THE EPIC GUIDE to AGILE

MORE BUSINESS VALUE ON A PREDICTABLE SCHEDULE WITH SCRUM

DAVE TODARO

THE EPIC GUIDE to AGILE

MORE BUSINESS VALUE ON A PREDICTABLE SCHEDULE WITH SCRUM

Tired of out-of-touch Scrum training that doesn't work? Discover practical agile delivery techniques to make your software shine.

Has your excitement over Scrum led to nothing but disappointment? Have months of agile training still left your company far short of optimal efficiency? Do you feel like your leaders and developers are speaking a completely different language? Ascendle CEO Dave Todaro has lived and breathed software development for over three decades. After running successful agile practices on a daily basis, he's ready to share his insights and techniques to help your company reap the benefits of his experience.

The Epic Guide to Agile: More Business Value on a Predictable Schedule with Scrum is a comprehensive guide to software-based team dynamics that both leaders and developers can understand. Unlike most agile training that doesn't work in practice, Todaro's step-by-step playbook rises above theory to save you time and money. Perfect for any sized business or level of experience, you'll get to the crux of each Scrum issue to have your team running sprints more efficiently than ever.

In *The Epic Guide to Agile*, you'll discover:
- Personal examples and anecdotes to tackle problems at their source
- Effective ways to introduce agile and Scrum into your organization with the right pilot team
- The exact blow-by-blow system to achieve productive sprint planning sessions
- The typical issues that can doom your design and how to conquer them
- The best technical environment setups to support your software project groups and much, much, more!

The Epic Guide to Agile is a powerhouse manual to help any ScrumMaster or Project Manager find productivity and success. If you like real-world examples, no-nonsense teaching, and clear communication, then you'll love Dave Todaro's extraordinary and practical guidebook.

Dave Todaro

Programming since age 11 and shipping his first commercial application at 15, Dave has lived and breathed software development for the past 35+ years.

Dave is founder and CEO of Ascendle, a Boston-based firm that leverages Microsoft technologies to build web and mobile apps for the world's leading companies.

US $49.95 *For bonus content, visit:* **davetodarobooks.com**

THE EPIC GUIDE TO AGILE

MORE BUSINESS VALUE ON A
PREDICTABLE SCHEDULE WITH SCRUM

DAVE TODARO

Published by:

R9 Publishing LLC
PO Box 232
North Hampton, NH 03862

Copy Editor: Noleen Arendse
Content Editor: Wayne Arendse
Proofreaders: Steven Codd, Diana Getman, Allison Grappone, Susannah Parnin Mitchell, Tom Schwendler
Marketing: Sara Janjigian Trifiro
Interior Design: Dave Todaro
Cover Design: Global Deziners

Paperback ISBN: 978-1-7330004-0-6

Digital ISBN: 978-1-7330004-1-3

Contents at a Glance

Table of Contents

Preface

An expert is a person who has found out by his own painful experience all the mistakes that one can make in a very narrow field.

- ***Neils Bohr**, Danish physicist.*

Years ago, I was in charge of developing a new product at my software company. After about 10 months of discussions, research, and focus groups, we had identified a new business problem to solve for our customers, and we decided to spin up a development team to get it built.

About a year and a half later, we were nearing the release of version 1.0. The only problem was we were running about six to seven months behind my original schedule estimate, and we had started pulling developers off other projects to drive it to done. Everyone was working crazy hours to get the thing out the door, and we were all ready to be finished.

As we neared the release, I sat down for a meeting with the CEO and the lead sales rep to give them a status update. From my perspective, I felt that although it was later than I originally thought, the project was a huge success. We had worked *really* hard, with everyone putting everything else second for months—including their families. I came into the meeting with a smile on my face.

That didn't last long.

As I sat down, I looked over to the CEO—who was also my business partner—and he said, "You've failed."

It didn't really sink in at first. Feeling my face start to get hot I said, "What the hell do you mean? We busted our asses to get this thing done and it's about a week away from shipping. Customers are psyched to get their hands on it."

"Yeah, but you're *seven months late!* We were planning on having money from that product in the bank. You've put us in a tough spot."

After more than a year and a half of putting all I had into this new product—at the expense of everything else in my life—I was exhausted. In hindsight, I was probably on the edge of a nervous breakdown.

The meeting quickly went downhill from there, evolving into a yelling match with plenty of finger-pointing, and ending with me storming out of the conference room and leaving the office for the day.

It was one of the worst days of my life.

Product Development Is About Business Value

My team and I had *finished the product*, which is a feat given that over half of IT projects fail (Florentine, 2016). Everyone agreed it was a great piece of software.

I had done a road tour to a handful of customers during the previous month and they were all excited to get their hands on what we had created. How could I have possibly *failed*?

Once I had time to reflect, I realized two things:

- Software development—unless in an academic or research context—isn't about technology, it's about delivering business value. In this case, that business value was in the form of a new product to drive higher revenue from our customers.
- Delivering business value doesn't matter if it's not on a *predictable schedule*. Otherwise, it's impossible for the business to create plans to maximize that value. "When will I get this revenue?" is a critical question to answer.

The problem wasn't the product—customers loved it. It was that I had spent too much time and money getting it done. The company needed to get business value faster than my team and I provided.

Even though I had started to use an agile Scrum framework a year or two before starting this product, I later realized that I didn't really know what I was doing. I was using the right terminology and having the right meetings, but I wasn't properly leveraging Scrum to ship the product on a schedule.

After a lot of soul-searching, I finally admitted to myself that I had, in fact, failed.

Scrum Done Right Delivers Results

My experience with that project—now close to ten years ago—was a pivotal moment in my life. A year or two later, I decided to sell my business interest to my partner and start a new company, which I called Ascendle.

I had one goal: leverage all my hard-won experience to provide world-class, "turnkey" agile development teams to enterprise clients.

These self-contained teams could be free to innovate at an unprecedented pace, leveraging the best of today's tools and technologies.

Everett Rogers, in his book *Diffusion of Innovations*, described *skunkworks* to mean "an especially enriched environment that is designed to help a small group of individuals escape usual organizational procedures, so that innovation is encouraged." (Rogers, 1995)

This is exactly what we provide—an on-demand skunkworks—and we deliver results daily at a blistering pace our clients have never experienced.

Why I Wrote This Book

Software development is *hard*. I've spent more than three quarters of my life figuring out how to make it easier. And I've finally arrived at a specific set of tools and techniques to deliver on that promise.

It's still a lot of work to ship a world-class product, but using the information in this book will make your life a hell of a lot more enjoyable.

Over the last 35+ years, I've made three key discoveries:

- Leveraging an adaptable and flexible framework such as Scrum is critical. No two organizations are the same, so prescriptive, one-size-fits-all "methodologies" don't work.
- Being *fanatical* about disciplined adherence to the rules—while constantly innovating within their boundaries—is a key success factor.
- Attempting to use agile techniques without the appropriate technical underpinnings is a recipe for failure.

I tried to find one book that included a complete "how-to" guide for both Scrum *and* the required technical tools and techniques. As I searched through my own library and browsed books on Amazon, I couldn't find one.

Most focus on the fundamental details of Scrum, including forming a team and running the day-to-day process. Other books have a narrow focus on a small portion of the process, such as testing or how to scale Scrum in larger organizations or include only lightweight technical details.

None that I found included the right level of detail about *both* the agile process *and* technology.

What You'll Get Out of This Book

This book is a comprehensive cheat sheet, saving you the years of heartache and frustration you'd experience if you were forced to learn everything through trial and error. It has all you need, from an overview of Scrum and why it works, to practical tips to create and guide your Scrum team to deliver at a world-class level.

Most important, it addresses the technical foundation required for success. It has explanations of software construction and the technical tools and techniques your team needs to deliver rock-solid software consistently and predictably.

However, it's not written for a technical audience. Everything is spelled out in plain English, so you'll be able to understand it even if you're not technical.

This book will empower you to work with your entire team—project managers, business analysts, software architects, developers, and testers alike—and understand what they need to do in order to achieve your company's business goals.

How This Book Is Organized

This book is organized into four parts.

- **Part I, Understanding Scrum Fundamentals** provides an introduction and high-level overview of the Scrum framework and its associated roles, ceremonies, and artifacts. If you are well-versed in agile techniques, you may recognize a lot of the content here, but I encourage you to read this part to refresh your memory of the building blocks of Scrum.

