be deferred until a later revision or not incorporated at all. It’s crucial, in the spirit of collaboration, that these change acceptance decisions be made with the customer representative and not by the technical team or product owner alone.
What Is a Sprint Retrospective?

The sprint retrospective is a simple exercise that gives teams a chance to assess which parts of the sprint went well, which can be improved, and where gaps in their understanding or expertise might lie.

Why Do a Sprint Retrospective?

In conjunction with the sprint review or conducted as a separate exercise, the retrospective provides a time to review the team’s working practices and processes to ensure that they are as efficient as possible.
How to Do a Sprint Retrospective

This is emphatically not a groaning session in which unstructured complaints and personal gripes are surfaced, but rather a structured, facilitated exercise in self-governance and self-improvement. Substantive issues, such as a team member’s lack of participation in meetings or a Scrum master’s difficulty in adapting to agile techniques, should be raised, but personality issues are not welcome in this forum. This is the forum for determining if, from the team’s perspective, sprints are too long or too short, if assignments are unrealistic based on experience in this iteration, or if skills need to be augmented with new resources.

The agile environment should be adaptive but not chaotic. The ideas, innovations, and solutions that emerge from these interactions are not uncritically accepted and implemented just because someone suggested them, and they are not rejected simply because the development team thinks they’re “too difficult.” Agile teams apply a balance of adaptiveness and discipline.

Module Summary

The sprint review is the ceremony when the team shares publicly what it achieved in the last sprint, carrying out a demo with the aim of getting feedback from the customer. This feedback is then incorporated into the product backlog and prioritized.

The sprint retrospective is the ceremony when the team distances itself from the product and focuses on itself as a team. Common questions here are about what the team did well and what could be done differently. The ceremony is not an opportunity to lay blame; rather, it should be
an opportunity to consider all ideas and collaboratively problem solve about how to become more efficient.