Microsoft®

Professional Scrum Development with Microsoft® Visual Studio® 2012

Richard Hundhausen

Praise for this book

"Richard provides real Scrum guidance for real teams. If you're a Scrum team using Visual Studio, this book is a great resource."

—Aaron Bjork, Principal Group Program Manager, Team Foundation Server, Microsoft

"Richard successfully marries the best tools for .NET developers to the most effective practices without sacrificing the people."

—David Starr, Senior Program Manager, Visual Studio, Microsoft

"Finally, a book about Scrum from the Development Team's point of view; Richard's description of the best and worst ways to implement Scrum is priceless. The first chapter alone is one of the best descriptions of 'Scrum done well' that I've ever seen."

—Charles Bradley, Scrum Coach & Professional Scrum Master

"The very first book on Team Foundation Server that I read was written by Richard, and he's done it again this time with another fantastic read."

—Brian Keller, Principal Technical Evangelist for Microsoft Visual Studio

"Richard does a fantastic job of blending theory, practice, and tools in one easy to read book! This book will surely be a staple for many of our Scrum coaching engagements."

—Chad Albrecht, VP Centare, PST

"As an encore to helping introduce the industry shaking Professional Scrum Developer program, Richard reminds us in this book why he's a leading voice in Scrum and Visual Studio ALM."

—Ryan Cromwell, Professional Scrum Trainer, MVP

"I've known Richard a long time and it's been great to follow his progression towards becoming a Scrum 'white robe.' I'm so happy the community now has the ultimate resource on understanding the marriage of Scrum and TFS."

—Adam Cogan, Microsoft Regional Director, Visual Studio ALM MVP [of the year 2011]

“If you’re new to Scrum or even if you’ve been doing it for a while, this book will help you get the big picture.”

—Benjamin Day, Professional Scrum Trainer, MVP

“If you’re using Scrum and TFS and you haven’t read this book, then you’re probably doing it wrong.”

—Brian Randell, MCW Technologies, Visual Studio ALM MVP

“In this book, Richard uses the core values of Scrum to describe how to get the best Scrum adoption of Visual Studio 2012. This is a superb combination of principles and mechanics that should be on all teams’ bookshelves.”

—Simon Reindl Professional Scrum Developer Trainer

“I don’t keep a lot of technology books on my bookshelf due to the pace at which developer tools evolve but this book, with its focus on people and processes, is definitely a keeper. Richard’s book is to Scrum development as Petzold’s was to Windows development.”

—Charles Sterling, Visual Studio Senior Program Manager, Microsoft

“Among the plethora of Scrum literature out there, Richard’s book makes a difference by bringing Scrum closer to where it belongs: the day-to-day work in the context of a team, supported by suitable practices, and the state-of-the-art Visual Studio toolset. You’ll benefit from most of the advice it contains, even if you don’t use Visual Studio!”

—Jose Luis Soria, Plain Concepts ALM Team Lead, PST

“Scrum, Visual Studio, and Team Foundation Server are just tools, and they will not make you better by themselves. If you really want to improve you need to understand the tools and learn how to improve, and definitively, Richard’s book will help you to get there”

—Luis Fraile, Visual Studio ALM MVP, Globe ALM Division Manager

“A masterpiece which distills the world of Scrum in a Visual Studio environment; anyone who is using Scrum will recognize many of the ‘smells’ and appreciate the sharing of real-world experience and guidance.”

—Willy-Peter Schaub, Program Manager, Visual Studio ALM Rangers

“This book should be required reading for everyone on your team. It will help you bring people, processes, and technology together quickly with Scrum.”

—Mike Vincent, Professional Scrum Developer Trainer, Visual Studio ALM MVP

Microsoft®

Professional Scrum Development with Microsoft® Visual Studio® 2012

Richard Hundhausen

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2012948863
ISBN: 978-0-7356-5798-4

Printed and bound in the United States of America.

First Printing

Acquisitions and Developmental Editor: Devon Musgrave
Project Editor: Rosemary Caperton
Editorial Production: Christian Holdener, S4Carlisle Publishing Services
Copyeditor: Andrew Jones
Indexer: Jean Skipp
Cover: Twist Creative · Seattle

This book is dedicated to my Scrum Team: Esmay, Isla, Berlin, Blaize, Sawyer, and Kristen.

Contents at a Glance

Contents

Foreword

By 2001, the software industry was in trouble—more projects were failing than succeeding. Customers began demanding contracts with penalties, and increasingly sending work offshore. Some software developers, though, had increasing success with a development process known as "lightweight." Almost uniformly, these processes were based on the well-known iterative, incremental process.

In February of 2001, these developers issued a manifesto—the Agile Manifesto. The Manifesto called for Agile software development based on 4 principle values and 12 underlying principles. Two of the principles were 1.) to satisfy customers through early and continuous delivery of working software, and 2). to deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

By 2008, the Scrum Agile process was used predominantly. A simple framework, it provided an easily adopted iterative incremental framework for software development. It also incorporated the Agile Manifesto's values and principles. The two authors of Scrum, Jeff Sutherland and myself, also were among the authors of the Agile Manifesto.

I had anticipated some of the difficulties organizations (and even teams) would face when they adopted Scrum. However, I believed that developers would bloom in a Scrum environment. Stifled and choked by waterfall, developers would stand tall, employing development practices, collaboration, and tooling that nobody had time to use in waterfall projects.

Much to my surprise, this was only true for perhaps 20 percent of all software developers.

Note In 2007, Martin Fowler characterized most Agile software development as "flaccid." He stated: There's a mess I've heard about with quite a few projects recently. It works out like this:

- They want to use an Agile process, and pick Scrum.
- They adopt the Scrum practices, and maybe even the principles.
- After a while, progress is slow because the code base is a mess.

What's happened is that they haven't paid enough attention to the internal quality of their software. If you make that mistake you'll soon find your productivity dragged down because it's much harder to add new features than you'd like. You've taken on a crippling Technical Debt and your Scrum has gone weak at the knees. (And if you've been in a real scrum, you'll know that's a Bad Thing.) *http://martinfowler.com/bliki/FlaccidScrum.html*

Martin's description of flaccid Scrum resonated with our experience. Most developers were skilled, but not adequately skilled in the three dimensions required to rapidly build complete increments of usable functionality. These dimensions are:

People The ability to work in a small, cross-functional, self-organizing team.

Practices The knowledge of and ability to apply modern engineering practices that short cycle development mandates.

Tooling Tools that integrated and automated these practices so that successive increments could be rapidly integrated without the drag of exponentially accruing artifacts that must be handled manually.

We put our business on hold while we worked through 2008 to create what has become known as the Professional Scrum Developer program. Offered in both a three- and five-day format, we formulated a workshop. The input was developers whose knowledge and capabilities produced flaccid increments. The output were teams of developers who had developed solid increments of software called for by the Agile Manifesto and demanded by the modern, competitive organization.

Richard has been there since the beginning. His book, *Professional Scrum Development with Microsoft® Visual Studio® 2012* continues his participation in the movement started by us few in 2009.

When you read Richard's book, you can learn the three dimensions needed for Agile software development: people, process, and tools. Just like the course, Richard intertwines them into something you can absorb. If you are on a Scrum team, read Richard's book. List the called-for practices. Identify which practices pose challenges to your team. Order them by their greatest impact. Then remediate them, one by one.

Many people spend money going to Agile conferences. Save the money and more by buying this book, discussing it with others, and going to Code Camps, the "un-conference" for the serious.

Richard and I look forward to your increased skill. Our industry and our society need it. Software is the last great scalable resource needed by our increasingly complex society. The effective, productive teamwork of Agile teams is the basis of problem solving that our society also needs.

Scrum on!

Ken Schwaber
co-creator of Scrum
September, 2012

In 2009, Richard took on a daunting task. Ken Schwaber and I came together because we lamented the impediment facing software teams trying to improve their ability to deliver customer value on frequent, short cadence. They could learn about practices, they could learn about tools, or they could engage coaching, but putting it all together was an exercise left to the readers.

That's when Richard Hundhausen stepped into the breach. He put together Professional Scrum Developer in a whirlwind. Quite literally, he toured the world delivering beta courses, relentlessly receiving feedback, and inspecting and adapting. The result was the first highly scalable training program that combined modern software engineering practices and readily available tooling at the global scale. Richard has been improving the course for three years through a dedicated community of certified trainers and has now distilled the basics into an easily accessible book.

If you're new to Scrum and want to get better at delivering high-quality software that your customers want quickly, Professional Scrum Developer is a great place to start.

Sam Guckenheimer
Product Owner, Visual Studio Product Line
Microsoft Corporation
September, 2012

Introduction

Scrum is a framework for developing and sustaining complex products, such as software. Scrum is just a set of rules, as defined in the *Scrum Guide* (*www.scrum.org/Scrum-Guides*), and it describes the roles, events, and artifacts, as well as the rules that bind them together. When used correctly, this framework enables a team to address complex problems while productively and creatively delivering products of the highest possible value. Scrum is an Agile method. In fact, it is the most popular Agile method in use today.

Scrum employs an iterative and incremental approach to optimizing predictability and controlling risk. This is due to the empirical process control nature of Scrum. Through proper use of inspection, adaptation, and transparency, a Scrum Team can try a new way of doing something (an experiment) and gauge its usefulness after a short iteration. They can then collectively decide to embrace, extend, or drop the practice. This includes the tools a team uses and how they use them.

Combining Scrum with the application lifecycle management (ALM) tools found in Microsoft Visual Studio 2012 is a powerful combination. It is the purpose of this book to establish a baseline understanding of Scrum, as well as how Scrum is supported in Visual Studio 2012. I will also illustrate which practices provide more value when executed *without* the use of tools. In addition, I will point out those tools which have been erroneously marketed as healthy when used by a collocated, collaborative Scrum Team.

In software development, anything and everything can change in a moment's notice. Healthy teams know this. They also know that continuously inspecting and adapting the way things are done is a way of life. High-performance Scrum Development Teams take it a step further. They know that within every dysfunction or impediment identified is an opportunity to learn and improve. Reading this book is a great first step.

Who should read this book

This book will be of value to any members of a software development team using Scrum. I primarily focus on the responsibilities and tasks of the developer (which in Scrum includes designers, architects, coders, testers, technical writers, etc.). Product Owners and Scrum Masters will also derive value from this book, as they will be using many of the same Visual Studio tools to plan and manage their work and assess progress. Stakeholders, including customers, users, and managers, will also gain value from this book, especially when they learn what they can and cannot do according to the rules of Scrum and which tools in Visual Studio support this.

Who should not read this book

This book is intended for teams using Scrum and Visual Studio 2012 together. It won't provide *as much* value for teams executing Agile (non-Scrum) software development and won't provide *any* value for teams running more formal "waterfall" software development projects, although Chapter 1 may hopefully change the minds of such proponents. Likewise, if a team is using Scrum, but not yet using Visual Studio 2012, the bulk of the book won't be very interesting. This is also the case for teams using Visual Studio 2012 *Express* or *Professional* editions, which don't contain the high-value, team-based tools for planning and managing the backlogs and team collaboration.

Organization of this book

This book is divided into three sections, each of which focuses on a different aspect of the marriage of Scrum and Visual Studio. Part I, "Fundamentals," sets a baseline understanding of the Scrum framework, Visual Studio 2012 editions and their interesting ALM features, as well as the Visual Studio Scrum 2.0 process template. Part II, "Using Scrum," provides several chapters detailing the practical application of how a Scrum Team would use the relevant features of Visual Studio 2012. Part III, "Improving," includes a chapter on identifying common challenges and dysfunctions in order to remove them, as well as techniques to continually improve your game of Scrum. By reading all sections sequentially, you will see how Visual Studio and Scrum can be used together in an effective way and how a team can become high-performance in the way it develops software.

Finding your best starting point in this book

The different sections of *Professional Scrum Development with Microsoft Visual Studio 2012* cover a range of topics. Depending on your needs and your existing understanding of Scrum, Visual Studio, and the related development practices, you may wish to focus on specific areas of the book. Use the following table to determine how best to proceed through the book.

If you are	Follow these steps
New to Scrum or have never heard of it	Read Chapter 1
New to Visual Studio 2012 or its ALM tools	Read Chapter 2
New to the Visual Studio Scrum process template or want to know what's new in version 2.0	Read Chapter 3
Familiar with Scrum and Visual Studio and only want to learn how to setup and manage a Product Backlog.	Read Chapters 4 and 5
Familiar with Scrum and Visual Studio and only want guidance on overcoming common challenges and dysfunctions.	Read Chapter 9

Conventions and features in this book

This book presents information using conventions designed to make the information readable and easy to follow.

- Screenshots from relevant Visual Studio 2012 features are provided for your reference.
- Boxed elements with labels such as "Note" or "Tip" provide additional information and guidance related to the subject.
- Some notes and tips are practical guidance provided by fellow Professional Scrum Developers who have helped review this book.

In addition, I have included two additional boxed elements, one labeled "Smells" and the other labeled "Tailspin Toys Case Study.". These are discussed in the following sections.

Smells Throughout this book, I point out specific situations and traps that a Scrum Team should avoid. I refer to these as *smells*. These smells typically indicate an underlying dysfunction or other unhealthy behavior. For teams new to Scrum, these smells may be hard to identify. Once they are brought to light, however, they should be used as learning opportunities. As a team improves, it should be able to recognize dysfunction on its own, as well as remove it. High-performance Scrum Teams reach the ability to identify potential waste, evaluate the risks, and even decide to opt-in to specific behaviors, including those that may be a smell to the uneducated.

Tailspin Toys case study As you flip through the pages, you will read about Tailspin Toys as a case study. This is a fictitious organization and team that is building an online retail website that sells model aircraft and accessories. The team has been using Scrum for some time and is moving to Visual Studio 2012. My opinions on healthy and unhealthy behaviors are made evident through the choices made by the Tailspin Toys team.

Code samples

Although this book contains almost no code samples, I did build a utility application to help create and manage the Product Backlog and Sprint Backlog. This helped me prepare the data seen in the various screen captures in this book. I affectionately named this utility the *Scrum Robot*. The source code is yours if you think it can be helpful. If nothing else, it demonstrates how to connect to a Team Foundation Server 2012 instance and manipulate basic team project data. The Scrum Robot can be downloaded from the book's companion content page:

http://www.microsoftpressstore.com/title/9780735657984

Note You will need to have Visual Studio 2012 with Team Explorer installed in order to use the Scrum Robot.

Installing and using the Scrum Robot

Follow these steps to install the Scrum Robot on your computer so that you can programmatically access Team Foundation Server and manipulate a team project's areas, iterations, Product Backlog, and Sprint Backlog.

1. Unzip the *ScrumRobot.zip* file that you downloaded from the book's website (name a specific directory along with directions to create it, if necessary).
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.

Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the *ScrumRobot.zip* file.

3. Once unzipped, you can open the *ScrumRobot.sln* solution and review the code. Press F5 to run the utility after changing any variables or constants, such as the name and address of your Team Foundation Server.

Acknowledgments

There are several people who helped me write this book: Christian Holdener, for his infinite patience. Devon Musgrave and Rosemary Caperton, for yet another opportunity to write for Microsoft Press. Fellow Professional Scrum Developers: Mike Vincent, Simon Reindl, Jose Luis Soria, David Starr, Jeroen van Menen, Chad Albrecht, Ryan Cromwell, Luis Fraile, Rob Maher, and Peter Gfader for helping me sharpen the message. Fellow Scrum and Visual Studio practitioners: Charles Bradley, Bob Hardister, Graham Barry, Anna Russo, Christofer Löf, Willy-Peter Schaub, and Peter Provost for providing great ideas and reviews. Thank you everyone.

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/ 9780735657984

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

PART I

Fundamentals

The chapters in this section will establish a baseline understanding of the three areas that every professional Scrum developer using the Microsoft tools platform must know:

- Scrum
- The Microsoft Visual Studio 2012 Application Lifecycle Management (ALM) tools
- The Visual Studio Scrum process template

We will begin by looking at Scrum and the rules of Scrum from the developer's perspective. The focus will be on how and when the Development Team interacts with the Product Owner and Scrum Master, participates in the various Scrum events, and uses the various Scrum artifacts. Remember that in Scrum, the term *developer* equates to a Development Team member. This does not necessarily equate to programmer or coder. In fact, Scrum recognizes testers, coders, designers, architects, analysts, and database administrators (DBAs) as developers. It's important for all developers to understand the rules of Scrum, and what's expected of them and their team, as well as when and how they can interact with the Product Owner, the Scrum Master, and the various artifacts.

The remaining chapters will be more technical in nature and cover the ALM tools found in Visual Studio 2012, including Team Foundation Server and its Scrum process template. This is Microsoft's fourth release of these tools and a lot has been added and improved from prior versions. With a full install of Visual Studio and Team Foundation Server, there are many tools available for a Development Team. I will endeavor to list and discuss the relevant ALM tools, but I won't explore the practice of using each. In my opinion, some tools are better left in the toolbox, allowing the team to exercise higher-valued collaborative practices instead.

CHAPTER 1

Scrumdamentals

Scrum is a framework for developing and sustaining complex products. Software is a complex product. Scrum is ideal for managing the development of software. Scrum is not a methodology or a process, although you can employ various processes within it. Software development doesn't generate the same output every time, given a certain input. Scrum embraces this fact and is empirical, which means that it promotes the use of observation and experimentation in order to inspect and adapt. This enables a team to regularly see the effectiveness of its development practices and make changes accordingly.

Even today, more than 60 years into the evolution of software development, the chances are a medium-sized to large software project will fail. Fortunately, the industry has finally noticed, understands, and has started to respond to this problem. Some organizations have turned this around. Things are improving. Evidence shows that Agile practices, such as Scrum, are leading these successes.

Tip Using a software development analogy, you can think of Agile as being an *interface*. Agile defines 4 abstract values and 12 abstract principles (*http://agilemanifesto.org*). While there are many ways to implement these values and principles, Agile does not describe them. Scrum does. You can think of Scrum as a *concrete class* that *implements* Agile.

Agile teams know that they must continuously inspect and adapt—not just their product, but their practices as well. Being book-smart on Scrum, Application Lifecycle Management (ALM), and Microsoft Visual Studio is a good start. Having experience using them together in practice is better. Being able to identify and act on opportunities for improvement as you use them is awesome. That should be your goal. Don't just settle for a non-failed project. Strive for completing the project better, faster, and cheaper than the stakeholders thought possible.