- **Part II, Understanding Agile Software Construction** introduces the fundamental technical concepts required to support the agile process. If you are non-technical, don't be intimidated; I wrote this section with you in mind. It's critical for everyone involved with agile to understand these foundational concepts.

- **Part III, Laying the Groundwork** outlines the steps required to set the stage for launching an agile process in your organization. From forming your first Scrum team to creating a concise vision and initial schedule estimate, the steps in this part will set your team up for success.

- **Part IV, Building the Product** walks you through the steps to launch and run iterations for the Scrum team. Starting with the all-important process of creating a plan for each sprint via the sprint planning ceremony, and including details about tackling day-to-day challenges, this is where vision turns into reality by producing a shippable product increment. This part ends by describing the process to smoothly get the product into production so you can put it into the hands of your end users.

Although each part builds upon those that come before it, feel free to jump around to different chapters of the book as you see fit.

Each part, chapter, and section is written in a fairly standalone fashion, so you can get details about the topics that interest you the most. At the end of each chapter you'll find a *Key Takeaways* section, summarizing its main points.

Final Thoughts

Agile transformed my life. Yelling matches with business stakeholders are a thing of the past, and our clients are amazed by how we leverage Scrum to deliver on a schedule while allowing them to retain full control over the priorities of the team.

I'm excited to share my hard-won experience with you, so you can experience the same skunkworks-driven results that delight our clients each and every day.

Thank you for reading and enjoy your journey!

Dave Todaro
North Hampton, New Hampshire
Spring, 2019

If you have questions, see any errors, or would like to provide suggestions for new books or a second edition of this book, I'd love to hear from you!

The best way to contact me is via LinkedIn: http://bit.ly/davetodarolinkedin

For bonus materials and to receive updates about new editions of this book, sign up at https://www.davetodarobooks.com

Part I
Understanding Scrum Fundamentals

The business we're in is more sociological than technological, more dependent on workers' abilities to communicate with each other than their abilities to communicate with machines.

- ***Tom DeMarco**, Peopleware: Productive Projects and Teams*

Chapter 1

Introduction to High-Performance Teamwork

Never confuse movement with action.

- Ernest Hemingway

Have you ever been frustrated by the work of your software development team? Do you sometimes feel like they aren't working on the most important things? Or they can never give you an accurate estimate of how long it will take? Or they're not communicating very well?

Do you feel like the quality of their work could be better? Do they demonstrate ownership and accountability? Does it seem like they could be happier?

You're not alone. Almost everyone I've talked to about software development in the past 35 years has had at least some level of frustration. I've certainly experienced everything outlined above with my own software teams prior to really learning how to "do agile right."

1.1 Creating Dream Teams

Some people have the good fortune to stumble upon high-performing "dream teams." They hire perfect people, assemble them into teams, and enjoy the astounding results they produce with little to no training or management.

The team members "gel" almost immediately and exceed all expectations. Their communication skills make it seem like they can read the minds of product managers, executives, and each other. They somehow magically produce software applications that everyone loves. They also frequently review their process and consistently improve it to deliver even better results. They are a *machine*.

Although it's possible, what I outlined above seldom happens in the real world. Some team members will be great communicators. Others will have amazing technical skills. Some will always stay focused and get things done. But seldom will everyone on the team have the discipline to do all the right things—at the right time—every day. Very rarely will a high-performing team materialize by chance.

It seems like it should be pretty simple. As a business leader I want the following from my development team:

- Have them work on what delivers the highest value to the business.
- Get an accurate schedule forecast I can rely on.
- See them take ownership and be accountable.
- Get a high-quality product that users love.

To achieve the above, we need a way to help teams become…teams. We need a way to ensure they are always working on producing what's most important to the company and its customers. We need to help them learn how to understand business goals, effectively communicate with each other, and coordinate their work in a way that drives the best results.

We also need a way to predict how long it will take to complete their work, and ensure they consistently produce the same high-quality results. Finally, we need a way for them to improve over time so they can keep getting better at what they do.

We need a *system* to ensure that groups of individuals employ the same proven practices which allow the "dream teams" to produce their amazing results. The system must be simple and flexible. It must empower business leaders to drive priorities and help teams produce results consistently, reliably, and predictably. In short, we need a system that facilitates and creates high-performance teamwork.

Scrum is such a system.

A lightweight framework consisting of three roles, four meetings, and three artifacts, Scrum is a flexible and adaptable tool that allows a group of individuals to become a high-performance team.

Scrum provides a way for business leaders to drive priorities and have a development team produce results consistently, reliably, and predictably—which to me is the definition of "high-performance teamwork."

1.2 Why Does Scrum Work?

Scrum is what I call a *social engineering framework*. By imposing a small set of rules, it allows a group of normal individuals to do the right things, at the right time, to produce extraordinary results.

Work is completed in fixed-duration production cycles called "sprints," sometimes also called "iterations." This time period—most commonly two weeks, but ranging from one week to a calendar month—turns the traditional long-term deadline on its head and forces the Scrum team to produce results *now*. This imminent deadline creates a heightened sense of awareness that drives the team's focus and effort.

Leonard Bernstein—a composer and one of the most successful musicians in American history—once said, "To achieve great things, two things are needed: a plan, and not quite enough time." (Classic FM, n.d.) This is exactly what Scrum provides, with its imminent deadline at the end of each sprint.

Scrum Forces a Team to Make a Plan

At the beginning of each sprint, the Scrum team works together to create a plan, breaking down items into individual subtasks. Each subtask represents a discrete step required to complete the work.

This forces them to think strategically about what they need to do, leading to increased productivity during the sprint because they can keep their heads down and focus on completing the work.

Scrum Doesn't Give a Team Quite Enough Time

Each fixed-duration sprint helps prevent the development team from working on anything that's not critical to driving things to "done" as quickly as possible.

This helps avoid time-consuming but low-value activities, and also helps avoid over-building the product—so-called "gold plating."

Scrum Forces a Team to Communicate

Many technical people are not natural communicators, and most companies don't know how to create a framework that drives effective collaboration. Scrum solves this by creating an environment that requires communication on a regular basis.

From a 15-minute daily team meeting to coordinate their work, to an emphasis on personal interaction over comprehensive written specifications, Scrum gets team members communicating much more than they might without such a framework.

Scrum Forces a Team to Be Held Accountable

When I talk to business leaders and ask them what they want most, I typically hear, "I'd like my team to have more ownership and accountability." Scrum delivers on this promise by empowering the team to decide on what they will commit to completing, and putting them in front of business stakeholders at the end of the sprint to report on their results.

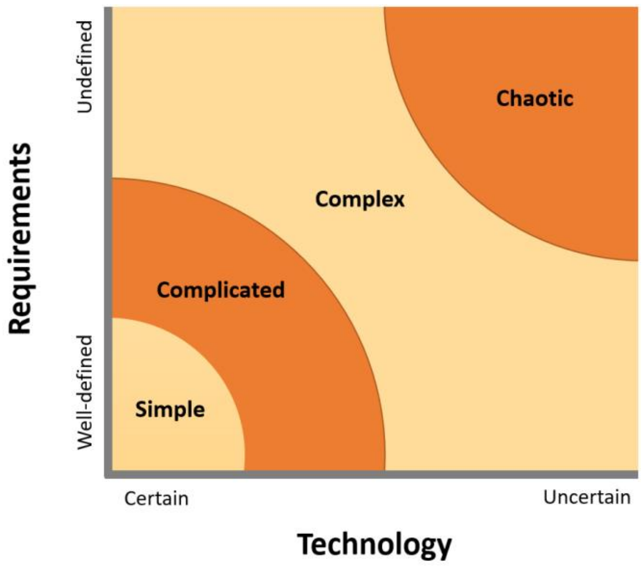
Everyone knows the "higher-ups" will be reviewing the results of their work and holding them accountable, which results in a strong feeling of ownership and a, "We really need to get this done" attitude.

1.3 Is Scrum Just for Software Development?

Scrum was created in the early 1990s by Jeff Sutherland and Ken Schwaber, to help organizations struggling with complex software development projects. According to them, Scrum is, "A framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value." (Schwaber & Sutherland, 2017)

Scrum is especially helpful for managing projects that have the following characteristics (Lenfle & Loch, 2010):

- *Emergent requirements.* There is a general vision or direction, but the detailed goals of the project are not 100 percent known and are at least partially emergent. That is, they become clearer during the course of the project.
- *Capabilities don't necessarily exist.* How exactly the project will be completed is not known at the beginning, and some research and invention will be required along the way.
- *Unforeseeable uncertainty.* There are risks and other "gotchas" that will come up during the project that can't be anticipated at the start of the project.