The *Scrum Guide*

Scrum has been around since the early 1990s. During that time, Scrum's definition and related practices have come from books, presentations, and professionals doing their best to explain it. Unfortunately, those messages were not always accurate and almost never consistent. Scrum, as it has emerged today, doesn't look like it did 10 years ago.

In 2010, Scrum.org codified Scrum by creating and publishing the *Scrum Guide* for free. This roughly 15-page guide represents the official rules of Scrum and is maintained by Scrum's creators, Ken Schwaber and Jeff Sutherland. It is available in 30 languages and downloadable at *http://www.scrum.org/scrumguides*. It is a great reference that you can use even as you are reading this book. As you read the guide, you will see that Scrum is lightweight and quite easy to understand. Unfortunately, it is extremely difficult to master. The *Scrum Guide* will continue to be updated and may supersede the guidance you read in this chapter and the rest of the book.

Tip You can think of Scrum as being like the game of chess. Both have rules. For example, Scrum doesn't allow two Product Owners just as chess doesn't allow two kings. When you play chess, it is expected that you play by the rules. If you don't, then you're not playing chess. This is the same with Scrum. Another way to think about it is that both Scrum and chess do not fail or succeed. Only the players fail or succeed. Those who keep playing by the rules will eventually improve, though it may take a long time to master the game.

The Scrum framework consists of the Scrum team and the associated roles, events, and artifacts. Each of these items serves a specific purpose, as you will see in this chapter. The rules of Scrum, as defined in the *Scrum Guide*, bind together the roles, events, and artifacts. Following these rules is essential to the success of a team's ability to use Scrum to develop a high-value, quality software product.

Scrum in action

If you study the *Scrum Guide*, you will understand the components and related rules. You won't necessarily see how they flow together. This requires you to actually experience Scrum while developing software on a team. As a substitute for that experience, Figure 1-1 was created by a fellow professional Scrum developer to illustrate the Scrum framework in action.

In Scrum, the Product Backlog is the single source of requirements for any changes to be made to the software product. This list includes features to be added, as well as bugs to be fixed. It is the Product Owner's responsibility to ensure that the Product Backlog is available, transparent, understood by the Development Team, and ordered (prioritized). The Development Team collaborates with the Product Owner, and others as needed, during Sprint Planning and Product Backlog grooming to understand and estimate the effort required to deliver the items in the Product Backlog.

The Sprint is a time-boxed event that contains the other Scrum events. Sprints should be a month or less in duration. The first event within a Sprint is the Sprint Planning meeting. In this time-boxed event, the Scrum team collaborates to plan the work of the upcoming Sprint. The Product Backlog items (PBIs), ordered at the top of the Product Backlog by the Product Owner, are discussed. The Development Team forecasts those Product Backlog items that it believes it can complete by the end of the Sprint. A Sprint Goal is crafted, and the Sprint Backlog emerges. The Sprint Backlog contains those items selected by the Development Team plus a plan for delivering them. The Sprint Backlog shows the work remaining in the Sprint at all times.

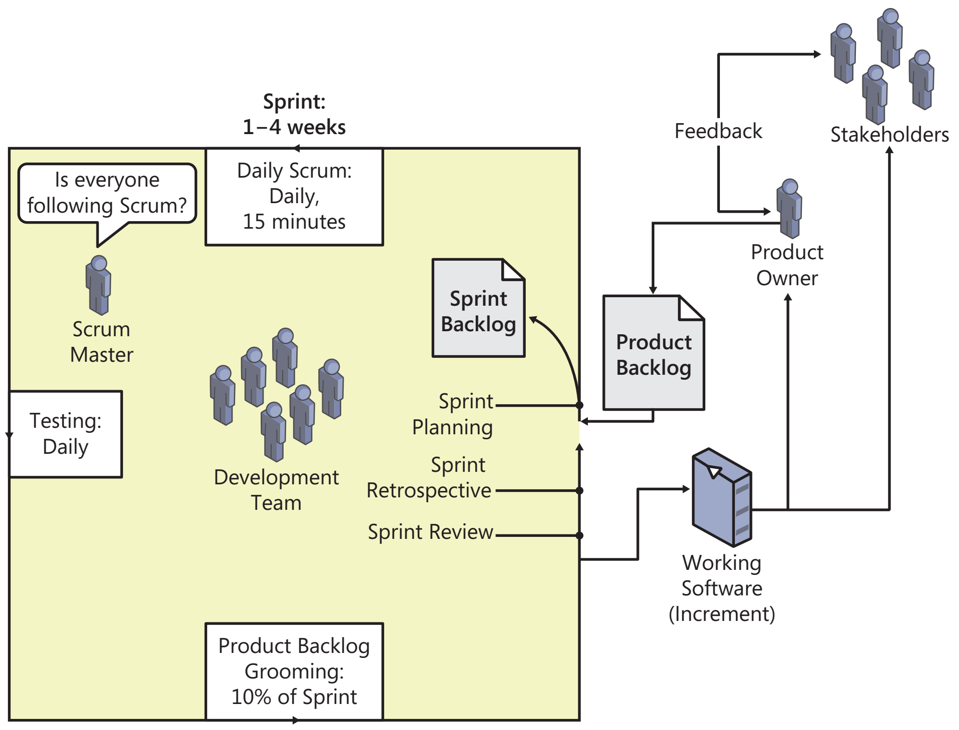


FIGURE 1-1 The Scrum framework in action.

The bulk of the Sprint's time-box will be spent developing the items in the Sprint Backlog. The rules of Scrum are fairly silent on what occurs each day during development. The Development Team must meet regularly for the Daily Scrum. This short meeting is for the Development Team to synchronize on what work will be executed in the next 24 hours. The Development Team should also meet with the Product Owner to groom the Product Backlog. During grooming, items in the Product Backlog are given additional detail, and estimates are given by the Development Team. This keeps the Product Backlog healthy so that the Product Owner can plan the software product's release and make better decisions on the items to develop next.

During the Sprint, the Development Team completes items in their Sprint Backlog according to each item's acceptance criteria and the team's Definition of "Done". This definition lists the practices and standards that must be met for every item before it can be considered complete. The definition is created by the Development Team but must be understood by the Product Owner. Both parties must understand that if work does not meet the Definition of "Done," it is not done and cannot be released. Ideally, the Development Team collaborates with the Product Owner throughout the Sprint to ensure that all criteria are being met. If the Development Team completes their forecasted work early, they should collaborate with the Product Owner to find another suitable Product Backlog item to work on. Conversely, at the first indication that the Development Team knows that they *won't* be able to complete their forecasted work, they should collaborate with the Product Owner to identify and discuss trade-offs and modify the Sprint Backlog to reflect the reality of the Sprint without sacrificing quality.

Sprint Backlog items done according the Development Team's definition are demonstrated during the Sprint Review meeting. The Product Owner may invite various stakeholders to this meeting for their feedback on the Increment. This Product Owner and stakeholder feedback might be captured and end up as new items in the Product Backlog. Existing items may also need to be updated or removed. The Product Owner may decide to release the Increment as soon as possible or delay it. This should be a business decision. Regardless of when the Increment is released, the Development Team should always develop the Increment as though it were *going to be* released as soon as possible.

The last event in the Sprint is the Sprint Retrospective meeting. This meeting provides an opportunity for the Scrum Team to inspect themselves and identify what went well and what needs improving. If improvements are identified, the team should create an actionable plan for the next Sprint. Nothing is out of scope during this meeting—people, relationships, process, and tools can all be discussed. The Scrum Team may also decide to adjust its Definition of "Done" to increase product quality. After the meeting, the next Sprint begins.

Scrum roles

The group of individuals who are responsible for, and committed to, building the software product is known as the Scrum Team. The Scrum Team is a superset of the Development Team. The Scrum Team consists of the following Scrum roles:

- Development Team
- Product Owner
- Scrum Master

As you will learn, there is an implied equality (that is, lack of rank or seniority) of the developers on the Development Team since Scrum does not recognize titles. That is not the case with the Scrum Team as a whole. The Product Owner is the visionary leader who chooses what is built, when it is ready to release, and when to stop or cancel the project. If you think of the roles in terms of providing service, the Development Team serves the Product Owner, while the Scrum Master serves both the Development Team and the Product Owner. Therefore, the Development Team has strong influence to select (that is, hire or fire) the Scrum Master. Correspondingly, the Product Owner has strong influence to select the Development Team he or she wants to turn the Product Backlog into done Increments. Because of this separation of duties, the roles should be played by separate individuals. This mitigates any chance of a conflict of interest. That said, smaller teams may find it necessary to combine roles.

Note Scrum Team != Development Team. The Scrum Team refers to the Development Team *plus* the Product Owner and Scrum Master. The Development Team refers to the subset of the Scrum Team that contains only the developers who will be developing the Increment. When someone uses the unqualified term "team" during conversation, it could refer to either. You may want to ask the person using the term to provide additional context.

The Development Team

The Development Team consists of between 3-9 professionals who are capable of building and delivering a potentially-releasable Increment of software at the end of a Sprint. The size of 6 +/– 3 developers allows the team to be small and nimble, while being large enough to complete increments of complex development. A team with only 2 developers doesn't need Scrum, as they can simply communicate directly and be productive. Also, there is a greater chance that the 2 developers won't have the skills required to do the work. On the other hand, teams with more than 9 developers require too much coordination. These larger teams tend to generate too much complexity to derive value from Scrum's empiricism.

Note The Product Owner and Scrum Master are not on the Development Team and are not included in the 6 +/– 3 Development Team size count. However, if the Product Owner or Scrum Master is *also* a developer who will be executing development tasks during the Sprint, then you should count them.

In Scrum, Development Team members are called "developers," regardless of their background, job title, or skill set. Development Team members may have experience in software engineering, testing, architecture and design, graphic design, database administration, business analysis, technical writing, or other similar specialties. Regardless of what their resume says, they are now "developers" as far as Scrum is concerned. They should burn their business cards and focus on delivering value in the form of working software. Also, there are no subteams in Scrum, such as testing or QA. The Development Team performs all of the work required to deliver the done increment of the software product.

It's important to note that just because a team member is called a developer, this does not necessarily mean that they will be developing (writing) code. Depending on the task, they may be *developing* architecture, *developing* user interface or design, *developing* test cases, *developing* database objects, *developing* installers, or *developing* documentation, etc. Everyone *develops* something. Table 1-1 lists the high-level activities that a Scrum Development Team will perform.

TABLE 1-1 Development team activities within Scrum.

Activity	When
Collaborate with the Product Owner to forecast the Sprint's work and craft a Sprint Goal.	Sprint Planning.
Collaborate with fellow developers on a plan to implement the forecasted work (including task estimation).	Sprint Planning, Daily Scrum.
Attend the Daily Scrum meeting.	Daily Scrum.
Develop the Increment according the acceptance criteria and the Definition of "Done."	After Sprint Planning and prior to Sprint Review.
Collaborate with the Product Owner to groom the Product Backlog (including PBI estimation).	During the Sprint. Product Backlog grooming makes up to 10% of Development Team's capacity during the Sprint.
Collaboratively identify additional development when forecasted work is completed early.	During the Sprint as needed.

Activity	When
Collaboratively discuss trade-offs and create a contingency plan for when the forecasted work can't be completed.	During the Sprint as needed.
Demonstrate each Increment allowing inspection by stakeholders and Product Backlog adaptation.	Sprint Review.
Reflect upon itself and its practices making delivery improvements.	Sprint Retrospective.
Continuously learn and improve.	Always.

Don't assume that a developer will execute only those types of tasks that he or she is good at or familiar with. For example, just because Dieter has a background in Microsoft SQL Server programming, that doesn't mean he'll be the one executing those types of tasks. If, during the Sprint, the team decides that the next logical task to execute requires SQL Server programming and Dieter is busy or unavailable, another developer should jump in and take on that work if at all possible. During development, the person who is best suited to perform a given task will emerge based on many factors, including expertise and availability. It is for this reason that estimates are made by the Development Team, not individuals—even if those individuals are experts in those domains. It's also why you should have more than one developer with a necessary skill set.

Tip I find very few teams whose members refer to each other as "developers." There is still a reflex to equate "developer" to programmer or coder. Our industry reinforces this. For these teams, and for the time being, using the term "Development Team member" or "team member" is a suitable substitute in my opinion.

Development Teams are cross-functional. This means that there is at least one developer on the team who has the necessary skill set to execute some type of work required for the Increment. Put a different way, it means that the Development Team has all the skills needed to complete its work. Being a cross-functional Development Team doesn't mean that each developer is cross-functional. Ideally, there will be more than one developer who has a required skill set. If not, then the team should strive to improve that by pairing and sharing, or by leveraging some other instructional techniques during development. Having one single developer on a team with a key skill is a recipe for dysfunction.

The composition of the Development Team does not change during the Sprint. If it must change, it may only change "in-between" Sprints. This is typically the result of a decision made collaboratively during the Sprint Retrospective meeting. Changes may include adding a new team member, swapping a member with another team, removing a team member, or changing a team member's capacity. Any change to the team composition is a disruption. Since Velocity is typically computed empirically, by looking back at the Development Team's accomplishments in prior Sprints, any change to the team composition will most likely cause a variance. It will take time for the Velocity to normalize. In other words, productivity will initially decrease for a time and then should (hopefully) increase.

Note Velocity is a measure of Product Backlog items that a Development Team delivers in a single Sprint. Velocity can be measured in the number, size, or business value of those items. Velocity of a single Sprint is not useful, but trending this number of several Sprints shows the general direction of productivity of a Development Team. Once Velocity has normalized, it is useful in planning Sprints and releases. For example, if a Development Team has an average Velocity of 20 points per Sprint and the Product Backlog shows 12 PBIs totaling 96 points yet to be developed in this release, you can expect the release to be available in roughly 5 Sprints, or 2 1/2 months given a 2-week Sprint duration. The term "Velocity" is rooted in the User Story practice, so it is not an official Scrum term. That being said, it can be adapted to other kinds of Product Backlog items, such as use cases, and used in Scrum as a planning tool.

Tailspin Toys case study The Tailspin Toys Development Team consists of seven cross-functional developers with varying backgrounds, skill sets, and skill levels. The team members are Anna, Art, Dave, Dieter, Raj, Toni, and Wade. Art and Anna have architecture, design, and some C# experience. Dave, Wade, and Raj have solid C# experience. Raj and Dieter have SQL Server and Windows Server experience, including Windows PowerShell. With the exception of Raj and Dieter, the Development Team is co-located and spends the majority of their time on the Tailspin Toys development effort. As a team, they all went through professional Scrum developer training and achieved passing assessment scores.

The Product Owner

The Product Owner represents the voice of the user. This means the Product Owner not only knows the product, its domain, and its vision, but also the users. Good Product Owners are in touch with the needs of the users. Great Product Owners will actually share in user's passion. Either way, the Product Owner should understand users' requirements and expectations. Just knowing how the product works and what to fix is not enough to be a competent Product Owner.

Note Over the years I've heard that the Product Owner is the voice of the *customer*. Lately, however, I've been seeing that the Product Owner is the voice of the *user*. I tend to agree with the latter, but what's the difference? Fellow professional Scrum developer Jeroen van Menen explains the subtle difference: the customer is the one who *buys* the software, where the user is the one who *uses* it.

Therefore, the Product Owner must represent the needs of the user and drive value in his or her direction, rather than just trying to satisfy the person writing the check. There is only one Product Owner on a Scrum Team. This helps avoid confusion. When the developers have a question about the product, their first instinct should be to ask the Product Owner. The Product Owner may need to consult other domain experts and stakeholders for the answer, especially for very large and complex products. The Product Owner should be considered the go-to person for all questions about the product's vision, value, release goals, features, and bugs.

The Product Owner is responsible for maximizing the value of the product through the work of the Development Team. The Product Owner's primary communication tool for doing this is a well-groomed and -ordered Product Backlog. The Product Owner collaborates with the Development Team on what and when to develop. A common misconception is that the Development Team develops the product. In fact, it's done through the collaboration and cooperation of the Development Team and the Product Owner. Table 1-2 lists the Development Team's interactions with the Product Owner.

Tip The ideal Product Owner should know the product, know the product's domain, know the product's customer, know the product's users, know Scrum, have authority to make decisions related to the direction of the product, be highly available to the rest of the Scrum Team, and have good people skills. Unfortunately, I've never met a Product Owner who had all of these attributes, but I have met many Product Owners who desired to improve in all these areas and worked toward that goal.

TABLE 1-2 Development team interactions with the Product Owner.

Interaction	When
Collaboratively plan the Sprint and forecast work.	Sprint Planning meeting.
Answer product and product domain questions.	During the Sprint as needed.
Groom the Product Backlog (including estimation).	During the Sprint. Duration should be up to 10% of Sprint length.
Take on additional work.	During the Sprint as needed.
Collaboratively plan contingency work.	During the Sprint as needed.
Demonstrate the Increment and adapt the Product Backlog.	Sprint Review meeting.
Collaborate to inspect the Scrum Team's practices and plan for improvement.	Sprint Retrospective meeting.

High-performance Scrum Teams understand the separation of duties between the Product Owner and Development Team and have come to rely on each team member doing his or her part. Although the *Scrum Guide* doesn't explicitly state that the Product Owner cannot be the Scrum Master or a Development Team member, I think those are good rules to set and follow. Keeping the Product Owner focused on *what* to develop, the Development Team focused on *how* to develop it, and the Scrum Master focused on ensuring that everyone understands and follows the rules of Scrum is a recipe for success.

Since the organization may hold the Product Owner accountable for the profit or loss of the product, he or she should maintain a constant vigil for optimizing the product's value. Passionate Product Owners tend to be engaging Product Owners. They continuously want what is best for their software product and, more importantly, the value provided to its users.

Tailspin Toys case study Paula is the Product Owner of the Tailspin Toys web application. She is the daughter of Buzz, the company's founder, and shares his passion for aviation and model aircraft. She cares deeply about Tailspin Toys' customers and community. This inspires her to constantly improve and evolve the capabilities of the website. She even likes to brag that she's the site's most prolific user. Her vision is to make Tailspin Toys the number one site for aircraft models and hobbyists. Needless to say, Paula is an informed and engaging Product Owner who is available when necessary and has the authority to make the necessary decisions. Paula has been using Scrum for about three years. She has been through the Professional Scrum Foundations and Professional Product Owner training.

The Scrum Master

The Scrum Master enacts the Scrum values, practices, and rules throughout the Scrum Team and even the organization. He or she ensures that the Product Owner and Development Team are functional and productive by providing necessary guidance and support. The Scrum Master is also responsible for ensuring that Scrum is understood by all involved parties and that everyone plays by the rules.

Note The Scrum Master is not a project manager. He or she is considered a manager, but of Scrum itself, not the project, the people, or the product.

The Scrum Master must be resolute in holding fast to the rules of Scrum, giving the organization time to normalize and realize the benefits. This means keeping any old "waterfall" habits at bay. It also means keeping any unenlightened managers at bay, while continually quashing the illusion that command and control and opaqueness equates to better and faster software development. Sometimes the Scrum Master may become the de facto change agent, leading the effort for an organizational adoption of Scrum. If this is the case, then the Scrum Master's steadfastness must be able to scale!

The Scrum Master has a softer side too. He or she can be called upon to act as a coach, ensuring that the team is self-organizing, functional, and productive and shielding them from external conflicts while removing any impediments to their progress. The ability of the Scrum Master to serve the team by removing impediments to their success is a vital piece of Scrum. As a *servant leader*, the Scrum Master achieves results by giving priority attention to the needs of the team. Scrum Masters may also be of service to stakeholders and others in the organization, helping them understand the Scrum framework and expectations from the various players. Servant leaders are often seen as humble stewards of the people and processes in which they are involved. By having a "What can I do for you today?" attitude, it fosters an environment of collaboration and respect, providing fertile soil for a high-performance Scrum Team. Lao Tzu, the ancient Chinese philosopher, said it best:

When the master governs, the people are hardly aware that he exists. Next best is a leader who is loved. Next, one who is feared. The worst is one who is despised. If you don't trust people, you make them untrustworthy. The master doesn't talk, he acts. When his work is done, the people say, "Amazing: we did it, all by ourselves!"

The Scrum Master is not a technical role. Having a strong background in software development is not necessary, though it can be helpful at times. Scrum Masters must really know Scrum. That's not negotiable. A good Scrum Master will also have good communication and interpersonal skills. He or she may have to facilitate interactions with other team members or enable cooperation across roles or functions. It's important to have those abilities. Keep this in mind when considering who might make a good Scrum Master. Table 1-3 lists the ways in which the Scrum Master serves the Development Team.

Tip In my opinion, traditional project managers *don't* make good Scrum Masters. Unfortunately, this is a common reflex for an organization adopting Scrum. For example, the decision makers decide to send "Roger," their PMI-certified, *Henry Laurence Gantt* medal recipient (look it up), Microsoft Project MVP to Professional Scrum Master training. The expectation is that Roger will lead the change. What I've seen happen is that either Roger's project management "muscle memory" adversely affects the adoption of Scrum, or his old colleagues and managers do.

TABLE 1-3 Ways the Scrum Master serves the Development Team.

Service	When
Help facilitate Scrum events.	During the Sprint as needed.
Identify, document, and remove impediments.	During the Sprint as needed.
Provide training, coaching, and motivation.	During the Sprint as needed.
Coach the Development Team on self-organization.	During the Sprint as needed
Attend required meetings on the Development Team's behalf.	During the Sprint as needed
Be the Development Team's emissary to the organization.	During the Sprint as needed
Shield the Development Team from interruption and noise.	During the Sprint as needed.
Be relied upon less and less.	Over time as the team improves.

The duties of the Scrum Master may not require a full-time commitment. High-performance teams recognize this and may select a Development Team member to play the part-time role of Scrum Master. This role may rotate between developers over time. Full-time Scrum Masters may get folded back into the Development Team, or part-time Scrum Masters may start getting busier as new Scrum Teams emerge in the organization. The Scrum Master role is more flexible than the other roles in this regard. So long as a Scrum Team understands and follows the rules of Scrum and has access to someone who can perform the duties of a Scrum Master when needed, party on.

Tip The skills of a Scrum Master are unique and important. Being a Scrum Master is a career choice for some. In my experience, they tend to be high-performance and continuously improve their skills as they serve the team. These Scrum Masters should remain just that. If possible, they shouldn't be dismissed or converted to another role. They will bring more value to the team and the organization as a full-time Scrum Master.

Tailspin Toys case study Scott was hired by Tailspin Toys last year to serve as Scrum Master. Initially, he only served the web application team, providing the necessary coaching in order to transform them into a high-performance Scrum Team. Upper management plans on using Scott to help other teams within the organization learn and adopt Scrum. Scott is an expert in Scrum and has years of practical, hands-on experience with various companies and teams. He has been through Professional Scrum Foundations and Professional Scrum Master training and is active in the Scrum.org community.

Stakeholders

Although not an officially defined role in the *Scrum Guide*, stakeholders include everyone else involved or interested in the development of the software product. Stakeholders can consist of managers, executives, analysts, domain experts, members from other teams, customers, and users of the software. Stakeholders are very important. They represent the necessity for the software. They also drive the vision and usability of the product by influencing the Product Backlog. Without stakeholders, who would use the software, pay for its development, or derive benefit from it?

In my experience, developers have a tendency to discount non-technical individuals. This is unfortunate. Stakeholders should not be ignored. That said, some stakeholders can take too much interest in the development effort and its status, becoming a distraction. Scrum has clear delineations of when stakeholders and the Development Team can interact, and it's very limited, as you can see in Table 1-4. Inspecting and providing feedback on the product, such as requesting a feature, should be handled by the Product Owner. Inspecting and providing feedback on the development process, such as inquiring about status, should be handled by the Scrum Master. In other words, stakeholders should almost always be kept *out* of the development process.

Tip Burndown charts posted in a common area or on a web portal are a great way to keep stakeholders informed, which This keeps the interruptions of the Scrum Team to a minimum. If anyone has questions about the charts, the Scrum Master can educate them.

The Scrum Master should strive to keep stakeholders out of the various Scrum events, with the exception of the Sprint Review meeting. Stakeholders should not be involved in any planning or estimation meetings unless their domain expertise is required. Attendance to any event is by invitation of the Scrum Team only. Stakeholders should also not attend the Daily Scrum, as its purpose is to allow the Development Team to synchronize with each other on the upcoming work. Even the Product Owner's presence at this meeting is considered a distraction from its purpose.

TABLE 1-4 Development Team interactions with stakeholders.

Interaction	When
Answer any questions the Development Team might have about items in the Product Backlog (estimation, planning, etc.).	During the Sprint as needed.
Review the product Increment built during the Sprint and provide feedback to be captured in the Product Backlog.	Sprint Review.

Tailspin Toys case study The Tailspin Toys company has a rich history in aviation, both commercial and military. As founder of the company, Buzz brought with him many of his pilot buddies to serve as advisors. While they are not technical when it comes to software, they do have deep expertise in the domain of aviation, aircraft, models, and the community. In addition to these experts, there are a number of other stakeholders who provide feedback on the web application. Some of these are die-hard users of the software—affectionately called the Fans of Tailspin. Having previously been an executive of an airline, Buzz understands the importance of capturing user feedback. To that end, he insisted on setting up *wish@tailspintoys.com email address to receive email feedback*. These emails are routed to a support person who triages the content and works with Paula to add the item to the Product Backlog.

Scrum events

The Scrum framework uses events to structure the various workflows of incremental software development. Each event is time-boxed, which means that there is a fixed period of time to execute the activities within each event. Time-boxing ensures that an appropriate amount of time is spent planning without allowing waste in the planning process. Figure 1-2 illustrates how the events and related artifacts flow together.

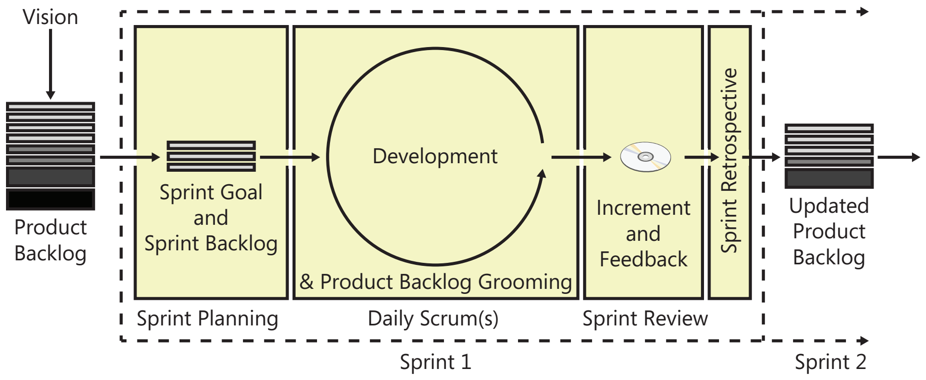


FIGURE 1-2 The sequence of Scrum events and related artifacts.

These Scrum events are meant to establish regularity and a cadence. They are also meant to minimize the need for wasteful or impromptu meetings that are not part of Scrum. All events are a formal opportunity to inspect and adapt something. Inspecting allows the team to assess progress toward a goal, as well as identify any variance in the current plan. If an inspection identifies any unacceptable deviation, an adjustment must be made to the product or process. These adjustments should be made as soon as possible to minimize further deviation. Failure to include or attend any of the Scrum events results in reduced transparency and is a lost opportunity to inspect and adapt. There are five prescribed events in Scrum:

- Sprint
- Sprint Planning meeting
- Daily Scrum
- Sprint Review meeting
- Sprint Retrospective meeting

Note The Sprint is not a meeting. It is a container for all of the other events. This means that the Sprint has begun when the Sprint Planning meeting commences. A notion exists that the Sprint is that time period after the Sprint Planning meeting and before the Sprint Review in which the actual development occurs. This is incorrect. Unfortunately, this "event" doesn't have a name. I refer to it as "development."

The Sprint

A Sprint is the set period of time in which an Increment of the software product is developed. A *Sprint* is Scrum's term for an iteration. Sprints are typically fixed at two to four weeks in length and run end to end, one after another. The frequency of feedback, experience of the team, and Product Owner's need for agility are key factors in determining the length of a Sprint. For example, if the software product is an enterprise desktop application with fairly well defined release goals, longer sprints are fine. If the application is software as a service (SaaS), with demanding customers and several competitors, shorter sprints would be more desirable. Both the customer and the Scrum Team need to collaborate to determine the ideal length of the Sprint.

In Scrum, the Sprint is the outer (container) event for the other four events. In other words, the Sprint Planning, development, Sprint Review, and Sprint Retrospective meetings all take place within the Sprint. This is a change from earlier Scrum guidance, which suggested that the Sprint began once Sprint Planning completed. Once you start using Scrum, you are always in a Sprint—assuming the software still requires development. When this Sprint's Retrospective meeting ends, the next Sprint begins and you repeat the inner events again. There should never be any breaks in between Sprints.

Sprint length I asked Ken Schwaber once how long a Sprint should be. His answer was, "As short as possible and no shorter." Sprints of longer than four weeks (one month) have a smell—the smell of water falling. When a Sprint's length is longer than a month, the definition of what is being built may change or complexity and risk may increase. By limiting the maximum length of a Sprint, at most one month of development effort would be wasted, rather than several months in a classic waterfall project. Conversely, Sprints with a length of less than one week are possible, but should be executed only by a high-performance Scrum Team. Even with very short Sprints, the overhead of the inner events must be factored in, leaving even less time for actual software development. Teams working in ”micro sprints” like these need to be on their A-game every day.

Ideally, the length of the Sprint does not change. If it must, it can only change in between Sprints, as a result of a decision made collaboratively during the prior Sprint’s retrospective meeting. Any change to the length of a Sprint will cause disruption to the Development Team’s cadence. This will correct over time, as will its Velocity.

Each Sprint is like a mini-project. The Sprint has a definition of *what* is to be developed. It also includes a flexible approach on *how* to develop it. During the Sprint, *all* aspects of the development work are executed. This will typically be more than just designing, coding, and testing. The scope of work may be clarified as more is learned, and the Product Owner may collaborate with the Development Team to renegotiate adding new items or swapping different items in the Sprint Backlog. The Development Team may not decrease any quality goals in order to finish its work. The resulting product Increment is produced and (hopefully) accepted by the Product Owner, who may also decide to release the Increment to production.

The choice of which day of the week to start (and end) a Sprint is entirely up to the Scrum Team. Some practitioners prefer Mondays or Fridays. Most don’t. Fellow professional Scrum developer Jose Luis Soria Teruel cautions against teams that try to always start a Sprint on a given day. The team can inadvertently give the day more importance than having a fixed Sprint length. For example, if a holiday falls in the middle of a Sprint, the team might shorten the Sprint so they can stick with it beginning on a Monday. Changing the Sprint length, even by a day, can affect cadence, Velocity, and the ability to achieve the Sprint Goal.

Canceling a Sprint Rarely does a Sprint need to be canceled, but it does happen. If a Sprint’s forecasted work becomes irrelevant, then there is no reason to continue developing it. This can occur if the product or organization needs to change direction immediately due to a technology or market reason. Only the Product Owner has the authority to cancel a Sprint. He or she may do so under the advisement of others, including stakeholders, the Development Team, or the Scrum Master. Canceled Sprints require the Scrum Team to collaborate and decide if any done work is acceptable and potentially releasable. The Scrum Team should also re-estimate any undone work, returning it to the Product Backlog. The work done on partially completed PBIs depreciates quickly and may not have any value in the future. Needless to say, canceling a Sprint will generate waste.

Tailspin Toys case study Originally, the Scrum Team tried four-week Sprints. They felt that the longer time-box would be closer to the quarterly delivery schedule they had been accustomed to. Unfortunately, since the team was new to Agile, they continued to take a sequential approach to development. They spent a lot of time on analysis and design at the beginning of the Sprint and deferred QA until the end. The resulting high-intensity crunch in the last few days of the Sprint was not sustainable and was really just a backslide into waterfall habits (a.k.a. “Scrummerfall”). The team did not experience the productivity gains everyone anticipated. When they hired Scott (the Scrum Master), he recommended moving to two-week Sprints. This caused the developers to experience a sense of urgency, change the way they worked, and maintain a comfortable level of intensity throughout the Sprint. Scott also recommended starting the Sprint on a Wednesday. This increased the chances of the whole team being in the office and operating at peak capacity. It also allowed stakeholders to fly in for a Sprint Review and the subsequent Sprint Planning meeting without having to stay over a weekend. The Scrum Team has completed many successful Sprints while on this two-week cadence. Their average Velocity over the last six Sprints is 22.

Sprint Planning meeting

The Sprint Planning meeting is for identifying and planning the development work that will be performed during the Sprint. This is the first event that occurs within the Sprint, and the most important. The entire Scrum Team attends this meeting. The Development Team collaborates with the Product Owner on the scope of work that can be accomplished. A groomed and ordered (prioritized) Product Backlog is required as an input for Sprint Planning. This forecasted work, along with a Sprint Goal and a plan for doing the work (the Sprint Backlog), are the outputs.

The Sprint Planning meeting is time-boxed, so everyone needs to be laser-focused. Distractions, such as non-topical conversations, should be minimized. The length of the Sprint Planning meeting is a function of the length of the Sprint, as you can see in Table 1-5.

TABLE 1-5 Length of the Sprint Planning meeting.

Sprint length	Sprint Planning meeting length
4 weeks	No longer than 8 hours
3 weeks	No longer than 6 hours
2 weeks	No longer than 4 hours
1 week	No longer than 2 hours
Less than a week	In proportion to the above lengths

The forecast During Sprint Planning, the Development Team considers the highest-ordered PBIs from the Product Backlog one at a time. The order is decided by the Product Owner. Each item’s requirements and acceptance criteria are discussed. Clarification is provided by the Product Owner as well as other domain experts who might be invited to the meeting. After obtaining a sufficient understanding of the PBI, the Development Team estimates the effort. If the consensus believes that they can deliver the item in this Sprint, the item is added to the forecast. Lack of consensus may require the PBI to be split or deferred until a later Sprint, when more is known. The Development Team moves to the next item in the Product Backlog. This is repeated until the Development Team thinks that they have forecasted a comfortable amount of work for the Sprint, given their capacity and past performance. These forecasted PBIs are moved from the Product Backlog to the Sprint Backlog.

The Development Team may use their Velocity to make the determination of what is an acceptable amount of work. New Development Teams, who don't yet have a normalized Velocity, as well as high-performance teams, may just use their instinct to decide what *feels* like the right amount of work. If the Development Team completes their forecasted work early, they can collaborate with the Product Owner mid-Sprint to identify and develop an additional PBI. Because of this, their Velocity may go up, and a larger forecast might occur at the next Sprint Planning meeting. The Development Team should never forecast more work than they *know* they can complete.

Note In 2011, the *Scrum Guide* introduced a somewhat controversial change to Sprint Planning. The word "commit" was replaced with "forecast". Scrum practitioners had an issue with the word *commit* for some time. The problem was that "commit" implied that the Development Team was obligated to deliver the PBIs at the end of the Sprint. This was especially true when stakeholders, who tend to not understand the complexities of developing software, heard the word. Since software development is very difficult and full of risk, delivering all PBIs every Sprint is unrealistic. The Development Team might have to cut quality in order to make good on their promise and this is essentially forbidden in Scrum. The term "forecast" is more realistic and easier to understand by business stakeholders who have heard terms like "sales forecast." It suggests that, while the Development Team will do their best, given what they know, new information will emerge during the Sprint that might impede their best-laid plans. It will take some time to get used to the new term. It may sound like a weasel word to some, but in the long run, its usage will be deemed more honest and transparent.

The Sprint Goal After the Development Team forecasts the PBIs that it thinks that it can develop in the Sprint, they should collaborate with the Product Owner to craft a *Sprint Goal*. The Sprint Goal is an objective, in narrative format, that guides the Development Team as they develop the Increment. The Sprint Goal also provides stakeholders the ability to see a synopsis of what the Development Team is working on. While the Development Team only *forecasts* the individual PBIs to be implemented, they actually *commit* to achieving the Sprint Goal.

Note Some teams like to craft the Sprint Goal first, or at least in parallel with the forecasting of work. This way, there is more cohesion with the goal and the PBIs that are developed during the Sprint. This cohesion makes it easier to understand the value of the Increment and how it fits into the goals of the product or release. This approach can be difficult for teams who need to develop disparate features and bug fixes for a given Sprint.