Scrum is a good fit for complicated and complex projects. (Schwaber & Beedle, 2001)

Scrum isn't for every project. If you're building a new office building using traditional construction techniques, you probably don't need this type of adaptive framework.

However, if you have requirements that are likely to evolve over the course of the project, and at least some technical unknowns about how exactly the work will be completed, Scrum will likely make your life a heck of a lot easier.

Software development projects are a particularly good fit for Scrum. This is one reason it continues to become more popular as software development continues to grow at a rapid pace. As Marc Andreessen would say, "In short, software is eating the world." (Andreessen, 2011)

Although Scrum has its origins in software development, it's been used to successfully manage many types of projects. From software and hardware to autonomous vehicles, schools, and marketing, Scrum has been used to manage a variety of complex work projects. (Schwaber & Sutherland, 2017)

In short, Scrum is a good fit for *complicated* or *complex* projects. The diagram illustrates these types of projects, defined by a degree of uncertainty about requirements and a degree of uncertainty about technology—or how exactly the project will get done.

1.4 How Agile Differs from a Traditional Waterfall Approach

A few years ago, I was approached by the CEO of a small software company in Massachusetts which specialized in a vertical market software application for the construction industry. After five years of building their software application, the technical co-founder and sole programmer had decided to leave the company, and they were in a bind.

The product he had built was working fairly well, but it was a traditional client-server application that required installation on-site at each of their customer locations. Plus, it included a custom mobile application based on legacy, outdated hardware.

Maintaining all those on-site systems and finicky mobile devices was becoming a hassle. Plus, the product had some fundamental design flaws, and was buggy and slow.

Their strategy for the future involved moving the application to the cloud and utilizing contemporary Android-based industrial mobile devices. That would eliminate the on-site deployment hassles, allow the use of more modern mobile hardware, and would give them an opportunity to take a "clean-sheet approach" to build a fresh, new replacement product.

I stepped in as a temporary "virtual" Chief Technology Officer and helped stabilize the current product while assembling a development team to build the new product. The company didn't have much money, so the team was tiny, with just two half-time developers and a tester working about ten hours a week.

Because time was very limited, using an agile approach was critical. We prioritized ruthlessly and drove new features to “done” each sprint, with demonstrations to company stakeholders every two weeks.

We released version 1.0 two months ahead of schedule when they deemed some of the features on the list could wait until after the initial version.

To put this in perspective, we had replaced a product that took *five years* to build in *less than seven months*, with the same resources—one full-time equivalent developer—and we *shipped two months early*. This was in stark contrast to my past experience outlined in the Preface.

This was possible *only* because we used an agile approach as opposed to a waterfall approach.

The Waterfall Approach

A plan for a traditional waterfall-based software development project often looks something like this at the beginning:

- Perform customer research: 3 mo.
- Write thorough requirements specification: 3 mo.
- Create detailed technical design: 2 mo.
- Implement technical framework: 3 mo.
- Build product functionality: 6 mo.
- Perform user acceptance testing and fix bugs: 3 mo.
- Deploy to production: 1 mo.

Total: 21 months

There are two big problems with this type of plan:

- By the time the team reaches the alpha and beta testing stage, there’s really no way to know how long it will take to fix all the things that were broken along the way.
- When the team reaches the two-year mark—assuming it didn’t grow to three years—what customers wanted at the beginning of the project is no longer what they need. Business conditions have changed.

Most 21-month waterfall software projects do not get done in 21 months, and they seldom meet the needs of users in the way everyone envisioned up front.

The Agile Approach

In contrast, a typical agile-based approach to build the same product might look something like this:

- Perform customer research: 1 mo.
- Write user stories: 2 wks.
- Create detailed technical design: (during each sprint)
- Implement technical framework: (during each sprint)
- Build product functionality: every two-week sprint; reaching "minimum viable product" state after 10 sprints, or 20 weeks.
- Perform user acceptance testing and fix bugs: (during each sprint)
- Deploy to production: 2 wks.

Total: less than 7 months

Just to be clear, I am in fact saying that by using Scrum, the agile approach shaves approximately *14 months* off the schedule.

How Agile Accelerates Delivery

How was it possible for the agile approach to remove over a year from the waterfall schedule?

There were a few reasons.

First, the team built a *fraction of the functionality* that they would have otherwise. They built only enough of the product to be "viable." That is, they completed enough functionality that a potential customer looked at it and said, "I can use that." In agile, we call this the "minimum viable product" and it's a key reason why using this approach leads to shorter development timeframes.

Second, they did technical design, built out the framework, built the functionality, tested and fixed bugs, *all during each two-week sprint*.

This provided two benefits:

- They spent no time building any technical framework that wasn't absolutely needed to support the features being built.

- They were forced to build the simplest thing that could possibly work. They had no time to get distracted by things that were more "fancy" than what was required.

Finally, instead of writing a requirements specification with hundreds of pages that no one would ever read, the requirements *emerged* throughout the process.

Later, I'll talk about how Scrum embeds the perspective of business stakeholders right into the team. This allows on-the-fly changes to requirements as necessary; especially when a requirement turns out to be too difficult to implement within the two-week deadline that otherwise would have been overlooked when the deadline was months or years away.

By getting the software into the hands of users more quickly, they'll give you feedback about what additional functionality they need. This avoids building features that sound like a great idea to the development team but will never be used by most users.

1.5 Scrum Is for Any Complicated or Complex Project

Many of the techniques, tips, and tools in this book can be applied to non-software projects, even though my examples and the technical portion of this book are focused on designing and building software.

If you have a complicated or complex project on your hands, I encourage you to experiment with Scrum techniques. I've seen non-software teams add just a daily meeting to their process and experience a significant improvement in their ability to get things done.

Scrum is very flexible and lightweight and has been used on a variety of projects in numerous industries—even to build a new jet fighter (Furuhjelm, et al., 2017). I'm sure you can adapt the techniques in this book to your specific challenge.

Key Takeaways

- Although you can get lucky, most teams don't naturally reach a state of high performance on their own. Scrum ensures teams do the right things to produce the highest business value in the shortest amount of time.

- Scrum is a social engineering framework to get teams to deliver amazing results to the business, including forcing them to make a plan, not giving them quite enough time, ensuring they communicate effectively, and driving empowerment and accountability.

- Scrum shortens timeframes by forcing all work to be done during each fixed-length production cycle, called a sprint. This ensures the only work that's completed is driving the highest business priorities. At the end of each sprint, the product is in a usable state, including all completed functionality to date. At any time, business leaders can decide to ship with fewer features than originally outlined, even if it's months earlier than planned.

- Although Scrum was created for software development, it can be used for work that has uncertainty about requirements and uncertainty about how exactly the work will be completed—so-called *complicated* or *complex* projects.

Chapter 2

Scrum's 10 Guideposts

Everything should be made as simple as possible, but not simpler.

- ***Albert Einstein***

One of the reasons I love Scrum is because it's *simple*. Three roles, four ceremonies, three artifacts. Within these 10 guideposts, you have free reign to decide exactly how to do the work.

By design, Scrum is super lightweight and incomplete. It acknowledges that no two teams will have exactly the same needs. Instead of attempting to create a rigid methodology, Scrum instead provides a framework within which you evolve your own customized process.

This book provides you with lots of ideas based on what I've learned over the last 35-plus years, but what I've outlined is by no means the only way you can get things done. I fully expect you to take what works for you and abandon the rest.

This chapter dives into the fundamentals of Scrum, providing a foundation of knowledge for you to leverage as you read through the rest of this book. Later, I delve deeper into the details and how to apply these fundamentals in the real world.

2.1 Three Roles

Instead of talking about job titles, Scrum identifies *roles*, with each having specific responsibilities.

Titles within the Scrum team are discouraged because a key part of the process is ensuring team members avoid, "That's not my job." Team members do whatever they're capable of, which helps speed the process.

The **Product Owner** is responsible for the overall vision of the application being built. They are responsible for what the development team will work on, and in what order. I like to call the product owner the "embedded business representative" on the team, as they are responsible for distilling the views of all the various business stakeholders and carrying that into the Scrum team.

The **ScrumMaster** is responsible for the process. I like to call them the "embedded Scrum coach" on the team. They help everyone understand and adhere to the fundamental Scrum concepts I outline in this chapter. They also protect the team from outside influences to avoid distractions, and they help the team resolve impediments to their progress. Finally, they help the team reflect on how well the process is running and help facilitate improvements.

Sometimes I jokingly call the ScrumMaster the "mom" of the team—male or female, it doesn't matter—as their job is to protect and nurture the team to the greatest extent they can.

The **Development Team** is a group of individuals who collectively possess all the required skills to complete the work. For a software project, this may include skills such as coding, testing, architecture, databases, user experience, visual design, etc.

This group—typically comprised of "seven plus or minus two" individuals—does all the work on the product. The product owner and ScrumMaster are not included in this number unless they are performing work to get the product done, in which case they are also part of the development team.

Note that even though this is termed "development team," it is used to represent any group of individuals performing the work of a Scrum team, regardless of the type of work being done.

2.2 Four Ceremonies

Instead of calling them "meetings," in Scrum we call get-togethers of team members "ceremonies" or "events." I prefer ceremonies, so that's the terminology I've used in this book.

Each of the ceremonies below is timeboxed. That is, they must be completed in the time defined by the timebox, or less whenever possible. This avoids losing productivity by spending too much time in group discussions, while at the same time ensuring the right amount of group planning and communication actually happens.

Sprint Planning happens at the beginning of each sprint, and is comprised of what I call "micro-planning." The development team, with help from the product owner and ScrumMaster, starts with the most important item and creates a plan to get the work done by breaking down the item into subtasks, each with a time estimate in hours. This ceremony is timeboxed to no more than four hours for teams with two-week sprints, and eight hours for month-long sprints.

The **Daily Scrum**, sometimes called the daily standup, ensures all members of the development team are communicating on a regular basis. Timeboxed to 15 minutes, this ceremony occurs at the same time and place each work day during the sprint.

The exact format is up to the development team, but a typical structure is to have each member of the development team answer the following three questions:

1. What did you get done since the last daily scrum?
2. What do you commit to getting done before the next daily scrum?
3. What impediments are standing in our way of completing the work in the sprint?

The product owner and ScrumMaster only participate if they are also a member of the development team.

The **Sprint Review** occurs at the end of each sprint and is a way to provide insight and transparency to business stakeholders. The Scrum team discusses the work of the sprint and their results. It typically includes a demonstration of the product, including any new functionality completed during the sprint and driven to a "shippable" level of quality. Finally, the Scrum team usually discusses what's coming up next on the product backlog.

Although it often includes a demonstration, this ceremony is designed to be a two-way interactive conversation between the Scrum team and the business stakeholders.

This ceremony is timeboxed to two hours for a two-week sprint and four hours for a month-long sprint.

The **Retrospective** is a way for the Scrum team to discuss how things went during the sprint and determine changes to improve their process. Because Scrum is just a series of guideposts, it's critical that each team adjusts what's inside those guideposts to what works best for them.

This ceremony typically follows the sprint review and often marks the end of the sprint.

Formats vary but one approach is to have the team discuss the following three questions:

1. What went well?
2. What could be improved?
3. What experiments should we try next sprint to improve?

This ceremony is timeboxed to a maximum of an hour and a half for two-week sprints and three hours for month-long sprints.

2.3 Three Artifacts

Keeping within Scrum's theme of extreme simplicity, there are only three artifacts that you need to manage in order to effectively utilize the framework.

The **Product Backlog** is a single shared source of truth as it pertains to the vision for the product. The product backlog represents the entire scope of the product to be produced by the team, typically broken up into releases or versions. For example, the Scrum team may be working toward version 2.3 of their product.

Included on the product backlog is all the work the team needs to do. This includes:

- *User stories,* which represent new functionality to be built.
- *Bugs* that were discovered in user stories that were previously thought to be "done."
- *Tasks* that provide value to the development team but not direct value to business stakeholders. Some Scrum teams call these *chores* but I call them *tasks,* to align with the terminology used in Jira, the most popular agile management tool.

The product owner prioritizes the product backlog by business value, and the development team works from the top down, completing as many product backlog items as possible during each sprint.

The **Sprint Backlog** is a set of product backlog items the development team has committed to driving to "done" during the current sprint, as well as a plan to get the work done, in the form of subtasks for each product backlog item.

The **Product** is the end result and is used to describe the output of the team, regardless of the type of work being done. For software, the product is the application being built. The Scrum framework dictates that the product must be at a "shippable" level of quality at the conclusion of each sprint.

This artifact is sometimes referred to as the "shippable product increment," as each sprint builds onto the state of the product at the end of the prior sprint, adding the product backlog items completed during the current sprint.

2.4 This Is Too Simple. How Can It Possibly Work?

When I explain these fundamental concepts to project managers unfamiliar with Scrum, I usually encounter a fair bit of skepticism. They can't believe that something that's so simple can be used to manage the types of complex projects they're used to.

"Where's the rest?" they may wonder, accustomed to a variety of charts, graphs, tables, documents and reports from their traditional project management training.

I think the answer is threefold.

First, Scrum puts a heavy reliance on interpersonal interaction, as opposed to comprehensive written specifications. My feeling is that most traditional project artifacts are designed to accommodate a lack of human communication. Specifications are detailed to such a degree that they'd hold up in a courtroom, all because they're designed to ensure no amount of misinterpretation, months or years later when the developers implement them.

Second, Scrum forces everyone needed to produce the product to work side-by-side throughout its production. Unlike waterfall, there is no "tossing it over the fence" from one group of specialists to another—for example from business analysts to technical architects, to user experience designers, to coders, to testers.

Finally, business interests are connected into the team through the role of the product owner. Because the business is represented the whole time, throughout every day, decisions can be made on the spot as questions come up.

The founders of agile software development summarized their principles succinctly in the Agile Manifesto: (Beck, et al., 2001a)

Individuals and interactions *over processes and tools*
Working software *over comprehensive documentation*
Customer collaboration *over contract negotiation*
Responding to change *over following a plan*

That is, while there is value in the items on the right, we value the items on the left more.

I realize that in some environments—for example, development of life sciences and medical software—there are additional documentation needs. There is nothing wrong with adding additional documentation as necessary, for your unique combination of business problem, team, and technology. However, Scrum essentially says, "Start small. Start with the simplest thing that could possibly work. Add additional things only if you really need to."

This approach has proven to result in better products produced in less time, that better meet market needs. This is in stark contrast to traditional software development approaches which seldom "start small."

Key Takeaways

- Scrum includes three roles: the **product owner**, who is in charge of the vision and is the embedded business representative on the team; the **ScrumMaster**, who is in charge of the process and is the embedded Scrum coach on the team; and the **development team**, comprised of five to nine individuals who collectively can do all the required work to produce the product.

- Instead of "meetings," which have a bad reputation, Scrum uses four "ceremonies" for the team to coordinate their work. These include **sprint planning**, during which a plan is created for getting the work done during the current sprint; the **daily scrum**, where development team members coordinate their work for the day; the **sprint review**, which is an interactive discussion with stakeholders about the state of the product; and a **retrospective**, where the team improves their process.
- There are three artifacts in Scrum. They include the **product backlog**, which is a prioritized "wish list" representing the wants and needs of users and other stakeholders; the **sprint backlog** which includes a selection of product backlog items plus a breakdown of the work required to get them done; and the **product**, which is the result of the team's work and must be in a potentially shippable state at the end of each sprint.
- Scrum is much simpler by design than traditional project management approaches. It works by supporting the key concepts from the Agile Manifesto, including individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan.
- Scrum ensures the work of the development team is always aligned with business needs, by connecting stakeholder priorities directly into the team through the role of the product owner, and the use of the product backlog.

Chapter 3

The Product Backlog

The bottom line is, when people are crystal clear about the most important priorities of the organization and team they work with, and prioritize their work around those top priorities, not only are they many times more productive, they discover they have the time they need to have a whole life.

- **Stephen Covey**, *author of The 7 Habits of Highly Effective People.*

Many software teams fail to produce the right functionality to drive the priorities of the business. After months or even years of work, they'll show the finished product to stakeholders only to hear, "That's not what I wanted."