It's important that the Product Owner and Development Team craft the Sprint Goal together and agree on its verbiage and meaning. Everyone on the team should then commit it to memory. Stakeholders should have access to see it as well. Once development has begun (that is, the Sprint Planning meeting is over), the Sprint Goal should not be changed. It is the *theme* that the team has committed to, and the T-shirts have already been printed—so to speak. If the Development Team isn't able to achieve the Sprint Goal, or the goal becomes obsolete, the Product Owner might decide to cancel the Sprint—another indication of the Sprint Goal's importance.

The Sprint Goal gives the Development Team some flexibility and guidance regarding the functionality implemented within the Sprint. Even if the Development Team delivers less PBIs than were forecasted in Sprint Planning, they can still achieve their Sprint Goal. For example, let's assume the Development Team forecasts the following PBIs during Sprint Planning:

1. Add a Twitter feed to the homepage.
2. Create a Facebook page for the company.
3. Create and host a wiki page for product support.

Given this forecast, the Sprint Goal might read, "To increase community awareness of our company and its products." As the developers work, they keep this goal in mind. If the team is unable to finish the third PBI, they didn't fail because they were still able *to increase community awareness of our company and its products* by successfully completing the first two PBIs. If it sounds like Sprint Goals give the Development Team "wiggle room," you are correct. Remember that what developers do is very difficult and full of risk. That's why they should *forecast* the individual items they think they can deliver, but *commit* to the goal that embodies them.

The plan Sprint Planning is not complete until the Development Team has devised a plan for how they will develop the forecasted PBIs. The plan must ensure that all PBI acceptance criteria are satisfied while meeting the team's Definition of "Done." The plan gets added to the Sprint Backlog. On a whiteboard, this might be visualized as a collection of sticky notes in the same row as the associated PBI sticky note. In software, it might be several child records related to a parent record. Regardless of the tool the team uses, the Sprint Backlog contains both the forecasted PBIs and the plan (tasks) to develop them.

Tip Go lightweight during Sprint Planning. Whiteboards are a great medium for sketching ideas and brainstorming tasks. Laptops aren't. Whiteboards can be easily photographed and wiped clean after the meeting. Files on laptops tend to linger and yearn to be updated. They also indicate a finality set in stone that is not necessarily the truth.
Using sticky notes to brainstorm tasks in the plan is also good. They can be moved and removed easily from the board. A high-performance Scrum Team will avoid using any software during Sprint Planning unless its value outweighs its distraction. Sticky notes and whiteboard sketches can be translated into digital files later, once the Development Team agrees on the plan.

Because of the meeting's time-box, the Development Team probably won't be able to identify every task required to develop a particular PBI. For expediency, a minimum amount of information should be recorded—perhaps just a title and estimate of effort. Sprint Planning is not the time for detailed design. The Development Team needs to focus on the high-level plan and its tasks.

For example, let's assume that the team will have to create several database tables, stored procedures, and related data access code. Rather than go down the design "rat hole" during the meeting, the team should just identify a couple of high-level tasks: *create database objects* and *create data-access code*. Each of these would include an aggregate estimate of effort to perform all the related activities.

The tasks to be performed first in the Sprint should be decomposed as necessary so that no executable task is larger than can be achieved in one day. Estimates can be in whatever unit of measure the Development Team decides. For tasks, hours are the most common unit. I've seen teams also use days or story points. Personally, I think using story points for estimating tasks can lead to confusion. Rarely would you want to relatively compare the estimations of two tasks that could end up being done by different team members. Regardless of the unit of measure, all of these numeric values will enable a Sprint burndown chart, should the team choose to employ one.

It's important for the Development Team to leave the Sprint Planning meeting with a plan to accomplish the Sprint Goal. This plan should be documented, in the Sprint Backlog, in a way that the Product Owner and Scrum Master can understand the approach. Task ownership is not a required outcome of the Sprint Planning meeting. In fact, it's important to leave "to do" tasks unassigned so that team members who have capacity can pick a relevant task to work on next. That said, it is fine if the team decides to assign one or a few tasks to individuals by the end of the meeting. The Development Team will then self-organize to undertake the work in the Sprint Backlog as needed throughout the Sprint. Table 1-6 lists the activities expected of a Development Team during the Sprint Planning meeting.

TABLE 1-6 Development Team activities during Sprint Planning.

Activity	Where is it captured?
Forecast PBIs to be delivered that Sprint.	PBIs in the Sprint Backlog.
Collaborate with Product Owner to craft a Sprint Goal.	Whiteboard, sticky notes, Microsoft SharePoint, etc.
Develop a plan for delivering the forecasted PBIs.	Tasks in the Sprint Backlog.

Tailspin Toys case study The first Sprint Planning sessions were chaotic. The Development Team were introduced to new PBIs for the first time *at the meeting*. Paula (the Product Owner) wasn't always prepared and the domain experts were sometimes unavailable. Most of the meeting was spent understanding *what* was to be developed, and planning the *how* got deferred until the first few days of the Sprint. This corrected itself over time, as the team members got used to Scrum. Sprint Planning also became much more efficient when the team started meeting regularly to groom the Product Backlog.

The Daily Scrum

The Daily Scrum is a 15-minute, time-boxed meeting for the Development Team to synchronize their activities and create a plan for the next 24 hours. It allows developers to listen to what other developers have done and are about to do. This leads to increased collaboration, as well as accountability. If one developer hears that another developer is about to work in a similar area of the product, they may choose to pair up for the day. On the other hand, if the team hears that a developer is on day 3 of a 4-hour task, it may be time to pair up or inquire about the root cause. Team members need to understand that commitments are being made at this meeting and that these commitments will be tested 24 hours from now.

Note I hear a lot of teams refer to this event as the “daily standup.” The event is called the “Daily Scrum.” If the team decides to stand during the meeting, they may do so.

The most popular technique that Development Teams use during the Daily Scrum is to stand in a circle facing each other. Each developer, in turn, answers the following three questions:

1. What have I done since the last Scrum?
2. What will I do between now and the next Scrum?
3. What impediments are in my way?

The Development Team can use the dialogue heard during the Scrum to assess their progress. By hearing what is or isn’t being accomplished each day, the team can determine if they are on their way to achieving the Sprint Goal. As teams improve in their collaboration, this vibe will become more noticeable—even outside the Daily Scrum. High-performance teams may even outgrow the need for a formal assessment tool, such as a Sprint burndown chart. Stakeholders will only outgrow this need once the Scrum Team has earned their trust, which takes time. The sustained increase of business value being added to the software product should serve as its own assessment.

The meeting should be held in the same place and at the same time every day to reduce complexity and to maximize the likelihood of attendance. Ideally, the meeting is held in the morning so that the Development Team is able to synchronize their work that day. The Daily Scrum is not a status meeting. Problem solving can occur in the meeting, but it is usually deferred to just after the Daily Scrum because the problem solving can often lead to the team violating the 15-minute time-box for the event, as well as conversations that are not relevant to all attendees.

The Daily Scrum is not meant to be attended by anyone other than the members of the Development Team. This includes the Product Owner. In fact, the Scrum Master is not even required to attend. He or she just needs to ensure that the Scrum takes place and that the rules are followed. Any impediments can be identified, tracked, and even mitigated by the Development Team members.

Tip Keep laptops, burndown charts, and other artifacts and props out of the Daily Scrum. These tend to distract from the purpose of the meeting. Each developer should know their own information without having to look anything up. Observations and impediments can be recorded on a whiteboard or using sticky notes. High-performance teams will use a “parking lot” to track anything not relevant to the Scrum, and a follow-up meeting can support those conversations. The Development Team is self-organizing and can decide to meet formally or informally at any time during the day for any reason. The Scrum framework has no guidance on what the Development Team does the other 7 hours and 45 minutes of the day, other than to say that the Development Team should be maximizing their self-organization capability.

Tailspin Toys case study The Development Team has their Daily Scrum at 9 A.M. in the hallway near their team's area. Prior to the meeting, each developer updates their work remaining estimates on their tasks. By doing this, it gives them a fresh perspective on their remaining work and enriches the conversation. A side benefit is that this keeps the burndown reports accurate, which is good if they are consulted at any follow-up meeting. During the Scrum itself, the developers have adopted the practice of tossing a small rugby ball (a "talking stick") to the next developer to speak. Sticky notes are created and placed in a parking lot section of a nearby whiteboard as needed. The Daily Scrum usually takes less than 10 minutes.

Sprint Review meeting

After the Sprint's development time-box has expired, a Sprint Review meeting is held. The entire Scrum Team attends, as well as any stakeholders the Product Owner invites. This informal meeting is for inspecting the increment developed by the team. Stakeholders get to observe an informal demonstration of the working software. Their feedback is elicited and captured. This collaboration can produce new, updated, or removed PBIs.

The Sprint Review meeting is time-boxed. Its length is half that of the Sprint Planning meeting, or 1 hour for every week in the Sprint, as you can see in Table 1-7.

TABLE 1-7 Length of the Sprint Review meeting.

Sprint length	Sprint Review meeting length
4 weeks	No longer than 4 hours
3 weeks	No longer than 3 hours
2 weeks	No longer than 2 hours
1 week	No longer than 1 hour
Less than a week	In proportion to the above lengths

During the Sprint Review, the Sprint Goal and forecasted PBIs should be restated. Keeping their audience in mind, the Development Team may give a short summary about what went well, what didn't, and how they overcame any problems. If applicable, completed PBIs are demonstrated by running the working software, not by showing slides, mockups, or passing tests. Techniques can be employed to provide context and value. For example, the demonstrators might role-play the personas that would be using and benefiting from a particular feature being demonstrated. The Development Team describes what the attendees are seeing and, if necessary, how it works behind the scenes. They will also answer any questions the stakeholders might have.

Tip The Development Team should never surprise their Product Owner at a Sprint Review meeting. This should not be the first time that he or she sees the completed work. High-performance Scrum Teams know the value of continuous collaboration with the Product Owner. At a minimum, the Development Team should ask the Product Owner's opinion on individual PBIs as they approach completion. Product Owner acceptance doesn't have to wait until the Sprint Review meeting. In fact, you don't want Sprint Reviews to become "sign-off" meetings. They are more about improving the product through inspection of the Increment.

The Sprint Review meeting can generate one or more outcomes:

- Unfinished or unstarted PBIs are moved back to the Product Backlog.
- New feature ideas are added to the Product Backlog.
- Unnecessary items are removed from the Product Backlog.
- The Product Backlog is groomed.
- The Increment is released ("Ship it!").
- Product development is canceled.

As previously mentioned, the Sprint Review is an informal meeting. The Development Team should not spend much time preparing for it. Nobody should feel like they are attending a technical presentation at a conference. On the other hand, the team should be organized enough so it doesn't waste the stakeholders' time. If necessary, the Scrum Master can intervene and make corrections to maximize the meeting's value for everyone. Any corrections can be discussed at the Sprint Retrospective meeting and implemented in the next Sprint.

There are many ways to run a Sprint Review. Some Scrum Teams like it to be structured. Others don't. Some like the Scrum Master to kick it off. Others like it to be the Product Owner. Some like to rotate developers so everyone gets a chance to "drive" during the demonstration. Others like their strongest communicator driving. Regardless, the Sprint Review should be down to earth and foster an environment of collaboration and discussion. The Scrum Team should be inquisitive, and all feedback should be welcomed and captured, preferably in the Product Backlog. Later, the Product Owner can provide feedback on any of the captured PBIs regarding business value—or not. Inane ideas will eventually sink to the depths of the Product Backlog.

Being mindful of the time-box, unfinished or unstarted PBIs can also be discussed with the stakeholders. If they have blocked time out of their busy day, don't squander the opportunity to get their feedback on any PBI that might be coming up in an approaching Sprint. These discussions can create valuable input for the next Sprint Planning meeting.

Tailspin Toys case study Sprint Reviews have always been a big deal for the Scrum Team. They meet every other Tuesday morning in the large conference room and invite all of the stakeholders and even members from other teams. Paula (the Product Owner) kicks off the meeting with a review of the Sprint Goal and forecasted work. Scott (the Scrum Master) then gives a summary of the Sprint, including the team's progress (using the Sprint burndown chart), any obstacles, and how the Development Team overcame them. The bulk of the two-hour meeting is spent by the Development Team demonstrating the completed functionality. They do so in a storytelling way, with the developers playing different personas as they act out the user stories. This fun approach makes everyone in the room feel safe and comfortable in sharing their opinions and ideas. Scott or another team member captures this feedback in real time using Microsoft OneNote. Stakeholders also tend to send feedback in the form of an email after the meeting. This is captured using the product TeamCompanion (*www.teamcompanion.com*). Paula then wraps up the Sprint Review by discussing the forecasted items that didn't get finished or started, as well as her ideas for the next Sprint. Paula may also update everyone present on progress toward a goal via a release burndown chart or other tool.

Sprint Retrospective meeting

The last event in the Sprint is the Sprint Retrospective meeting. In this meeting, the Scrum Team will inspect and adapt its own behaviors and practices, looking for opportunities to improve. The Sprint Retrospective meeting occurs after the Sprint Review meeting and before the next Sprint Planning meeting. The exact time and location are up to the Scrum Team. It's important for the Product Owner, Scrum Master, and the entire Development Team to attend. The Sprint Retrospective meeting is time-boxed, as you can see in Table 1-8.

TABLE 1-8 Length of the Sprint Retrospective meeting.

Sprint length	Sprint Retrospective meeting length
4 weeks	No longer than 3 hours
3 weeks	No longer than 2 1/4 hours
2 weeks	No longer than 1 1/2 hours
1 week	No longer than 3/4 hour
Less than a week	In proportion to the above lengths

The purpose of the Sprint Retrospective meeting is for everyone to share their observations, thoughts, and ideas on what went well and what didn't with regard to people, relationships, process, and tools. These discussions can get heated, especially when you are talking about social interaction problems with other people. The meeting should be constructive and it's the Scrum Master's responsibility to keep it that way.

Note Impediments and struggles with the development process and practices can be inspected and adapted at any time, such as during the Daily Scrum or throughout the day or Sprint. The Sprint Retrospective meeting provides a *formal* opportunity for such inspection, as well as time for planning any adaptations.

The output of a Scrum Retrospective is a plan for implementing improvements. These improvements can target the development process as a whole or individual practices within it.

Improvements might include changing the way the Development Team works, or where, or when. Improvements might also include changing the way the developers use their tools, or what tools they use. Improvements might be more aesthetic, such as ways to make the work more enjoyable by making the work area more or less stimulating. Any potential improvement is really just an experiment, since the Scrum Team constantly inspects and adapts its practices. Table 1-9 lists some other changes that the Scrum Team is allowed to make during the Sprint Retrospective or in between Sprints. Some of these changes can be pretty major, so they should be executed only with the consensus of the full Scrum Team and a complete understanding of the ramifications of making the change. Any change made must still abide by the rules of Scrum.

TABLE 1-9 Changes that can be made at the Sprint Retrospective meeting or in between Sprints.

Change	Examples
Increase product quality by updating the Definition of "Done."	Increase the minimum code coverage percentage.
Change the person playing the Scrum Master role.	Relieve Scott of his duty while attributing the role to Dave.
Change the team composition.	Add another developer or drop Wade's capacity to 50%.
Change the Sprint length.	Change from two weeks to one week to increase agility.

Tip Don't be flaccid. Don't just hold the Sprint Retrospective meeting for the sake of the meeting. If problems are identified, make sure solutions are also identified. If solutions are identified, make sure they are actually implemented in the upcoming Sprints. Inspect *and* adapt!

There are many techniques that a Scrum Team can use during a Sprint Retrospective meeting. The most common is to have each Scrum Team member answer three questions:

- What did we do well this Sprint?
- What could we have done better?
- What will we try to do better next Sprint?

There are other approaches to start the conversation, elicit feedback, and brainstorm solutions. Entire books and websites have been devoted to running successful retrospectives and related techniques. Table 1-10 lists some of the techniques that my fellow professional Scrum developers have employed successfully. You will have to search the web for additional information, such as the instructions for using the technique.

TABLE 1-10 Sprint Retrospective meeting techniques and activities.

Technique	Description
Timeline	A timeline for the Sprint is marked on a wall, and team members add sticky notes to it to indicate good and bad events that occurred at that point in time.
Emotional Seismograph	Similar to the timeline, but team members mark their emotional level as a point on a *Y*-axis throughout the Sprint.

Technique	Description
Mad, Sad, Glad	Team members brainstorm on the events that made them mad, sad, or glad during the Sprint. Sticky notes are clustered together, normalized, discussed, and mitigated as necessary.
The 4 L's	Create four posters or whiteboards, one for Liked, Learned, Lacked, and Long For. Team members add sticky notes to the respective board. They are clustered, discussed, and mitigated as necessary.
The 5 Why's	A question-asking technique used to explore the cause-and-effect relationships underlying a particular problem.
Remember the Future	Used to create a vision of what the team wants to achieve by inquiring about a future point in time that follows another future point in time where the hypothetical change was made.
Car Speeding Toward Abyss	Draw a picture of a speeding car heading towards an abyss and use this analogy to identify the engine, parachute, abyss, and bridge comparisons to the current Sprint's work. The *Speedboat* and *Sailboat* are variations on this technique.
Happiness Metric	Similar to the emotional seismograph, but team members track their happiness levels throughout the Sprint using a scale of 1–5 with comments. A chart is produced for the Sprint Retrospective meeting and the peaks and valleys are discussed.
Perfection Game	A technique used to maximize the value of ideas. Team members rate an idea from 1–10 and provide positive feedback on how to make it a 10. No feedback means they've given it a 10.
Fishbowl	Arranging chairs in an inner and outer circle in order to attract team members to an empty chair in the inner circle (the fishbowl) and participate in the conversation.
Starfish	Using a starfish diagram, team members add sticky notes in these categories: do the same (=), do less of (<), stop doing (-), start doing (+), do more of (>). They are normalized, discussed, and mitigated as necessary.
Problem Tree Diagram, or *Ishikawa (Fishbone) Diagram*	A technique for visualizing the cause-and-effect relationships pertaining to a particular problem.
Team Radar	The team defines the factors (that is, communication, feedback, collaboration, etc.) and then each team member rates their interpretation of that factor on a scale of 0–10, where 0 means not at all and 10 means as much as possible. The chart is discussed and saved for later comparison.
Circles and *Soup*	A technique for helping identify what is and what is not the responsibility of the Scrum Team. This is similar to the *Circle of Concern and Circle of Influence* technique.

It's also important during the Sprint Retrospective to celebrate the team's victories. The good things that occurred should be encouraged to persist. Likewise, challenges in this Sprint should be seen as opportunities for victory in the next. This continuous improvement mentality is foundational in a high-performance Scrum Team. They live it every day. Since not every team member is wired this way, encouragement and team building are important and should be part of the retrospective too, if required. Everyone should see that the Development Team is more productive and happy.

Tailspin Toys case study In the early Sprints, the Retrospective meetings would not generate much return on the time invested. The entire Scrum Team would return to the large conference room after lunch and go through the basic questions. To them, it just felt like a longer version of the Daily Scrum and a waste of time. Retrospective notes were captured and the plan for improving was sometimes executed. When Scott joined the Scrum Team as Scrum Master, this changed. He introduced new techniques to get everyone involved. He focused on what went well and team building. He also ensured that any action items were implemented during the next Sprint. He called it his *Scrum Master backlog*. More important, he convinced Paula and Buzz to hold the Retrospective meeting in the back room at Fourth Coffee.

Product Backlog grooming

Maintaining a well-groomed Product Backlog helps the development of a successful product. Product Backlog grooming is the periodic meeting of the Product Owner and the Development Team to add detail to upcoming PBIs. This is the time when the requirements and acceptance criteria are explored and revised. When the Development Team has sufficient understanding of the PBI, they will estimate the effort required to develop it. This estimate may change over time, as more is learned about the item. In fact, the Development Team may re-groom and re-estimate the same PBI several times before it gets forecasted for development—usually as a result of new information.

Product Backlog grooming is a necessary and important part of Scrum. Although it is not a formal event, the *Scrum Guide* says that it is an ongoing process taking no longer than 10 percent of the capacity of the Development Team. The exact where and when of the Product Backlog grooming sessions are up to the Scrum Team. Some teams try to avoid doing a grooming near the very beginning or very end of the Sprint so that it doesn't collide with the other, more formal Scrum events, and closing out the Sprint. It is important to have the entire Development Team involved in grooming because the analysis and estimation will be more meaningful and accurate. Diligently grooming the Product Backlog minimizes the risk of developing the wrong product.

Tailspin Toys case study With the adoption of two-week Sprints, the Development Team now spends every Friday morning in a conference room with Paula for "story time"—a euphemism for Product Backlog grooming. All developers attend the meeting because each has valuable input and may be called on to collectively estimate the effort of the items being discussed. Because of these regular grooming sessions, Sprint Planning meetings have become more productive. The Scrum Team now spends less time forecasting because the most important PBIs and their estimates are fresh in their minds.

Scrum artifacts

Scrum's artifacts represent the work to be done in the product and Sprint, as well as the work that has been done within the product itself. Each artifact has clear ownership by a specific role. Each artifact is structured in a way that maximizes transparency of key information while providing opportunities for inspection and adaptation. There are three artifacts in Scrum:

- Product Backlog
- Sprint Backlog
- The Increment

Note Burndowns (product, release, and Sprint) were removed from the *Scrum Guide* in 2011. Their inclusion was considered too prescriptive. While it's important for the Scrum Team to monitor progress toward a goal, there are many practices that could support this. Burndowns are certainly a popular option and are still acceptable and used by some high-performance Scrum Teams. No technique will replace the importance of empiricism. In complex environments, such as software development, what will happen is unknown. The Scrum Team can only use what *has happened* to influence its decision making.

Product Backlog

The Product Backlog is an ordered list of everything required of the software product. It is the single source of requirements for any potential changes to be made. Each item in the Product Backlog is called a "*Product Backlog item (PBI)*." A PBI can be a *happy* thing that doesn't yet exist in the software product, like a feature or an enhancement. PBIs can also be *sad* things, like a bug to be fixed. PBIs can range from extremely important and urgent to silly and trivial. Because of this variety, I affectionately refer to the Product Backlog as a list of *desirements*. At some point, somebody, somewhere, for some reason *desired* each item in the Product Backlog.

Note The Product Backlog is a dynamic, living document. It is never complete and will constantly change as requirements change. The Product Backlog will exist so long as the software product exists.

These items are considered valid PBIs:

- Feature
- Enhancement
- Behavior
- User stories
- Use case
- Scenario
- Bug/defect

These items should not be PBIs:

- Task (that is, refactor code, write more tests, meet in the lobby for the Daily Scrum)
- Acceptance criterion (that is, page content in German and English, report exportable as PDF)
- Non-functional requirements (when they are used as acceptance criteria)

- Definition of “Done” (that is, code is peer-reviewed, code coverage > 50 percent, all tests pass)
- Impediment (that is, must reset my password on SQL Server, activate Windows)

Each PBI should be clearly identified by a title. This is the minimum amount of information required to add it to the Product Backlog. If the Product Owner decides it’s worth the time to describe it further, then a description should be added. This description should be written in a business language, perhaps as a user story description. The PBI should also be assigned a business value and ordered with the other items in the backlog. The Development Team will need to eventually look at it and provide an estimate. This can be done at a Product Backlog grooming session or during Sprint Planning. Table 1-11 lists the ways in which the Development Team interacts with the Product Backlog.

TABLE 1-11 Development Team interactions with the Product Backlog.

Activity	When
Inspect it.	Any time
Add a new PBI to it.	Any time (if allowed by the Product Owner)
Groom it.	Product Backlog grooming, Sprint Planning, or Sprint Review (with Product Owner)
Forecast work from it.	Sprint Planning (with Product Owner)

I’m often asked if being *responsible* for the Product Backlog means that the Product Owner has to be the person who actually creates the PBIs (that is, write the user stories). The answer is no. The Product Owner can have the Development Team or stakeholders, including business analysts and even the users themselves, create the PBIs. The Product Owner has the right to update any item, such as making it more understandable or changing acceptance criteria, or to remove any item deemed unnecessary. The Product Owner or Scrum Master may have to remind people that PBIs should only define the what, and not the how.

User stories A PBI represents a software requirement. It can take any number of shapes or forms. Of all that I have seen, the user story practice is generally the best choice for teams doing Agile software development. This is primarily because user stories are lightweight and *not* technical. User stories describe the requirement from the customer or user’s perspective. It is not a requirements document, nor is it a communiqué between the requirements giver and the Development Team. A user story represents a “what” that the software product should do. A well-written user story description will explain who wants or would benefit from the feature, as well as how and why it will be useful. In a single sentence, the user story provides lots of context, as well as a value proposition.

The most popular format of a user story description looks like this: *As a (role), I want (something), so that (benefit).* An example would be, “As a returning customer, I want to log in with my ID and password, so that I don’t have to enter my shipping and billing information each time I order a product.” Another example would be, “As a visitor to the Tailspin Toys website, I want to see a list of recent tweets, so that I know that Tailspin and its products are alive and well.” Anyone looking at either PBI instantly knows the context and value to the customer.

Having a title and the initial description in user story format is a good start. To properly complete a user story, communication between the Scrum Team and knowledgeable stakeholders is required. A complete user story includes the *three C's*: Card, Conversation, and Confirmation.

The *card* is already done at this point. You have written a title and the description (in user story format) on a sticky note, an index card, or a software record. This allows somebody to reference the user story during conversation, update it, estimate it, stack rank it, etc.

Next, the *conversation* takes place with the customers, users, or domain experts. This conversation is meant to exchange thoughts and opinions. It can take place at any time with the Product Owner and the stakeholders and the Development Team as needed. If the Development Team is to be involved, it should take place at the Product Backlog grooming session, the Sprint Planning meeting, or the Sprint Review meeting. Conversation that yields examples, especially executable and testable examples, is preferred over formal documents and mockups.

Finally, the *confirmation* occurs. Here the user story's acceptance criteria are agreed upon and recorded. These criteria will help determine when the PBI is done. In other words, when all criteria are met according to the team's Definition of "Done," the PBI is done. If and when the PBI gets forecasted for a Sprint, the Development Team will create the appropriate manual or automated acceptance tests to validate the acceptance criteria.

Tip Don't create tasks, tests, or code for a PBI before the Sprint in which you have forecasted its development. Conditions can change rapidly, forcing a change to the PBI or its acceptance criteria. Time spent creating these kinds of artifacts ahead of time will often be wasted. The plan on how to develop a PBI, as well as any code or tests, just like requirements, should be created at the latest responsible moment. Even though you will always know more tomorrow than today, you should avoid falling into the trap of doing things at the last *possible* moment.

Whoever creates a user story should be sure to INVEST in it. The mnemonic *INVEST* is a reminder of the characteristics of a good user story:

- **I–Independent** As much as possible, the story should stand alone, without any dependency on another story. Try to write stories such that they don't have long "dependency chains."
- **N–Negotiable** The story can be changed and rewritten up until it gets forecasted, but significant changes after being forecasted should be avoided and minimized. Minor tweaks are okay so long as they don't greatly affect the original estimate for the story.
- **V–Valuable** The story must deliver value to the customer or user. This value is often delivered in the graphical user interface (GUI), but not always.
- **E–Estimable** The Development Team must be able to estimate the effort to develop the story. If too little is known about the story, it will be difficult for the team to come to consensus on a story.

- **S–Small** The story must be small enough that the team can develop it in a single Sprint and preferably within a few days. There are many suitable techniques for decomposing stories.
- **T–Testable** The acceptance criteria is clearly understood and can be tested. This is probably the most important characteristic. It relates to the third "C" in the three "C's": confirmation.

Product Backlog iceberg You can think of the Product Backlog as an iceberg (see Figure 1-3). PBIs on the top, above the surface, are what the Development Team has forecasted for the current Sprint. These items should be crystal clear, estimated, and ready to be worked. Below the surface, the Product Owner knows what other PBIs he or she would like in the release, but it won't be clear which ones surface until the next Sprint Planning meeting. These items are generally understood and estimated so that a release plan can be devised. These are the items that will be in scope during upcoming Product Backlog grooming sessions. At the bottom of the iceberg, you will find all of the other PBIs that may or may not make it into a future release. Some of these may only have a title or a vague description of the desired functionality. Some PBIs will remain in these cold, chilly depths for eternity, which is typical of most Product Backlogs.

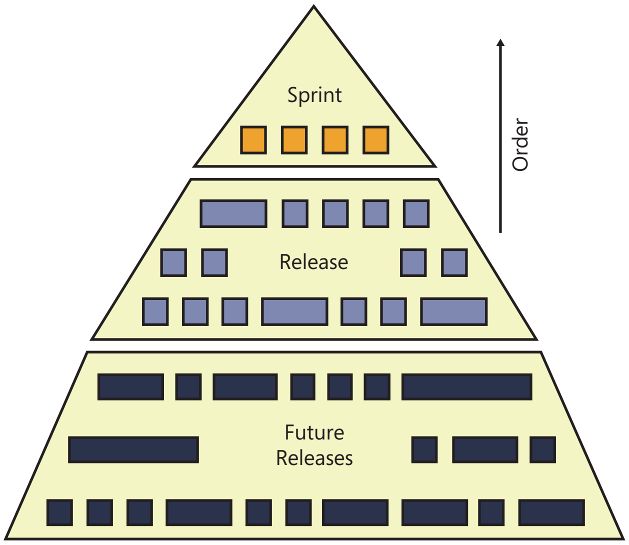


FIGURE 1-3 The Product Backlog iceberg.

Sometimes it's a chicken-and-egg problem when it comes to evolving a PBI. The Product Owner might need an estimate on the level of effort required to develop a PBI before he or she can order (prioritize) it. If it's going to require too much effort, the Product Owner may postpone it for the next release, or beyond. However, the Development Team's time is valuable and they shouldn't waste their time estimating PBIs that may not be developed. A solution I've seen work well is for the Development Team (or a proxy) to provide the Product Owner a rough order of magnitude estimate, such as a T-shirt size (XS, S, M, L, XL). This should give the Product Owner enough insight to be able to order (prioritize) the PBI effectively. A more thorough estimate, provided by the entire Development Team and using a more precise scale, will be performed at a future Product Backlog grooming session.

Note The Scrum Guide uses the term "order" instead of "prioritize". This subtle change has led to some confusion, which is why I've been using both terms together. Fellow professional Scrum developer Jose Luis Soria Teruel explains the difference eloquently. Assume that a Product Owner wants to have some software features as soon as possible, like the ability to sell products and accept payments (priority). However, before those features can be developed, other capabilities must be developed like the shopping cart feature (order).

The Product Owner is responsible for the Product Backlog, including the clarity and precision of its contents. He or she should also ensure that the Product Backlog is visible to all interested parties. The Product Owner will order (prioritize) the PBIs according to his goals for the product or release. The PBIs at the top of the ordered Product Backlog will, more than likely, be what the Development Team works on next. The Product Owner's vision should be discernible by studying the order and content of the PBIs. If necessary, the Scrum Master should help the Product Owner manage the Product Backlog more effectively.

Creating an effective Product Backlog can be very difficult. It can take a long time. It can become political. However, once you've gone through the exercise of creating the Product Backlog, you'll wonder how you ever got along without one.

Tailspin Toys case study Creating the initial Product Backlog *was* difficult. Requirements, feature requests, and bugs were tracked by different people in different formats. Giving up control of those lists started a turf war—but in the end, it was best for the product. When possible, all "happy" PBIs were converted to a user story format. Today, the Scrum Team maintains its Product Backlog in Team Foundation Server. The server administrator gave permissions to anyone on the Scrum Team to manage the Product Backlog. Everyone else can only view it. Paula (the Product Owner) is considering granting access to some additional stakeholders to help her create PBIs.

Sprint Backlog

The Sprint Backlog contains the Product Backlog items forecasted to be developed during the Sprint and the plan (tasks) for developing them. The PBIs were agreed upon and selected through collaboration of the Scrum Team. The plan for developing them was agreed upon and recorded through collaboration of the Development Team. The Sprint Backlog is the output of the Sprint Planning meeting and represents the Development Team's forecast of *what* functionality will be in the next software product Increment, and *how* it will happen. Some teams refer to the tasks as Sprint Backlog tasks (SBTs) or Sprint Backlog items (SBIs). Technically, the forecasted PBIs are also considered SBIs, so additional context will need to be provided when using that term in a conversation.

The Development Team owns the Sprint Backlog. This is to say that the Development Team is wholly responsible for how to implement the PBIs, so long as they do so according to the acceptance criteria and their Definition of "Done." Nobody can tell the Development Team how to develop the

Increment. In other words, nobody except the members of the Development Team can add, edit, or remove tasks from the Sprint Backlog. The Sprint Backlog should be kept up to date and visible to the Scrum Team. It provides a real-time picture of the work that the Development Team plans to accomplish during the Sprint.

Tip Increasing the Sprint Backlog's visibility beyond the Scrum Team is an invitation for the three "M's": meddling, misunderstanding, and micromanaging. Remember that the Sprint Backlog primarily contains the *how* and not the *what*. Allowing stakeholders, or any interested parties, to view the Product Backlog or burndown charts (if utilized) is preferable.

Table 1-12 lists the ways in which the Development Team interacts with the Sprint Backlog.

TABLE 1-12 Development Team interactions with the Sprint Backlog.

Activity	When
Inspect it.	Any time
Move a PBI from the Product Backlog into it.	Sprint Planning or any time afterward (with Product Owner collaboration)
Add, update, split, or remove a task in it.	Sprint Planning or any time afterward until Sprint Review
Take ownership of a new task in it.	Any time (as work demands)
Update status of a PBI or task in it.	Any time (as status changes)
Estimate work remaining for your tasks in it.	Daily

The entire Development Team should collaborate on the plan and create the tasks. Scrum Development Teams must be cross-functional for just this reason. Everyone can and should contribute. This will create a richer and more honest Sprint Backlog than if only one or two code gurus created the plan. A good approach is to start with a conversation in order to understand the PBI and discuss any potential plan. The plan can evolve onto sticky notes or a whiteboard, and then finally to records in a software application like Team Foundation Server. There's zero technical debt in a discussion, and close to zero in a set of sticky notes.

The Development Team must identify *all* tasks in the Sprint Backlog, not just the design, coding, and testing ones. There may be learning, installing, deploying, data entry, design meetings, and documenting tasks. The team's may indirectly require tasks to be created in the Sprint Backlog too. For example, a team's Definition of "Done" might require that every PBI implemented in the Increment has its own installer with notes and instructions in English and German. This self-imposed requirement could drive the creation of several additional tasks for each PBI in the Sprint Backlog.

Tip Have the team's Definition of "Done" nearby during Sprint Planning. It will help the developers as they brainstorm tasks. Also, depending on how the last Sprint went, there may be additional tasks related to improvements identified at the Retrospective meeting.

The developers should estimate their Sprint Backlog items at least daily. This can be done before or after the Daily Scrum, but not during. Most teams I work with prefer to re-estimate their tasks prior to the Daily Scrum, so that any follow-up meetings will have an accurate burndown chart to reference. Some high-performance Scrum Teams won't bother tracking hours or estimating remaining work on tasks. They focus on the Sprint Goal and delivering the PBIs, not the tasks. It is more difficult to assess progress without this information.

Note Scrum does not consider the time spent working on a task. Tracking actual hours is counterproductive to obtaining the Sprint Goal. I would even call it wasteful. If, however, an organization requires its employees to track their time to get paid, that's a separate discussion. The worry is that once such a metric is created, it would be used in a *command and control* way. For example, a manager might see that a set of UX design tasks took 28 hours and then use that as an estimate for future work, or as a stick to beat the designer with if her next set of tasks goes beyond that number—which it could, because software development is very difficult and full of risk.

The Sprint Backlog will be empty at the start of a Sprint. It will begin to emerge during Sprint Planning, and (ideally) be fully populated with tasks by the first few days of the Sprint. For teams new to Scrum or the product's domain, this can be unachievable. These teams may find themselves creating new tasks all the way through the Sprint. This makes it difficult to assess progress, if you don't know what the plan is or when you might achieve it. Even high-performance Scrum Teams need to change their plan sometimes. Each PBI introduces new complexities that can derail an execution plan. New tasks may have to be created mid-Sprint.

Tip In Scrum, work should never be directed or assigned. When creating a new Sprint Backlog task, don't assign it to anyone. For example, you should resist the urge to assign the *testing* tasks to Toni (even though she has a background in testing). Doing so will decrease collaboration and the opportunity for other team members to learn. When the time is right, the team should decide who will take on that task. The team will take many factors into account, including the background, experience, availability, and capacity of the developer.