The team coded against the specification and they felt pretty good about how well the product functionality matched what's in the document. Confused by their reaction, the team may ask for an explanation and stakeholders respond, "Things changed since that spec was written" or "I was thinking it would work differently."

The problem is threefold with traditional waterfall software development:

- Features are seldom force-ranked by business value, so teams spend too much time on functionality that produces little return on investment.
- Too much time goes by between the spec being written and the stakeholders seeing working, shippable software. By the time they provide feedback, it's too late to make substantial changes.
- There isn't enough interaction between business visionaries and the development team to keep them informed about changing priorities along the way.

Use of the product backlog in Scrum solves these problems by providing a shared vision used by business stakeholders and the team throughout the development of the product. Interaction between the product owner and the development team happens throughout every day. This ensures the needs of the business are directly connected to the ongoing development work throughout the entire process.

Finally—and probably the most important—the product owner places the most valuable stories at the top of the backlog. This ensures high value-generating items are completed first and are delivered sooner rather than later. This prevents less important items stealing development time from other, more profitable user stories.

3.1 The Single Shared Source of Truth

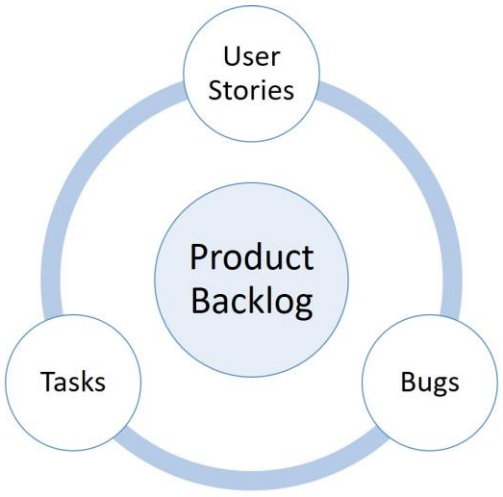
The product backlog provides a shared "source of truth" as it pertains to the vision for the product. It allows both the business stakeholders and development team to always understand the vision for the product.

It's essentially a whole-product wish list, owned by the business, and ordered by business value.

The backlog contains everything the team needs to do in order to complete the product in a way that matches the vision of the product owner.

Types of Items on the Product Backlog

The backlog contains three different types of items: user stories, tasks, and bugs.



Different types of items on the product backlog.

- *User stories* are lightweight requirements that represent new functionality that delivers value to business stakeholders; including end users. For example, "As a Shopper, I want to check out, so I can get my products shipped to me."

- *Tasks* represent items the development team would like to do, that provide direct value to the team and indirect value to business stakeholders. For example, "Update developer workstations to the latest version of coding tools."
- *Bugs* are defects that made it into the product when it was deemed shippable in a prior sprint—bugs that made it through the development team's testing process undetected. For example, "Search results are incorrect if I enter two words separated by a space."

The Order of the Product Backlog

When a new product backlog is first created, it will likely only contain user stories. An example product backlog for a fictitious e-commerce web application is shown below.

1. As a Shopper, I want to check out, so I can get my products shipped to me
2. As a Shopper, I want to review my cart, so I can make adjustments prior to checkout
3. As a Shopper, I want to view a list of products, so I can select some to purchase
4. As a Shopper, I want to log in, so I can manage my account
5. As a Shopper, I want to review my orders, so I can see what I've purchased in the past
6. As an Administrator, I want to modify the list of products, so I can adjust our offerings over time
7. As a Fulfillment Specialist, I want to print a picking report, so I can prepare products to ship
8. As a Fulfillment Specialist, I want to print packing labels, so I can ship packages
9. As a Finance Employee, I want to view analytics about orders and revenue, so I can see how we're tracking against our goals
10. As a Shopper, I want a gift registry, so I can share what I want with friends and family

At first glance, the order in which I've put the stories in my example may be a little confusing. You may wonder how a user can check out if they can't browse the list of products yet.

In this case, the company understands that their checkout experience is the most critical part of the application. By working on checkout first, they can ensure that they spend the most amount of time "living with" the feature, as the rest of the product is built around it.

Many teams start with building the login screen. They figure, "The first thing the user will do is log in." But how much time does a user spend on logging in to your application, and just how risky is it that you'll get it wrong?

Tasks and Bugs on the Product Backlog

Once the Scrum team is up and running, the development team will likely ask for some tasks to be added. For example, there may be an upgrade to one of their development tools that was recently released, and they'd like to upgrade the developer workstations. They might ask the product owner to add, "Update developer workstations to latest version of coding tools."

By managing this type of work on the product backlog, it provides visibility to the product owner and allows him to properly prioritize it. If the sales team is waiting on a handful of features to demonstrate to a billion-dollar customer in a few weeks, the product owner can work with the development team and ask, "Can this wait a few weeks?" Without this level of visibility and control over priorities, the development team could have gone off and spent a few days on the upgrade, instead of completing the critical features.

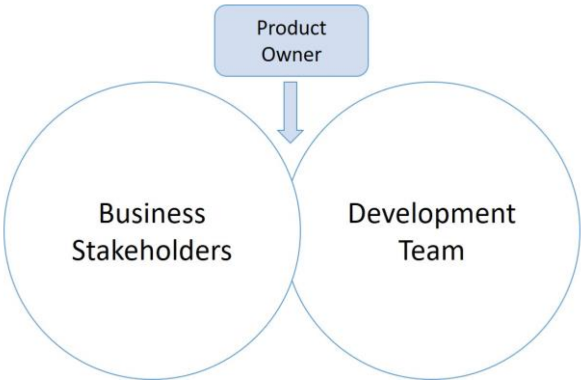
Once the first shippable product increment is produced, there will inevitably be a bug or two discovered during each sprint missed by the development team during their testing of the product in an earlier sprint. For example, a bug might be added along the lines of, "Incorrect message shown when removing an item from the cart if the item was the last one in the cart."

3.2 The Product Backlog Belongs to the Product Owner

With traditional software development, many teams take liberty with written specifications and decide independently to add some "neat" or "cool" enhancements—what I call "four-letter word" features.

They have the best intentions, and most teams make a valiant effort to attempt to support what they believe are the priorities of the business. But they seldom succeed. They don't have the perspective of the business to make the right decisions. They simply don't have the context of what stakeholders need from the product.

One of the things I love about Scrum is that we *shove* the business right into the Scrum team, in the form of the product owner. The product owner is the business representative, embedding the collective intelligence and vision of all stakeholders directly into the team.



The product owner is the only one who has the perspective of both the business and ongoing development of the product.

The product owner is responsible for understanding the business drivers related to the product. Because she is the only one who has the perspective of both the business as well as the ongoing development of the product, she is the only one permitted to decide what's on the product backlog, and in what priority order.

The product owner is the only person in the entire company who has the perspective of both the needs of business stakeholders and an intimate understanding of what's happening day-to-day on the development team. Therefore, she is the only one who is actually *qualified* to own the product backlog.

No one else in the company—not even the CEO—can overrule the decisions of the product owner.

Of course, this means there is quite a bit of pressure on her to make decisions that result in the team delivering the highest possible business value, but she won't be able to please everyone all the time.

Certainly, the opinions of the CEO and other powerful figures in the company are going to heavily influence her decision-making process, but her decisions about the product backlog are final.

3.3 Requirements Evolve Over Time

The comprehensive software specifications that are common with waterfall projects assume that requirements can be nailed down at the beginning and remain largely unchanged.

However, when we try this approach we discover three things:

- When the software is built, and it's placed in the hands of actual end users, changes will be requested when they discover that it doesn't quite work as well as it seemed "on paper." Since this happens at the end, it's too late to make significant functionality updates.
- Certain parts of the software application will be more difficult to build than originally anticipated, so it takes a lot longer to complete the product when the team has little flexibility in changing the specified behavior along the way.
- During the time the product is being built, business conditions change. New markets or specific customers are targeted, resulting in a change of the vision and, by extension, the required functionality.

Scrum acknowledges the high-risk nature of software development and instead of attempting to lock things down for long spans of time, it says, "Let's embrace change." The product backlog is in a constant state of evolution. The only requirement to "lock down" a portion of the product is for the scope of work in the very next sprint, ranging from one week to a calendar month.

3.4 Discussing the Backlog with the Team Is Important

Most development team members don't spend as much time reading project documentation as we'd like. To combat human nature, instead of assuming the team will read the documentation, we talk about the product vision instead.

The process of refining the product backlog—sometimes also called grooming the backlog—happens throughout the development of the product. Most Scrum teams get together one or two times during each sprint to talk about what's coming up next on the product backlog.

This has the following benefits:

- The product owner can explain the vision for each upcoming story, often discussing the background of the story and how it provides value.
- The team can ask questions to get more details, increasing their understanding of what the product owner has in mind.

- Because the team will be on the same page, the planning process at the beginning of the next sprint will go much more smoothly and take less time.
- If there is anything confusing about upcoming stories or open questions from the development team, the product owner has some time to get things straightened out before the next sprint planning ceremony.

Collectively, this discussion does what I call "programming the team's collective subconscious." It gets everyone thinking about what's coming up next and, in the back of their minds, they'll start thinking about how they might build it during the next sprint.

3.5 Tracking the Projected Schedule with a Release Burndown Chart

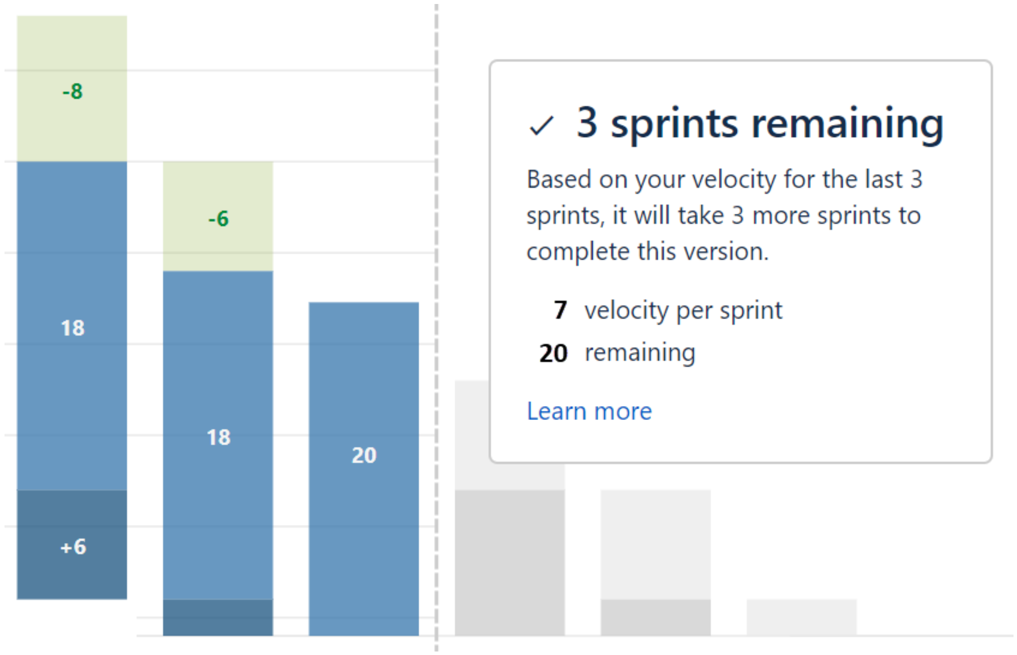
Although the product backlog typically represents all the envisioned work on the product far into the future, the focus of most Scrum teams and their stakeholders is on a more short-term goal, often measured in terms of months as opposed to years. This short-term goal is defined by the product owner, and is aligned with the needs of stakeholders and the organization.

This goal is represented by a subset of the product backlog, starting at the top and including everything the product owner believes should be the next milestone for development of the product. This group of product backlog items is called a *release*, and allows tracking progress toward the goal and forecasting a timeframe to reach it.

The release burndown chart helps everyone understand how much time is left to complete the release, based on how fast the team is moving and how much work is left.

The amount of work remaining is driven by a size estimate, measured in *story points*, which I'll talk about in the next chapter. The speed of the team is determined by how many story points they have completed during prior sprints, which is called *velocity*. Typically, an average velocity of the last three sprints is used for the release burndown.

For example, if there are 20 story points remaining to be completed for the release, and the team's average velocity for the last three sprints is 7 story points per sprint, the release burndown calculates that the team has 3 sprints left.



✓ 3 sprints remaining

Based on your velocity for the last 3 sprints, it will take 3 more sprints to complete this version.

7	velocity per sprint
20	remaining

Learn more

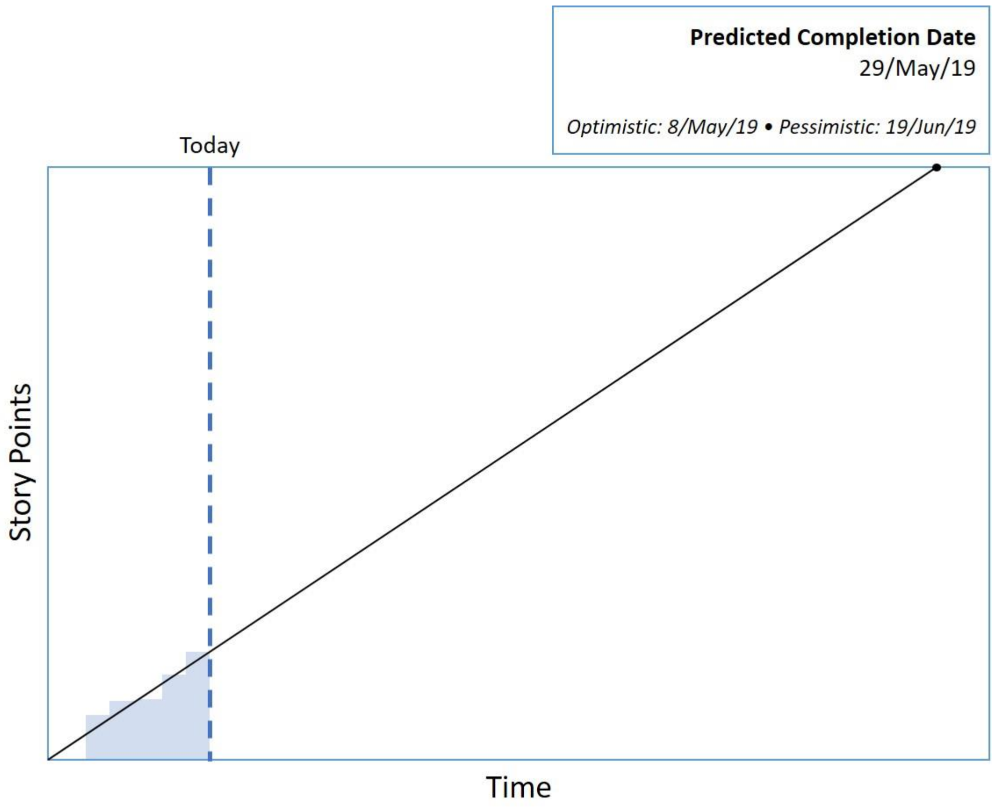
The release burndown chart uses the average velocity of the team to project the number of sprints to complete the release, based on the amount of remaining work. This report, generated by Jira, shows the velocity of the team, as well as any increase in scope added during each sprint.

If you multiply this by your sprint length—for example, 2 weeks—that gives you a projected timeframe. In this case, it's 3 sprints times 2 weeks per sprint, or 6 weeks.

Because the velocity of the team can change over time, this is only an estimate, so be careful about making promises about a hard date based on this information.

3.6 Projecting a Range of Dates with the Version Report

Other types of charts and reports can be used to track progress with the current release. For example, Jira includes a version report, which uses the same information as the release burndown—the velocity of the team and the amount of work remaining—but shows the information in a slightly different way.



The version report is an alternative way to visualize how long it will take to complete the current release.

One nice feature of the version report is it shows a completion date *range*, recognizing that this is an inexact science. Included is a predicted date based on the team's current average velocity, as well as an optimistic and pessimistic date based on a plus or minus 10% range of velocity.

This considers the possibility that the team's velocity to date may go up or down a bit from now until the completion of the release.

The release burndown, version report, and other tools are extremely valuable to the product owner and business stakeholders, as they allow them to see what happens if they add or remove scope to or from the current release.

As I talked about in an earlier chapter, one of our clients was able to ship months earlier than originally planned by trimming down the number of user stories in the release of their product. We were able to do this by leveraging these reports as we worked together to adjust scope and immediately see the effect on the date.