As the Development Team improves, it will learn to manage risk better, by taking on riskier work early. The team will also become better at identifying the full spectrum of tasks, at least at a high level, during Sprint Planning. It's okay for the more distant tasks to be coarsely defined and overestimated. As the time nears for that piece of work to begin, the eligible developer can decompose and re-estimate it. If Sprint burndown charts are being used, they will be more accurate, earlier in the Sprint. The trend lines, which predict when the Development Team will be done with their work, will also be more accurate. Observers of the burndown charts need to understand that the Development Team will know more tomorrow than they did today—so expect change. The Scrum Master should be able to provide this education.

Tailspin Toys case study During Sprint Planning, the Development Team brainstorms the plan for developing the Increment. When they were just starting out with Scrum, they would only get one or two PBIs planned out and delay the planning of the rest of the PBIs until the Sprint. They've improved in the way they decompose and plan their SBTs. They estimate the tasks in hours, and they've improved the way they've done that. Originally, they would have the "experts" in the various task areas do the estimates. That made estimation go quicker, but during development, they would usually blow their estimates because the expert didn't always do the work. They now estimate the tasks collaboratively and find that they are under as many times as they are over. They can live with that.

The Increment

Scrum is an iterative and *incremental* software development framework. The word "incremental" means "occurring in especially small increments." Each Sprint is an especially small period of time during which the team develops one of these small increments. As we've already discussed, the small period of times (the Sprints) reduce risk by maximizing collaboration and feedback. Incremental delivery of a done software product ensures that a useful version of the working product is always available.

Tip If possible, make the Increment available to the Product Owner and stakeholders throughout the Sprint. Think of it as a hands-on demo or lab environment. As the Development Team finishes a PBI, the demo environment is updated for people to play with the software. This doesn't have to be any kind of a formal testing area, just something that can drive feedback during the Sprint, rather than waiting until the Sprint Review meeting. For example, it would be very convenient to be able to send an email to the stakeholders letting them know there's a "beta" hosted on *http://demoserver1/sprint6/tailspin*.

In Scrum, the Increment is the sum of all the PBIs completed during the Sprint plus all previous Sprints. It's the aggregate of what's currently running in production plus the done PBIs from previous Sprints that haven't yet been released, plus the done PBIs from the current Sprint. Only PBIs done according to their acceptance criteria and the team's Definition of "Done" can be added to the Increment and become potentially releasable.

Note Potentially releasable means that the Increment *could* be released (to the customer or production) if the Product Owner chooses to do so. This is possible because the Increment contains only done PBIs. PBIs aren't done until they meet the level of quality defined by the Product Owner and the Development Team according to the Definition of "Done". The Product Owner may decide to wait until several related PBIs are completed (release by feature), until a certain point in time (release by date), as each PBI is done (continuous deployment).

Definition of "Done"

The Definition of "Done" is not a formal artifact in Scrum, but it should be. Done is the state when a PBI has been developed according to its acceptance criteria and team's Definition of "Done." Scaling that up, done is also the state when the Increment containing all the done PBIs becomes potentially releasable.

The Definition of "Done" is a simple, auditable checklist created by the Development Team. It must be understandable by the Product Owner, the Scrum Master, and any stakeholders. This is why it must be simple and as free of "geek speak" as possible. The definition can be influenced by organizational, product, and release standards and constraints. For example, C# may be a language standard in the organization, but a specific product must be written in C++ for compatibility reasons. Here is a simple Definition of "Done":

- All code compiles without errors or warnings.
- No code analysis errors or warnings exist.
- New code is covered by unit tests.
- An automated build exists.
- An .msi installer exists.

Definitions of "Done" can be quite long and complex. Everything in the definition should be achievable, although some items may not be applicable. For example, if the Development Team is working on a PBI that is mostly graphic-design-centric, there won't be any code to unit-test. For all PBIs that have code, however, the team must create unit tests. It's in the definition. The Development Team should never cut corners by ignoring all or part of the definition in order to finish the forecast. The team has already unanimously decided that quality, as defined by the Definition of "Done", is more important than all-out speed.

Note The Definition of "Done" is a *minimum* standard. There may be times when the Development Team will want to do more than the minimum. This is acceptable so long as the extra effort is justified and not considered "gold plating." Gold plating is when a developer continues to work on a PBI beyond what is fit for purpose. This extra work is typically not worth the value that it adds to the software product.

Undone work

An explicit and concrete Definition of "Done" may seem small, but it can be the most critical checkpoint during a Sprint. Without a consistent meaning of "done," Velocity cannot be estimated. Having a shared Definition of "Done" ensures that the Increment produced at the end of Sprint is of high quality, with minimal defects. High-performance Scrum Teams consider the Definition of "Done" to be sacrosanct. It is the soul of their entire development process. These teams will resist the urge to release undone work, or even demonstrate it at a Sprint Review meeting.

The Development Team should not generate undone work. They should also make sure the “done” means completely done. In the long run, it will be cheaper to hold fast to the Definition of “Done” by improving development practices than to keep sprinting with an unknown amount of work still to be done at the end of the release. If the Product Owner looks at an Increment and doesn’t know how much work needs to be done, he or she won’t really know when the release will be ready. There may be a need for one or more “stabilization” Sprints at the end of the release just to tackle all of the accumulated undone work.

What’s even worse is that the undone work from the Sprints accumulates exponentially, not linearly. Subsequent Sprints will require even more work to reach done: 4 hours of undone work per Sprint for 6 Sprints won’t be 24 hours of work, but more like 80 hours. This “undone work” uncertainty has no place in a framework that is supposed to promote transparency and predictability, so every effort should be given to eliminate undone work and “stabilization” Sprints.

As the Development Team improves, it is expected that their Definition of “Done” will improve too. The definition can be changed only in between Sprints. The Sprint Retrospective meeting provides the opportunity to discuss and change it if necessary. The definition should only expand to include more stringent criteria for higher quality. In other words, you should avoid removing items from the definition in order to get more “done” the next Sprint.

The professional Scrum developer

The *Scrum Guide* does not provide guidance on *how* to develop a software product. In fact, during the time between the Sprint Planning meeting and the Sprint Review meeting, the guide is intentionally vague. Other than requiring a Daily Scrum meeting and regular Product Backlog grooming, not much guidance is provided. In fact, the rules state that a Daily Scrum should occur, taking no longer than 15 minutes.

So what about the other 7 hours and 45 minutes of the day? What should the Development Team, and the individual developers, be doing during that time? That’s the million dollar question. The short answer is: the developers should be doing the right thing—even when nobody is looking. There are many longer answers. The contents of this book will hopefully reveal several answers to this question.

Remember that developing software is a risky endeavor for both the developer and the customer. The process is a complex undertaking consisting of specifying, designing, coding, and testing. More things can go wrong than right. Any small mistake or fault on either side can lead to wasted effort—if you are lucky. Some mistakes can lead to outright damage. Professional Development Teams understand this, and they make sure their customer understands this. Ideally the customer will share in these risks. This means that the customer and the developers understand that they are both equally responsible for identifying and mitigating these risks, as well as sharing responsibility if a risk evolves into a disaster of some sort.

Let’s drop the customer out of the discussion for a minute. Developers on a Scrum Team collectively own their successes and failures, just as they collectively own the code, bugs, technical debt, and other issues. These developers have also learned to rely on their fellow team members and to trust them. They know that they must be resolute, forthright, transparent, and able to compromise in order to reach their goals. These qualities sound similar to those of the chivalrous knights from the Middle Ages —except for the compromising part.

When I'm meeting with a new team, I will often ask what they think the developer's job is. "To write code," is the almost universal flip answer that I hear. Being a career developer myself, I used to agree with that answer. As I've improved my understanding of the profession of software development, this answer now irks me. I believe that a better answer would that a developer's job is *to provide value in the form of working software*. This answer encapsulates the attributes of a professional Scrum developer. Professional Scrum developers understand that:

- They have the right and responsibility to maximize the self-organization capability of the team.
- They should reflect Scrum's values: commitment, focus, openness, respect, and courage.
- They should only do work that provides value to the software product.
- They should plan realistic goals and then commit to achieving them.
- They don't know everything, and they should be always willing to learn.
- They shouldn't be afraid of working outside their comfort zone.
- They should respect the Scrum Guide and its "rules."
- They shouldn't be afraid of asking other team members for help.
- They should be transparent in what they do and how they do it.
- They are part of a team, and their voice is equivalent to others.
- They have a stake in the success (or failure) of the product.
- They look for and minimize waste in their practices.
- They are responsible for the quality of the product.
- They should be honest in their estimates.
- They should say "no" when appropriate.
- They should collaborate when possible.
- They are professionals, not hobbyists.
- They shouldn't release undone work.
- They are more than just a coder.
- They are part of a larger team.

Chapter burndown

Here are the key concepts we covered in this chapter:

- ***Scrum Guide*** The *Scrum Guide* codifies the rules of Scrum. You should download it from *http://www.scrum.org/scrumguides* and read it now. Its updates will supersede this chapter.
- **The Development Team** The Development Team contains a cross-functional group of three to nine professionals who develop the forecasted work during the Sprint.
- **Product Owner** The Product Owner is the voice of the user and is responsible for maximizing the value of the product and work of the Development Team.
- **Scrum Master** The Scrum Master is responsible for ensuring Scrum is understood and enacted.
- **Sprint** A time-boxed event of one month or less that contains the other Scrum events.
- **Sprint Planning** The meeting where the Scrum Team forecasts the work to be performed during the Sprint, along with a plan for developing it.
- **Daily Scrum** The daily meeting allowing the Development Team to synchronize activities and create a plan for the next 24 hours.
- **Sprint Review** The meeting where the Increment is demonstrated and feedback is captured.
- **Sprint Retrospective** The meeting where the Scrum Team inspects its practices and creates a plan to improve in the next Sprint.
- **Product Backlog** An ordered list of everything that might be needed in the software product.
- **Sprint Backlog** The forecasted Product Backlog items plus the plan for developing them.
- **The Increment** The sum of all done Product Backlog items (PBIs) during this and previous Sprints.
- **Definition of "Done"(DoD)** A shared understanding of what it means for the Development Team to be done with the development of an individual Product Backlog item or the Increment itself.

CHAPTER 8

Effective collaboration

There's a buzz—a kind of energy that you can feel—when a high-performance Scrum Development Team works in harmony to solve a problem. Each developer gets totally absorbed in his or her task. Each member of the Development Team does his or her part integrating the design, the coding, and the testing. Scenarios and features are completed and verified. Product Backlog items (PBIs) are moved to the done column. Everyone loses track of time. They are experiencing *flow*. Everyone feels happy and satisfied.

Bruce Tuckman wrote about the stages of group development. He identified four stages in the development model: forming, storming, norming, and performing. In the initial, *forming* stage, the individuals come together to form the team. They may not know each other or everyone's strengths and weaknesses. This leads to the *storming* stage, where each developer competes for their idea's consideration while working together to resolve their differences. This necessary stage can sometimes be completed quickly. Unfortunately, some teams never leave this stage. Once the team members are able to resolve their differences and participate with one another more comfortably, they enter the *norming* phase. Here, the entity of the team begins to emerge. The members converge on a single goal and come up with a mutual plan. Compromise and consensus decision making occurs in this phase. High-performance Scrum Development Teams have reached the fourth and final phase, known as *performing*. These teams not only function as a unit, but they also find ways to get the job done smoothly and efficiently. They are able to self-organize and self-manage effectively. In my opinion, very few teams reach this phase, but every one that does has mastered the art of collaboration.

In this chapter, we will look at some practices and tools that enable more effective collaboration. By learning and adopting these practices, a team will increase its ability to reach the performing phase of Bruce Tuckman's model.

Individuals and interactions over processes and tools

The Agile Manifesto clearly states that while there is value in process and tools, there is *more* value in interacting with individuals. This is to say that Agile software development recognizes the importance of people and the value that they bring when working together. After all, it's people who build software, not the process or the tool. If you put bright, empowered, motivated people in a room with no process and inadequate tools, they will still be able to get something accomplished. Their Velocity may suffer, but they will produce value. They will also inspect and adapt their processes, while looking for methods of improvement. Conversely, if the people don't work well together, no process or tool will fix that. A bad process can screw up a good tool, but bad people can screw up everything.

Tip Fellow Professional Scrum Developer Simon Reindl reminds us that to err is human, but to forgive is vital.

Software development is a team sport. To succeed in this sport, game after game, the team must share the vision, divide the work, and learn from each other. In other words, they must collaborate. Even a team of expert craftsmen (rock stars in their own right) is doomed to fail if they don't collaborate with each other. If the striker on a soccer team has his best game ever—scoring four goals—but the other team scores five goals, it is still a loss. The other team, with even mediocre players, probably collaborated better.

A few years ago, Ken Schwaber did a series of podcasts where he answered frequently asked questions about Scrum. My favorite question that he answered was, "Do I need very good developers for Scrum?" His answer was insightful: "You need very good developers for software development. You can do Scrum with terrible software developers, and you'll get terrible increments of functionality every Sprint."

When I hear about teams that have tried Scrum and given up because it was "too difficult," I know that they are not talking about the complexity of Scrum. These are software developers. They are some of the smartest problem solvers you'll ever meet. Besides, Scrum is easy to understand. Chapter 1 pretty much covered it. No, what these people are talking about is the *discipline* of practicing Scrum correctly within an organization that allowed them to do so, every single day. That's why they gave up.

I agree with the Agile Manifesto. This is evident throughout this book as I point out the value of interacting and collaborating with individuals. I have discussed process and tools as well, but have been most vigilant in pointing out that not all application lifecycle management (ALM) tools and automation frameworks are healthy for a team. Most are. Some, however, can lead to one or more dysfunctional behaviors. For example, social networks, televisions with digital video recorders (DVRs), and video games are appealing and fun, but sometimes the kids (or developers in this case) need to get outside and interact with others.

Years ago, I was once asked to build a web-based work item approval system on top of Team Foundation Server (TFS). The client designed it so that email alerts would be sent when a work item changed to a certain state. These emails contained embedded hyperlinks that would redirect the user to a webpage that allowed managers or leads to authorize the state change. It was a sophisticated system—it even knew which users could cover for others if someone was on vacation or out of the office. My company built it. The client installed it. It did exactly what they wanted, but they ended up not using it. The reason they mothballed it was that it was too mechanical and removed the opportunity for two people to meet face to face and have a discussion. This was a learning opportunity for me and something I keep in mind whenever I see a shiny new feature in Microsoft Visual Studio. I ask myself, "Does this feature encourage collaboration or discourage it?"

When it comes time to meet and collaborate with members of your Scrum Team or stakeholders, here are some tips to consider:

- Establish the scope and the goal of the meeting, and stay focused on these topics.
- Meet face to face, especially if you anticipate a substantive conversation.
- Meet at a whiteboard, especially if you're intent on solving a problem.
- Set a time-box for the meeting. Be prepared to explain the concept.
- Leave the gadgets in the other room, unless they are required.
- Employ active listening techniques.

In this section, I discuss some of the general—but important—collaboration practices that a Scrum Team can adopt.

Listen actively

Software developers tend to have a short attention span and be impatient with anybody who is not as smart as them or who doesn't have the answer that they are looking for. Of course, I could just be talking about myself. But as they say, acknowledging that you have a problem is the first step in curing it. For me, *active listening* was that cure.

Active listening is a communication technique where the listener is required to feed back what is heard to the speaker. This can be as simple as nodding the head, writing a note on a piece of paper, or restating or paraphrasing what was said. This demonstrates your sincerity and respect for what the person is saying. It also helps alleviate assumptions and other things that get taken for granted. Opening a laptop and clicking through emails or otherwise getting distracted by anything else is not active listening and may even be considered disrespectful. Even "lightweight" devices such as tablets, slates, and smartphones can fall into this category.

Another part of active listening is waiting to speak. This is my particular problem. I tend to complete other people's sentences in order to move the conversation along to a more interesting topic. In my mind, I think I'm being helpful, but I know that I'm probably coming across as being rude. This is especially true for people who don't know me and is especially apparent to me when I have a conversation with another ADHD individual. Fortunately, there are techniques that can be used to overcome this particular interpersonal dysfunction. My favorite is to take a stack of sticky notes with me and write down the things that come to mind while the other person is talking. Soon it will be my turn to talk, and I can go back through my notes. See what I did? I solved the feedback and interruption problems with a single solution.

I'll re-mention HARD at this point. HARD is a mnemonic for Honest, Appropriate, Respectful, and Direct. It is a reminder of how you should always communicate with people, especially those that don't know you. Actively listening plus HARD communication is a recipe for successful collaboration.

Tailspin Toys case study During a recent Sprint Retrospective meeting, Scott (the Scrum Master) brought up his observations made during the Sprint. He witnessed a few developers having difficulty conversing respectfully with each other (as well as with stakeholders) during a couple of meetings. As a team, they decided to improve their communication abilities, specifically their active listening skills. Scott did some searching online and found several websites dedicated to the subject. During the next few Sprints, Scott coached the team as they adopted more and more of the techniques that they learned.

Collocate

I think we can all agree that communication and collaboration provides more value when practiced face to face, rather than remotely. At least I would hope that everyone knows this, because we experience it every day of our lives. When two people communicate face to face, they exchange more than just words. There are facial expressions, body language, and other nonverbal gestures. This kind of sideband data can be just as important, if not more important, than the text that is exchanged. For example, the look on a Product Owner's face when you suggest a solution to a problem can short-circuit the need for a detailed explanation. Thank you, collocated Product Owner. You just gave me back 20 minutes of my day.

Remember that Scrum has several formal events (meetings) built into the framework where collaboration can occur. In addition, the Scrum Team, and certainly the Development Team, should be continuously "meeting." These are not traditional meetings, where someone speaks and everyone else listens. These are short, collaborative, time-boxed meetings with the specific purpose of solving a problem. In fact, I wouldn't even call them a meeting, but more of a conversation. It's important that they occur as needed, with no logistical impediments. For example, if two developers need to discuss something with the Product Owner, but all the conference rooms are booked, they should meet anyway, somewhere, anywhere. To some degree, business formalities, and even etiquette, go out the window during the Sprint when the Development Team is in the zone, developing and generating business value.

When forming a new Development Team, collocation should be a requirement. This is not just a nice-to-have feature. It's required if you want a high-quality product and process. By collocation, I'm not talking about being in the same time zone, city, or building. While these options are better than some I've seen, I want the team in the same room or in adjacent rooms. The Product Owner should be nearby too, but not necessarily in the same room. This way, the face-to-face communication can occur on demand.