We did this in *real time*. We moved some low-priority user stories to the next release, then we looked at the predicted, optimistic, and pessimistic dates on the version report. If the pessimistic date was after the date we wanted, we pushed off some more stories. We repeated this process until we were satisfied with the date.

Key Takeaways

- The product backlog allows stakeholders to drive the work of the Scrum team by directly connecting business priorities into the team. It represents a single shared source of truth representing the vision for the product.
- The product backlog includes **user stories**, which represent new functionality; **bugs**, which represent software defects; and what I call **tasks**, which represent the work the development team would like to do to move faster but provides no direct value to stakeholders.
- The backlog is force-ranked by business value, so the most important work is always done first. This prevents less important items stealing development time from other, more profitable user stories.
- The Scrum team routinely discusses the product backlog, so everyone is familiar with the vision for the product.
- The short-term goal of the Scrum team is represented by a subset of the product backlog, which is termed a *release*.
- The release burndown chart lets the Scrum team and stakeholders understand how much time is left to complete the current release, by using the velocity of the team and the amount of work remaining. This gives a lot of power to the product owner and stakeholders by immediately seeing the effect of any scope changes on the projected release date.

Chapter 4

User Stories

The single biggest problem in communication is the illusion that it has taken place.

- ***George Bernard Shaw****, Irish playwright and novelist, winner of the Nobel Prize in Literature.*

Much of the frustration I've seen on software teams over the years can be boiled down to one problem: communication. Whether it's a misunderstanding about what needs to be built or how a feature should work, time is lost and tempers can run high due to poor communication.

This problem is especially important to address with agile projects, with their focus on rapid delivery and adapting to change. Although with agile there aren't verbose written specifications, there still needs to be a way to ensure proper communication takes place.

User stories provide the basis of communication in Scrum and are the foundation of shared understanding. They facilitate the direct connection between the business and the development team, to ensure that what gets built delivers the highest business value in the shortest time.

User stories are part of an agile approach that helps shift the focus from *writing* about requirements to *talking about them*. User stories include a written sentence that summarizes the desired functionality in the product and—more importantly—a series of conversations about the desired functionality. (Cohn, n.d.)

4.1 A User Story Is a Descriptive Sentence

The key to any user story is the sentence, which is a brief phrase that describes a desired capability of the product from the perspective of a specific type of end user.

The type of end user is typically specified in the form of a role—what some may also call a persona or actor.

Here's an example user story for a fictitious e-commerce web application:

As a Shopper, I want to check out, so I can get my products shipped to me

This story is written using the following format:

As a [role], I want [capability], so I [benefit]

Where:

- *Role* explains the type of user who is interested in the capability. For example, if we were building an accounting application, a role might be *Bookkeeper* or *CFO*. As I'll talk about in Section 19.2, a role can also represent another software system, for example if your product will support electronic interaction via an API.

- *Capability* is a description of the desired functionality. This could be: *reconcile bank accounts* or *print a paper check*.

- *Benefit* is a description of the value delivered once the functionality is in the application. It helps answer the question, "Why does this type of end user care about this feature?"

Some Scrum teams include just the first and second parts for their stories—As a [role], I want [capability]. But this third part is often a helpful reminder about what's in the product backlog—the *why* behind each user story.

It may sometimes make sense to omit the benefit if it's obvious why someone would want the feature. However, I find it's helpful to always include the third part of the sentence, to ensure everyone is on the same page and they understand the "so what" behind each story.

This provides value not only to the product owner and business stakeholders—as a reminder of why the story is important—but also to the development team so they can understand what drives the desire for a story.

The development team may have creative ideas about exactly how to implement a user story. Helping them understand the underlying motivation can provide them with additional context to help them complete the story more quickly and leverage their collective insight and experience.

Here are some complete examples of user stories you might have for a financial application:

- As a Bookkeeper, I want to reconcile bank accounts, so I can make sure the bank balance in the application is accurate

- As an Accounts Payable Specialist, I want to print paper checks, so I can pay bills for vendors that don't accept electronic payment

- As a Chief Financial Officer, I want to view a financial dashboard, so I can monitor financial metrics

4.2 Details Are Added Through Conversations

When initially writing user stories, the sentence is typically enough to start a conversation with stakeholders and within the Scrum team, which is the primary goal of this technique.

As I said at the beginning of the chapter, *talking about requirements* is the goal, as opposed to creating comprehensive written specifications. This helps support the Agile Manifesto principle of *working software* over *comprehensive documentation*, which I discussed in Chapter 2.

However, once the conversations begin, it's helpful to make some notes about the details that come out of those discussions. These serve as a high-level reminder to the team about the fine-grained details of each story, so they don't have to keep everything in their heads.

This is *not* falling back into old habits by writing pages of information about each user story. Instead, a few notes—I recommend shooting for 5 to 15 bullet points—are added to the story to capture details.

There are different ways to manage these notes. I prefer the use of *acceptance criteria*, which is a list of conditions that need to be met to consider the user story "done."

I like to call the list of acceptance criteria the "product owner's checklist." That is, they document the answer if the product owner was asked, "What are the key things you want to see when you use the application to determine whether this story satisfied your vision?"

This is important because before a story can be called "done," the product owner must review and accept the work of the development team. The acceptance criteria, at that point, can be used as a high-level checklist for the product owner, along with any acceptance tests the team has chosen to write.

Here's an example story and its acceptance criteria:

As an Accounts Payable Specialist, I want to print paper checks, so I can pay bills for vendors that don't accept electronic payment

- I can set the starting number for checks.
- The number is automatically incremented each time I print a check.
- I can override the automatically-assigned number when I print a check.
- I can manually enter a check, to record a paper check I wrote by hand.

- I can customize the layout of printed checks.
- I can print checks for more than one checking account.

There are only a few acceptance criteria in the above example, but compared to just the sentence, the level of clarity has been dramatically increased. The team can now gain a much better understanding of what the product owner is thinking for this functionality.

You'll notice I started the acceptance criteria with "I can…" I like to use this convention because it helps put the team in the shoes of the end user. This helps the team think about what acceptance criteria are required and helps ensure everyone is always thinking about delivering value to the user.

Notice that *The number is automatically incremented each time I print a check* doesn't use the "I can…" format because it's something the application should do behind the scenes, as opposed to a capability it provides directly to the user.

4.3 What, Not How

Notice that the acceptance criteria in the previous section are fairly vague. There is no description of how the user interface might look, for example. And there isn't any description of how, exactly, the end user will customize the layout of checks.

Talking about the *what*, but not the *how* is by design, and provides a few benefits:

It gives the development team more latitude to come up with solutions. One of the development team members might have worked on another team a few months ago, and they built something that might be reusable in our product. We want them to have enough wiggle room to leverage their creativity and experience, which often results in faster development.

What that other team built for their product may not exactly match the product owner's vision, but since the user story is focused on the end result—and not how to achieve it—there's a higher likelihood the developer's experience can be used as-is.

It helps defer the details. The team can discuss the story at a high level, and start to think through it, without the user experience (UX) design. This is especially helpful to avoid progress being held up while waiting for the UX team to perform their initial usability research and product visioning.

It keeps the user story lightweight. Keeping the details at a high level helps avoid the urge to write pages of specifications about each story. However, there's enough detail captured to give the development team a good idea of the scope of the story and generate additional discussion as necessary as the functionality is built.

4.4 Each User Story Has a Size Estimate

One of the challenges business leaders face when working with development teams is determining how long it will take to add a specific feature to the product. If the feature is quick to implement and provides a decent amount of value to end users, it's likely to be prioritized higher on the backlog. However, if the feature will take a lot of work and provides only a small amount of value, it's likely to be deferred until later, or sometimes never built at all.

All too often business leaders will walk up to one of the developers and say, "Hey, Cyndi, how long do you think this will take?" Being put on the spot, Cyndi has only a few moments to think about the feature and replies, "Hmm. Probably a day or so."

The business leader smiles and says, "Thanks." Then he proceeds to promise the feature to a customer with a specific timeline. Chaos ensues when it ends up taking much longer than Cyndi anticipated.

What Cyndi isn't thinking about is all the various implications of implementing that feature, including properly testing it, finding and fixing any bugs, and discussions that may need to happen to ensure it's implemented properly. The other problem is *she might not be the one implementing it*, and another developer might take much longer.

Size, Not Time

Experience has shown this type of ad-hoc estimating is highly inaccurate.