Tip Fellow Professional Scrum Developer Simon Reindl suggests bringing a geographically dispersed team together periodically. This is especially true at the beginning of a new project, so they know with whom they are working.

Professional Scrum developers know the value of collocation, and they strive for it. That said, there may be cultural, political, or financial reasons for not collocating the Development Team. This is the reality that I see as I visit larger organizations. The most common justification I'm given when I ask why the team is not collocated is that it saves money to have one or more of the functions supported or outsourced remotely, usually overseas. When I hear that, I hope that somebody, somewhere is doing the math on that, taking into account the decreased quality of the product and the process. Even if this decrease is not detectable or measurable, the decision makers should consider what the increase in quality *could be* if they were to bring the entire team together.

Note Do I think that developers working remotely as part of a distributed team *can't* be professional? Of course not. They absolutely can be professional and the team absolutely can collaborate, deliver high-quality software, and create business value. That said, an attribute of a professional Scrum developer is to inspect and adapt constantly, such as looking for ways to improve the process. Collocating a dislocated team is one of the biggest improvements that can be made, usually resulting in an increase of quality and Velocity. That team's Product Owner should wake up in the morning and go to bed at night, thinking of ways to maximize the product's value through the work of the Development Team, such as through collocation.

Most organizations consider their custom software as a strategic advantage over their competitors. I will sometimes ask executives where they would be without their line-of-business (LOB) application or public-facing website. They all agree that it would be a complete disaster. Not only has their staff forgotten how to run the business manually using paper and pencil, but they don't even know where to find the paper and pencils. Next, I ask them why they try to save money by limiting the capabilities and productivity of the team developing that custom software. At this point, I'm either asked to tell them more, or I'm escorted out of the building.

Note I recently had a conversation with an IT director of a very large organization. He explained to me that the Product Owner worked out of the main office, as did the programmers. The testers were overseas—nearly 10 time zones away. He shared with me a problem that they'd been having for the past few months. He said the programmers would code a feature and then go home for the night. The testers would come in, download the binaries, begin testing, and run into a bug. This blocked them from doing any further testing until the developers could fix it. The programmers would come in the next day, see the lack of progress, fix the bug, and have to wait until the end of the day for the testers to do their thing. Sometimes this dance would take three to four days before testing could proceed. He asked me how TFS could help him. I answered by asking why the testers weren't collocated with the rest of the team. He told me it was because they save money by sending the work offshore. I'm glad we were having this conversation in person because he was able to see the awesome facial expression I made at that point.

Set up a team room

Having the entire Development Team work in a shared, common room can be a good thing. Whiteboards containing plans and design notes are visible to everyone. Artifacts such as the Sprint Backlog and burndown chart can be updated easily and seen by everyone. During critical design points, the team room can become a war room of sorts as the developers move from strategic planning to tactical planning. Communication becomes more open and happens in real time. Developers tend to focus their productivity toward solving problems, while minimizing time spent on wasteful activities. Team rooms allow everyone, including stakeholders, to feel that buzz that I mentioned in the beginning of the chapter.

However, not every developer wants to work in a war room every day. There needs to be the opportunity to have private conversations, take phone calls, or just take a timeout from the rest of the team. Developers are smart and can self-organize to come up with solutions for these requirements. I've seen developers put on headphones, adjourn themselves to quiet rooms, or work away from the office for a short time as needed. Ideally, the managers and the organization trust their developers to the point where they can accommodate their needs. If they don't, then that is a big impediment to self-organization. Generating business value in the form of working software is a way for the Development Team to earn that trust.

Some personalities and cultures see collocation as an impediment. These developers may actually be counterproductive in such an environment. Remember that Scrum is about people, and people are just human. Their idiosyncrasies map directly to their ability to collaborate and work effectively as a team. The Velocity at which the Development Team is able to create business value is a function of the Development Team's productivity. Perhaps for these people, being in close proximity to, but not in the same shared room with, the rest of the team is good enough at first. A strong Scrum Master, as well as open and honest Sprint Retrospectives, can be used to improve this.

Note An open-space team room is not the same thing as an *open-plan* office. Open-plan offices are typically inhabited by employees working on different tasks for different projects. Open-space team rooms are inhabited by developers working on a common software product. Both environments can generate noise, but the type of conversations found in an open-plan office will typically be more contrasting and thus, more distracting.

My recommendation is to set up a team room and just try it out. See if management will let the Development Team take over one of the conference rooms for a Sprint or two. If, during the Sprint Retrospective, the Development Team honestly believes that they were productive, then the Scrum Master can work with management to create a more permanent, open-space room.

Tailspin Toys case study The Development Team has been collocated since day one, with Paula (the Product Owner) in a nearby office. During the Sprint, they regularly meet and collaborate whenever and wherever it is required. Day to day, the developers sit near each other in a large, open-space room with a half-dozen whiteboards (approximately one for each PBI). Because the developers use laptops with wireless connections, there's a minimum amount of cables in the room, and individuals can be more nomadic as they work. When one of the developers needs to concentrate or requires some personal space, he or she will put on headphones or go to a quieter room down the hall. When a developer has to travel or otherwise work remotely, the team will set up a dedicated computer with an always-on Skype connection, including video. Scott (the Scrum Master) has done a good job of educating the organization. Although the stakeholders know where the team room is located, they know to avoid it during a Sprint—unless of course they're invited by the Development Team. Scott still has to remind them from time to time.

Meet effectively

High-performance Scrum Development Teams know to avoid meetings, if possible. To be clear, I'm not talking about the built-in Scrum events, such as the Sprint Planning meeting, the Daily Scrum meeting, the Sprint Review meeting, or the Sprint Retrospective meeting. I'm also not talking about the regular Product Backlog grooming sessions, nor those impromptu but important meetings requested by the Development Team in order to clarify requirements, gather feedback, or seek the Product Owner's acceptance. I hope, in fact, that I've made it clear that these meetings are important and they should be attended by all of the involved parties face to face, if possible. I am talking about all the other meetings that an organization might require its technical staff to attend. You know the ones that I'm talking about They are mandatory, read-only (they don't ask for your feedback), and provide zero business value to the software product being developed or the development process itself. Unfortunately, some of these meetings cannot be avoided. They are a fact of life and a requirement to keep your job and get paid.

When you are invited to such a meeting, try to identify its purpose and expected outcome. This may be stated in the invitation, but if it's not, you may have to query the meeting organizer or sponsor. I know many developers who will not accept a meeting invitation if no clear agenda or objective is given. From this information, hopefully you can determine who the intended audience should be. Will the meeting be technical? Will decisions be made? If you don't fit the audience profile, try to skip the meeting, or send the Scrum Master instead. Being a proxy for the Development Team at meetings like this is one of his or her duties and allows the Development Team to what they do best.

If the tables are ever turned, and you find yourself organizing a meeting, you can follow the same advice:

- Only schedule meetings that are absolutely necessary and that can't be satisfied by one of the other built-in meetings.
- Keep the meeting as short as possible.
- Establish a time-box to enforce it.
- Outline the agenda and expected outcome in the invitation.

- Send invitations only to those people who need to attend.
- At the beginning of the meeting, explain the time-box and its concept.

When someone who is versed in Scrum sets up and runs a meeting, he or she will end up sharing good behaviors and practices, such as transparency, active listening, and time-boxing. This is a good way to get others in the organization more educated on Scrum and some of its attributes and practices. If appropriate, email any retrospective notes to the attendees, including action items. These behaviors may even infect the organization as other business units and teams will want "to get some of that Scrum."

Tip One way to keep meetings constructive is to say "yes, and" instead of "yes, but." If the current topic or solution being discussed is one that there is partial agreement on, saying "yes, and …" comes across as being more constructive. If someone hears "yes, but," then they might think their idea is being discounted, or they may feel limited in what can be accomplished. If, however, they hear "yes, and," they will think that their idea was accepted, or at least understood, and be more prone to ideas. More importantly, the person will be more open to collaborating on a shared solution, which should always be the goal to avoid discussions becoming polarized.

Tailspin Toys case study Paula (the Product Owner) and Scott (the Scrum Master) are good at running interference for the development team. For meetings that are not related to the development of the software product, Scott will try to attend as a proxy for the Development Team. Some meetings, such as the "all hands" meetings, cannot be avoided, and the developers do attend them.

Collaborate productively

Collaboration means working with people. This typically means dividing the work between two or more individuals and working together. Both the process of dividing the work and the actual working together with others can require intense concentration. Getting into this productive state, otherwise known as the *flow* or the *zone,* can take time. Getting out of that state prematurely, as caused by any kind of interruption, can be considered waste. The irony is that collaboration *requires* interruption, and you will need to get used to it and master it.

We are taught at a young age that it is disrespectful to interrupt others. If your team is working in an open-space team room, it's easy to see when a fellow developer is deep in thought or in the zone. Your instinct should be not to interrupt them. When you're working by yourself, however, it may be harder to know when *you* are in the zone. Stopping to take a mental assessment may actually kick you out of the zone. High-performance Scrum developers know how to minimize interruptions in order to maximize productivity. There have been numerous books, blog posts, and white papers written about being more productive.

Here are some of my favorite tips:

- **Cell phone** Turn it to vibrate, turn it off, or leave it at home.
- **Exit Microsoft Outlook** Email can be a great productivity tool, but it can waste a lot of your time as well. If you can't or don't want to turn it off, then be sure to disable all notifications. Having an icon appear in the system tray, seeing the mouse pointer change, or hearing an audible alert when a new email arrives, can have the same conditioning effect as one of Pavlov's dogs hearing a bell ring. Try to check email only three times a day: at the start of your day, after lunch, and before leaving.
- **Exit IM/chat client** Close the program, or at least set your status to busy. The exception to this is if the tool is used by the Development Team to share code or quick questions and feedback.
- **Limit Internet searches** Developers can spend their whole day on the Internet if they are not careful. Time-box the search and keep the scope to just researching the problem at hand.
- **Just get started** Some planning is required before starting a task, but overplanning becomes the antithesis of productivity.
- **Avoid formal meetings** One reason that Scrum is so successful is that it defines the important meetings to minimize the need for unimportant ones. A developer's productivity drops when he or she is away from the keyboard. Feel free to attend the valuable ad hoc meetings over coffee or at another's desk, but send the Scrum Master to the formal meetings in the Development Team's stead.
- **Use active listening** When your colleague is talking, you should listen to what he or she is saying, and expect the same courtesy when you are talking.
- **Stop fiddling** Developers can have complex software environments. These can include multiple versions of software, one or more integrated development environments (IDEs), virtual desktops and servers, databases, frameworks , software development kits (SDKS), testing tools, installers, etc. Do yourself a favor. Get it working, script it, snapshot it, and forget about it. Endless tweaking tends to have a diminished return on value. Solve today's problem today and tomorrow's problem tomorrow.
- **Life happens** We're all human and have a life outside of software development. When issues emerge, be open and honest about it, and take the necessary time to get your head right. Be appropriately transparent with the rest of your team.

Tailspin Toys case study The Scrum Team is always looking to do better. This is evident during their Sprint Retrospective meetings where collaboration practices are almost always discussed as improvement is sought. Everyone knows that the best way to increase Velocity is to improve the individuals and interactions.

Achieve continuous feedback

Developers love feedback loops—the faster the better. As soon as we type a few lines of substantive code, we hit F5 to see what the compiler thinks. As soon as we've got the method refactored, we run our unit tests to see them pass. As soon as we have a tangible user interface (UI), we have a colleague or the Product Owner look at it to tell us how he or she likes it. As soon as we are done with a task, we check in so that the continuous integration build or another developer can evaluate our work. Continuous feedback like this is healthy for the product, as well as the developer.

Automated feedback provided by builds, unit tests, code coverage, code analysis, and acceptance tests are awesome. Developers can call upon Visual Studio or TFS to provide this feedback at any time, day or night. The results tell the Development Team that they are building the feature correctly. High-performance Scrum Development Teams will take advantage of all of these features to ensure that they are well informed about the progress and quality of their work.

Smell It's a smell if the Development Team doesn't ask for feedback from the Product Owner during the Sprint. Passing unit and acceptance tests only ensure that the quality of the feature or scenario has been met. The Development Team will want to make sure that the person requesting the feature (the Product Owner) is happy with its design, function, and usability. The Sprint Review meeting should not be the first time that the Product Owner sees a feature being demonstrated.

Product Owner feedback is just as important as other types of feedback. An engaged Product Owner who knows the product and the desires of its users can quickly give the Development Team positive or negative feedback on a feature being developed. Getting in-person guidance on the usability of a feature early in its development is very valuable. If the Development Team builds the wrong feature, it's essentially the same as if they introduced a bug into the software product. The same advice goes for features as for bugs—it's easier and cheaper to "fix" them earlier in their lifecycle.

Note The Product Owner feedback loop should be as short (fast) as possible as well. This is another argument for collocating him or her near the Development Team.

I'm often asked if the Development Team can reach out directly to the stakeholder (user or customer) who requested the feature in order to gather feedback. Technically, the answer is no. The Product Owner is the one source of feedback to the Development Team. If she wants to establish her own feedback loop to the stakeholders, that's her prerogative. That said, I feel that there are times and conditions where the Development Team can solicit feedback directly from a stakeholder if they decide that bypassing the Product Owner will provide them more value. The Product Owner should be informed and agree to this. During the next Sprint Retrospective meeting, this can be discussed to determine if it was a one-time thing or if there's a deeper dysfunction to address (like an untrained or absent Product Owner).

I see Product Owner feedback as falling into three broad categories in Scrum, with practices and tools that can support each. These are listed in Table 8-1.

TABLE 8-1 Types of Product Owner feedback with the associated practice and tools.

Type of feedback	When is it given?	Practice	Visual Studio tool
Can you give us more details about this PBI?	Product Backlog grooming, Sprint Planning meeting during development	Collaborate with the Product Owner or stakeholder at a whiteboard	PBI work item, PowerPoint storyboarding
Do you like this? Is this the behavior you were expecting?	During development	Sit down with the Product Owner or stakeholder and go through the feature	Microsoft Feedback client
What else do you want, not want, or want developed differently?	Sprint Review meeting	Collaborate with the Product Owner and stakeholders to update the Product Backlog	Team Web Access, Microsoft Excel

The rest of this chapter will discuss some of the more effective collaboration practices and tools.

Collaborative development practices

Even the simplest software product requires a team with many talents. Beyond having the standard capabilities of design, code, and test, there can be many types and levels of talent within each discipline. Every developer has a unique background, set of skills, expertise, and personality. Each brings something different to the team. For example, you may have two C# programmers with similar resumes and experience. The way in which they analyze and solve problems will vary radically. Both approaches can be fit for purpose according to the requirements, but they can be very different.

A high-performance Scrum Development Team understands this reality, and even uses it. These types of teams recognize everyone has a different way of solving problems, and so long as those solutions fit within the parameters of the product and the Development Team's practices, they should be embraced. Long, drawn-out discussions and arguments over approaches and coding styles tend to generate little value, and typically only lower Velocity and morale.

In this section, we will explore several contemporary practices that boost the Development Team's effectiveness during collaboration.

Note A self-organizing Scrum Development Team should pick and choose from these as well as other development practices and try them for a Sprint or two. Later, during a Sprint Retrospective, the team can decide whether to continue to embrace the practice, to amplify it, or to abandon it.

Collective code ownership

Extreme Programming (XP) gave us the notion of *collective code ownership*. With this approach to ownership, individual developers do not own modules, files, classes, or methods. All of those things are owned collectively, by the entire Development Team. Any developer can make changes anywhere in the code base.

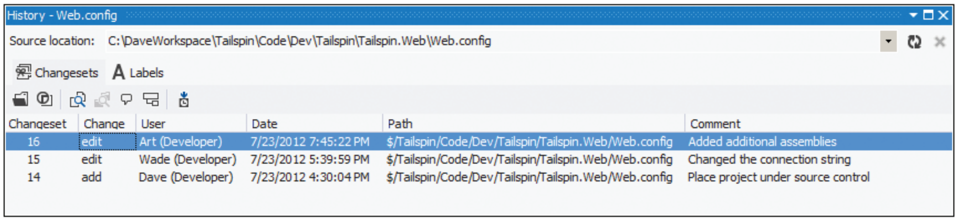
Consider the alternative to collective code ownership, where each developer owns an assembly, a namespace, or a class. On the surface, that may seem like a good idea. The developer is the expert on this component , as well as the gatekeeper for all changes. Strong code ownership like this has a tendency to block productivity. Consider the situation where two developers (Art and Dave) are working on separate tasks that both need to touch a common component owned by a third developer (Toni). Dave will have to wait while Art's functionality is coded and tested. A collective code ownership model would allow Dave to code the feature himself. The source control tools in TFS would track who made what changes to which files and enable a merge (or a rollback) to occur if there were any problems. Another potential problem with strong code ownership pops up when refactoring. Modern refactoring tools, like those in Visual Studio, can do this safely, but if the file or files are locked, then productivity is blocked again.

Adopting a collective code ownership mentality can take time. This is especially true if the Development Team used to have strong code ownership. Pairing and shared learning is a way to break up the turf and politics. Just as it takes time for the Product Owner and organization to trust the Development Team's ability to self-organize and self-manage, it also takes time for the individual developers to trust each other.

Tracking ownership in TFS

The biggest advantage with collective code ownership is the boost in the social dynamics of the Development Team. Because each developer has full control over all source code, there are less boundaries and more opportunities to find solutions. Remember that in Scrum, the Development Team owns all the problems and all the solutions collectively. This includes the artifacts of those solutions, namely the source code.

Should you ever have a need to determine who made a specific change to a file, TFS can help you. By right-clicking a file or a folder and selecting View History (as shown in Figure 8-1), you can see a history of changes, including who made them, the type of change, the date and time, and a (hopefully meaningful) comment. If you want to see what was changed between two versions, you can select them both and right-click, choosing Compare as shown in Figure 8-2. The UI will show removed text in red and new text in green. If you want to see who wrote which line of code in a specific version of a file, you can use the Annotate tool as shown in Figure 8-3.