First, it's based on the opinion of one individual, which doesn't consider the dramatic variation from developer to developer in the amount of time it will take to complete a particular feature. Second, one person is unlikely to think about everything involved in implementing that feature.

Instead of using time, I recommend an agile estimating technique of determining the *size* of each user story. The size of the story is related to other stories. This allows understanding that some stories are larger, which will take more effort, and some smaller, which will take less effort.

Techniques for Specifying Size

There are different ways to specify a size of a user story. One technique used by some Scrum teams is using t-shirt sizes—extra-small, small, medium, large, extra-large, etc. This is an OK approach, but I don't like it because you can't easily add the sizes from multiple stories. What's the sum of a small, two larges, and an extra-large?

Instead, I prefer using *story points*, which is a scale that allows assigning a number-based size to each story. 0.5, 1, 2, 3, 5, 8, etc. The big benefit of this approach is you can add the story points together. 1 + 3 + 5 + 5 = 14.

This allows the use of simple math to determine the size of the remaining work for the next version of the product and—based on how many points the team can complete during each sprint on average—how much time is left.

The Development Team Determines the Size Estimate

An agile estimating technique called *Planning Poker* is used to facilitate a group discussion about each user story and generate a collective opinion from all members of the development team about its size. This avoids the problem of asking just one person to come up with an estimate, and results in a more accurate size for each story.

I get into much more detail about story points and agile estimating techniques later in this book.

4.5 Additional Written Details Are OK

As I've worked with teams over the years, I've seen them want to capture additional details in the user stories. There is nothing wrong with this. The detail in the user stories should be kept succinct, but not so abbreviated that the team gets slowed down.

The agile concept of always trying to find "the simplest thing that could possibly work" applies here, so you want to start with lightweight details. But don't kill yourself by saying, "We're not really supposed to be writing a lot of stuff down," if that comes at the expense of better communication.

How do you know when to add additional details? The simple answer is, "When the development team needs them." When discussing a story and there's any level of confusion, talk it through then capture the important points in the story.

Below are some examples of additional details I've found can be helpful to capture. Keep in mind that you don't need *all* of these details for *every* story, so use them selectively.

Background

When the product owner describes a story to the rest of the Scrum team, he will often talk a little bit about the origins of the story—the background of why the story is needed. Sometimes it's helpful to write a sentence or two to capture this background, describing what led to including the story in the product backlog. This can be especially important for stories that seem like they are at a higher level of priority than they deserve.

Here's an example:

> Background: When we released version 1.0, we thought this feature would be used only intermittently, and we implemented a very basic version. We found that some of our largest customers are using this feature almost every day, so this story includes enhancements they've requested.

Notes

This is a general catch-all for any information that might be helpful, but isn't captured in the acceptance criteria.

Example:

> Notes: We're still waiting on some feedback from a handful of top customers for some of the fine-grained details about this story. We should make sure we have that feedback before this story is added to a sprint.

Technical Notes

When discussing a story with the development team, they will often talk a bit about how they might implement it. Although extensive discussion related to every detail should be deferred until the story is in a sprint, sometimes it's helpful to capture some technical details when the information is fresh in the team's head.

Example:

> Technical Notes: Alex has used the following spell-checking library in some other products. We should consider it when we work on this story:
>
> https://www.npmjs.com/package/azure-cognitiveservices-spellcheck

Estimating Assumptions

When the development team estimates the size of stories, they will often make some assumptions. It's helpful to make a note about these assumptions because if they change, the story will need to be re-estimated.

Example:

> Estimating Assumptions: We will use an open-source component and will not build the spell-checking engine from scratch.

A Complete Example

If the team decided they needed *all* the optional information I've described above, a complete example might look like the following:

As a Bookkeeper, I want to reconcile bank accounts, so I can make sure the bank balance in the application is accurate

Background: Customers need to make sure their banking transactions in the application match what's in their bank account, so they can ensure the bank balance in the app matches reality.

Acceptance Criteria:

- I can import transactions from my bank.
- I can see matching transactions based on dollar amount.
- I can create a new transaction on the fly if it's not in the system yet.
- I can create a new vendor on the fly if it's not in the system yet.
- I can create rules that will automatically classify recurring transactions.

Notes:

Jane Smith from our customer ACME Inc. said she'd love to show us an example of how she does this by hand today using Excel.

Technical Notes:

Natalie has used this package on a prior project:

- https://www.nuget.org/packages/Mocoding.Ofx/

Estimating Assumptions:

- Transactions will be imported from a file.
- We'll be able to find a pre-built component, so we don't have to write this from scratch.
- Up to two import formats. If more formats are required, we should re-estimate.

4.6 Some Traditional Specifications Are OK, Too

When I work with new teams that are used to working with extensive written documentation, they sometimes ask, "Are we *allowed* to use any other types of specifications?" The answer is absolutely "Yes!"

The idea behind this lightweight user story-based approach to software specifications is not to *prevent* you and your team from utilizing documentation that's helpful, it's to avoid doing *more than what's necessary*.

Many traditional software specifications include a multi-part document template for each feature. With agile what we're saying is, "Let's not assume we need to complete this entire 14-part template for *every* feature. Let's just add what we need."

This lets teams move more quickly because they have less baggage. But there's no rule in Scrum that says, "Don't you dare create a flowchart!"

The simple fact is, some stories are more complex than others and can use more written details to ensure proper communication and shared understanding. As is the case with the additional written details I described in the previous section, the key way to decide whether to produce additional materials for each story is only if the development team needs them.

Examples

Some examples of supporting materials you might utilize are:

- User interface wireframes, showing the layout of screens in the application.
- User interface mockups or comps, showing the exact visual look of screens, including colors and other visual design elements.
- A business domain model, showing real-world concepts and the relationships between them.
- Spreadsheets, modeling calculations in the application.
- Flowcharts, showing various paths through the story.
- Technical reference documents, such as specifications for a third-party API.
- Written standards, such as company or legal documents or regulations that include compliance standards to which certain stories must adhere to.
- Other charts, diagrams, pictures of whiteboard brainstorming, and anything else the development team finds useful to facilitate accurate and clear communication.

Add Additional Details as Late as Possible

When you do add supporting materials based on the needs of the development team, add them as late as possible.

Because priorities are always changing on an agile team, if you invest a lot of time creating supporting materials for a user story, you may find next week that same story has been moved to the bottom of the product backlog.

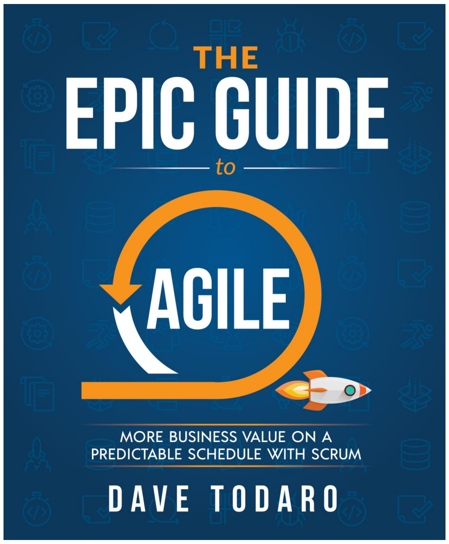
Ideally, create supporting materials in the *same sprint as the work to code and test the user story*. Completing them the sprint before is the second-best.

Avoid creating supporting materials months before the story will enter a sprint.

Key Takeaways

- User stories are part of an agile approach that helps shift the focus from *writing* about requirements to *talking about them*.
- A user story is a descriptive sentence. For example, *As a Shopper, I want to check out, so I can get my products shipped to me*, which follows the conventional format of *As a [role], I want [capability], so I [benefit]*.
- Additional details are captured using acceptance criteria, which is a high-level checklist for the product owner, to ensure key points about each story are recorded as a reminder to the Scrum team about conversations they've had about the story.
- User stories describe *what* is desired but not *how* to build it. This provides flexibility for the development team to come up with creative solutions.
- User stories have size estimates, expressed in story points, which allows everyone to understand the level of effort the development team estimates will be required to complete the story and drive it to a shippable state. This technique eliminates a focus on ad-hoc time estimates, which are highly inaccurate.
- It's OK to include additional notes and written details to clarify a user story, and even add some traditional requirements specifications. The key is to spend time writing down additional details only when needed to provide clear communication for the development team. Creating pages of documentation for every single user story is not necessary.

Enjoyed this free sample?



Buy Now