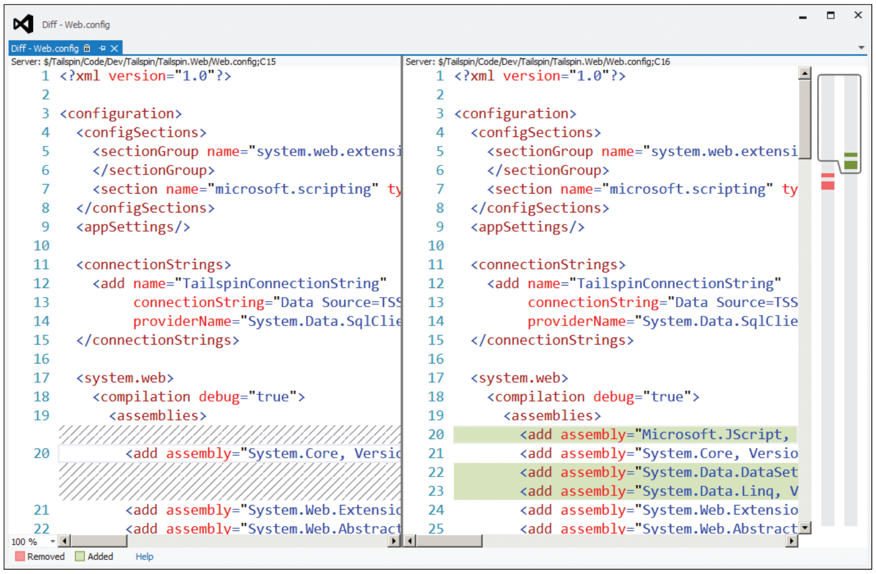
History - Web.config

Source location: C:\DaveWorkspace\Tailspin\Code\Dev\Tailspin\Tailspin.Web\Web.config

Changesets | Labels

Changeset	Change	User	Date	Path	Comment
16	edit	Art (Developer)	7/23/2012 7:45:22 PM	$/Tailspin/Code/Dev/Tailspin/Tailspin.Web/Web.config	Added additional assemblies
15	edit	Wade (Developer)	7/23/2012 5:39:59 PM	$/Tailspin/Code/Dev/Tailspin/Tailspin.Web/Web.config	Changed the connection string
14	add	Dave (Developer)	7/23/2012 4:30:04 PM	$/Tailspin/Code/Dev/Tailspin/Tailspin.Web/Web.config	Place project under source control

FIGURE 8-1 Viewing a history of changes to a specific file in Team Foundation Server.



Diff - Web.config

Server: $/Tailspin/Code/Dev/Tailspin/Tailspin.Web/Web.config;C15

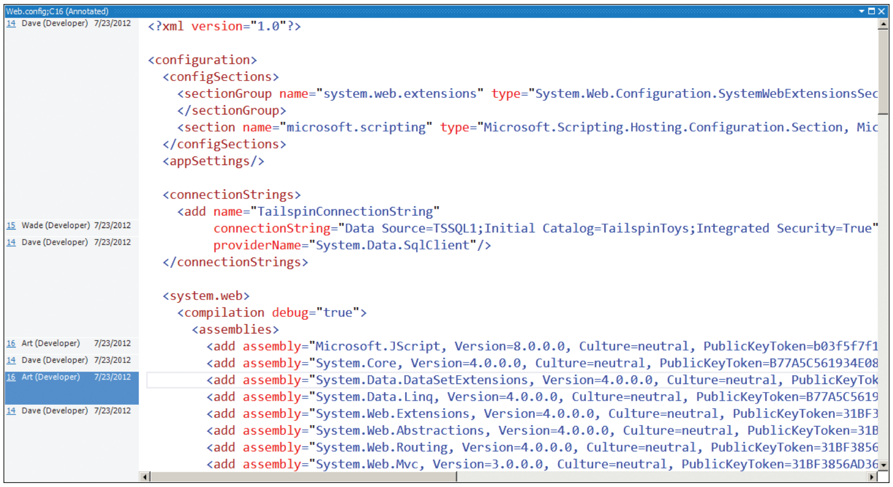
```
<?xml version="1.0"?>

<configuration>
  <configSections>
    <sectionGroup name="system.web.extensi
    </sectionGroup>
    <section name="microsoft.scripting" ty
  </configSections>
  <appSettings/>

  <connectionStrings>
    <add name="TailspinConnectionString"
         connectionString="Data Source=TSS
         providerName="System.Data.SqlClie
  </connectionStrings>

  <system.web>
    <compilation debug="true">
      <assemblies>

        <add assembly="System.Core, Versic


        <add assembly="System.Web.Extensic
        <add assembly="System.Web.Abstract
```

Server: $/Tailspin/Code/Dev/Tailspin/Tailspin.Web/Web.config;C16

```
<?xml version="1.0"?>

<configuration>
  <configSections>
    <sectionGroup name="system.web.extensi
    </sectionGroup>
    <section name="microsoft.scripting" ty
  </configSections>
  <appSettings/>

  <connectionStrings>
    <add name="TailspinConnectionString"
         connectionString="Data Source=TSS
         providerName="System.Data.SqlClie
  </connectionStrings>

  <system.web>
    <compilation debug="true">
      <assemblies>
        <add assembly="Microsoft.JScript,
        <add assembly="System.Core, Versio
        <add assembly="System.Data.DataSet
        <add assembly="System.Data.Linq, V
        <add assembly="System.Web.Extensio
        <add assembly="System.Web.Abstract
```

100 % 🟥 Removed 🟩 Added Help

FIGURE 8-2 Comparing two versions of a file to see the differences.



Web.config;C16 (Annotated)

```
14 Dave (Developer) 7/23/2012   <?xml version="1.0"?>

                                <configuration>
                                  <configSections>
                                    <sectionGroup name="system.web.extensions" type="System.Web.Configuration.SystemWebExtensionsSec
                                    </sectionGroup>
                                    <section name="microsoft.scripting" type="Microsoft.Scripting.Hosting.Configuration.Section, Mic
                                  </configSections>
                                  <appSettings/>

                                  <connectionStrings>
                                    <add name="TailspinConnectionString"
15 Wade (Developer) 7/23/2012            connectionString="Data Source=TSSQL1;Initial Catalog=TailspinToys;Integrated Security=True"
14 Dave (Developer) 7/23/2012            providerName="System.Data.SqlClient"/>
                                  </connectionStrings>

                                  <system.web>
                                    <compilation debug="true">
                                      <assemblies>
16 Art (Developer)  7/23/2012           <add assembly="Microsoft.JScript, Version=8.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f1
14 Dave (Developer) 7/23/2012           <add assembly="System.Core, Version=4.0.0.0, Culture=neutral, PublicKeyToken=B77A5C561934E08
16 Art (Developer)  7/23/2012           <add assembly="System.Data.DataSetExtensions, Version=4.0.0.0, Culture=neutral, PublicKeyTok
                                        <add assembly="System.Data.Linq, Version=4.0.0.0, Culture=neutral, PublicKeyToken=B77A5C5619
14 Dave (Developer) 7/23/2012           <add assembly="System.Web.Extensions, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31BF3
                                        <add assembly="System.Web.Abstractions, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31B
                                        <add assembly="System.Web.Routing, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31BF3856
                                        <add assembly="System.Web.Mvc, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD36
```

FIGURE 8-3 Using Annotate to see which developer made which changes in a specific file.

Tailspin Toys case study Because each member of the Scrum Team is a team project administrator, everyone has full control over every aspect of the team project. This includes the ability to view, edit, and even delete files from source control. Should the need arise to see who made a change, the developers are all trained in TFS and can view history, compare, and annotate as needed. Sometimes they will use the Annotate feature to praise another developer for good work.

Commenting in code

With collective code ownership comes a certain amount of responsibility. Other developers on the team will need to understand the code. If a developer or pair of developers is working on a rather complex part of the code, they should consider adding some comments. This can be a block of comments that give another developer enough information to understand this code. The comments can also be regularly sprinkled throughout longer algorithms. You can think of comments as being messages to the future, and it might be *you* reading those comments a year from now.

Tip Comments shouldn't tell the reader how the code works. The code should tell them that. If the code isn't clear, then you should refactor the code rather than add descriptive comments.

When commenting in code, only comment about what the code can't say for itself. If the code is well formed and follows popular patterns and principles, it probably doesn't need comments. When someone looks at the source code, its logic and purpose should be apparent. Keep this in mind while you are coding. Constantly ask yourself how clearly your code is telling you, or another developer, what it is doing.

Tip Fellow Professional Scrum Developer Jose Luis Soria Teruel suggests that commenting in perfectly readable code can sometimes be useful too. For example, in Microsoft Visual C#, you can create documentation for your code by including XML tags in special comment fields in the source code directly before the code block they refer to. If you are developing an application programming interface (API) for third parties, you may want to at least use the *summary* tag to describe a type or a type member.

Remember that comments live inside your source code files, and as such, they become inventory just like the code itself. Comments can even be a form of technical debt if they are wrong or misleading. Be diligent about updating your comments or removing them as you refactor and improve your code. Adding more comments isn't necessarily a good thing unless they add value. Perhaps it's time to refactor the code into simpler units rather than adding more comments. You should prefer unit tests over comments. The best comment is a set of working unit tests with high coverage.

Smell It's a smell when I see a file with the author's name at the top. I understand a developer wanting to get credit for his or her work, but this kind of comment tells everyone else to go away. It could be that the code file is really old and hasn't been touched since the team started practicing collective code ownership. If that's the case, someone should remove it. TFS tracks this through changesets, so it is redundant anyway. It could also be an organizational requirement to have predefined headers and require authors to add their names. If that's the case, meet with the decision makers and ensure that the value delivered by the practice outweighs the waste that it seems to generate.

Tailspin Toys case study The Development Team uses popular frameworks, principles, and practices as they design and code. As a result, there's not a lot of opportunity for meaningful comments. Only when they are coding some complex LOB methods do they add comments. The Development Team also knows that when checking in to TFS, they will associate a Task work item (which links back to a PBI or Bug work item) and a meaningful comment. Together, these two items provide more than enough context to explain later *why* the changes were made. Additional comments in code are not required.

Code reviews

A code review is a simple way to assure code quality by having another developer look at the code. This assurance can cover multiple levels of quality. It can assure that the code works, is fit for purpose, is absent of bugs, is absent of avoidable technical debt, is readable, and meet's the team's agreed-upon coding standards, as well as the Definition of "Done." Additionally, the developer whose code is being reviewed can use the conversation as an opportunity to learn about the way that he or she writes code.

Professional Scrum developers recognize that the candid feedback (otherwise known as criticism) given during a code review is targeted at the code and not themselves. For new developers, or developers new to code reviews, there can be a tendency to take these criticisms as an insult, even becoming defensive. Over time, these developers will see that even experienced developers make mistakes. Everyone is human. Everyone screws up now and then. Everyone can improve. Code reviews are just another type of shared learning activity, where any developer can learn from another.

Tip Code reviews can also catch and enforce coding style and standard issues. Be careful spending too much time with these kinds of topics during a code review, as they can become a rathole. A *rathole* is any discussion that detours the original purpose of the conversation. Don't get me wrong—discussions around coding styles and standards are very important, but any debate or decisions around changing existing standards, or establishing new ones, should be deferred until the Sprint Retrospective meeting. High-performance

Scrum Development Teams know that matters of style are not absolute. Developers should be allowed to self-organize and use whatever style is fit for purpose. Once a Development Team has been working together for a while, their coding standards will begin to emerge. These standards may even become part of the Definition of "Done."

When reviewing someone else's code, you should avoid appearing as a "senior" developer. The truth is that you may be the senior developer, but because everyone is equal within a Scrum Development Team, it's all about the sharing and learning. Choose your tone and your words carefully as you identify problems and improvements in someone else's code. Developers new to Scrum may be put on the defensive. Don't aggravate the situation by also going on the offensive.

Code reviews don't have to be a formal process. They can happen spontaneously. They also shouldn't be despised or avoided. High-performance Scrum Development Teams actually look forward to code reviews. This is because those teams know that the code is owned collectively. Problems and criticisms aren't directed at a single developer; rather, they are learning opportunities for the entire team. Every code writer and code reviewer will have different perspectives and approaches to solving problems.

Tip Typically, most developers know the code that needs to be reviewed. This can change, depending on the frequency of the code reviews. Developers can forget the changes that they made and the context if too much time elapses. Fortunately, TFS knows what files a developer has worked on for any given date range. From inside Source Control Explorer, the developer can right-click a parent-level folder and view the history. Unfortunately, this will show activity from every developer. There's no way to filter out other developers' changes. If you drop the command line, however, this filtering can be accomplished using the Tf.exe command-line utility.

Here's an example where Dave is asking TFS to list all of his changesets for a given date range:

```
tf.exe history $/Tailspin/Code/Dev /version:D"07/04/2012"~D"07/17/2012" /user:"Dave
(Developer)" /recursive
```

Professional Scrum developers should build solutions that are fit for purpose while avoiding gold plating. *Gold plating* is any design or coding that is above and beyond what is absolutely necessary for the task at hand. For example, if a PBI requires a method that calculates the sales tax for the state of Washington, and the developer adds additional logic to handle the nearby states, that's gold plating. The developer may try to justify the extra coding as being required down the road for a future Sprint. In order to maximize value and minimize waste, Development Teams should solve today's problem today and tomorrow's problem tomorrow (in the next Sprint, as it is in Scrum). Code reviews can be a good way to unearth gold plating.

Pair programming

You can think of pair programming as a form of code review—one that happens in real time. The practice of pair programming has two developers sit together at one computer. One developer types at the keyboard (drives), while the other observes, navigates, spellchecks, and otherwise reviews the code being typed. The two developers will switch roles frequently.

A benefit of this two-person approach is that the driver can focus on the tactical (coding) activities, while the observer is thinking about the broader, strategic solution to the problem. This collaboration leads to better and simpler designs and fewer bugs, in shorter periods of time. Pairs of developers working in close proximity like this are also less prone to get sidetracked from the task at hand.

During pair programming, knowledge is passed back and forth. The two developers can learn new practices and techniques from each other. Pairing a newly hired developer, or a developer with a different or weaker skill set, with a developer who is stronger will help improve the overall effectiveness and Velocity of the Development Team. Some teams scale this idea using an approach called "promiscuous pairing." Each developer cycles through all the other developers on the team, rather than pairing with only one partner. This behavior causes knowledge of the software product and its inner workings to spread throughout the whole Development Team. This reduces risk if a key developer leaves the team. Figure 8-4 demonstrates the possible outcomes of pairing weaker and strong developers together.

		Developer B	
		Weak	**Strong**
Developer A	**Strong**	Mentoring	Flow
	Weak	Danger	Mentoring

FIGURE 8-4 Possible outcomes when pairing developers together.

Tailspin Toys case study The organization has no policies around code reviews. They leave it up to the Development Team to decide. Sometimes the developers perform ad hoc code reviews. These are done whenever a developer hits the wall or needs a better solution for a complex problem. These kinds of code reviews are almost like an impromptu pair programming session. In addition, the entire Development Team likes to sit down in the conference room and use a projector. Each developer in turn shows off code. With a full room, this approach encourages discussion on design and style. As the team has improved, each review begins by showing the automated tests.

Collaborative development tools

Over the years, I have met with hundreds of software development teams. In my opinion, the most productive, collaborative tools for software developers to facilitate a discussion are a whiteboard and a dry erase marker or a laptop running Visual Studio and a projector. Using tools like these implies several things: the collaborators are collocated, they each see the same thing, and they are having a discussion in real time. There are no environmental impediments blocking the flow.

In a perfect world, all discussions and brainstorming meetings would occur like this. Unfortunately, some of us work in a world that is not collocated. Our team members don't work in the same office, or even live in the same city, state, or country. When we are in bed, our colleagues are at work, and vice versa. For environments like this, high-bandwidth collaboration tools like a whiteboard don't have the same impact. Alternatively, electronic tools must be substituted. Fortunately, Visual Studio 2012 includes several good ones.

Tip There are countless more collaborative development tools available as open source or for commercial license. A popular example is *join.me*. It is a free (and ridiculously simple) screen-sharing tool for meetings on the fly. You can learn more at *http://join.me*.

In this section, I will discuss some of the Visual Studio 2012 features that enable collaboration.

Team Foundation Server

Team Foundation Server is the team's hub for coordinating development efforts on a shared code base using shared work items and shared, automated builds. TFS directly supports the first two, and Team Foundation Build (Team Build), a feature of TFS, enables automated builds. The team can use Team Build to automate the compilation, deployment, and testing of its software products. Having at least one automated build for the product should be a goal. High-performance Scrum Development Teams will have several.

TFS should be at the center of the development team at all times, especially when coding. There are challenges, however, when supporting a busy team of developers working on a shared code base. Parallel development such as this can lead to concurrency issues. In the time between a developer getting the latest version of code, making changes, and then checking it back in, one or more developers may have checked in their changes to the same folder. This means that when the original developer checks in his or her code, the probability that a conflict will occur increases with the length of time that the code is not checked in. Because these conflicts usually require a merge operation, you should check in frequently.

Merging occurs when two variants of the same file are combined in a logical way to create a new version of the file. Manually integrating files like this is a time-consuming process and should be avoided. TFS can often auto-merge for you, but not always. One way to avoid having to merge is to enable locking so that each developer locks the file(s) that he or she is working on. While this will prevent anyone else for making changes to the file until the first developer is done, it will block other developers from working on the file and being productive. There is a better way—to integrate, or merge, continuously.

Smell It's a smell when I see a team project that does not have *multiple check-out* enabled. Either the team has been burned in the past by an inferior revision control system and wants to play it safe, or they haven't learned how to collaborate together effectively. Either way, wholesale locking like this is a recipe for an impediment. To overcome this, I usually start with a bit of education, letting the developers know that even with multiple check-out enabled, they are still able to lock individual files as needed, such as when performing a tricky refactoring operation. I've yet to see a team project that truly required locking of this nature that didn't have a deeper dysfunction driving the need.

Continuous integration

High-performance Scrum Development Teams have learned how to work smarter, not harder. One way that they do this is by continuously integrating their code changes with others on the team and running automated tests to verify the integration didn't break anything. While these same automated tests can be run inside Visual Studio, Team Build can probably run them faster and they will be asynchronous, enabling the developer to work on something else. Another benefit is that the tests can be run in a controlled environment that will show any configuration management problems quickly.

A better way to avoid painful, manual merge operations is to do smaller, less painful merges throughout the day. This is the basis for continuous integration (CI). Automated CI takes this a step further. Upon a check-in, an automated build gets launched, having been triggered by a check-in event. Source code is compiled, binaries are deployed, automated tests are run, and feedback is returned to the team *quickly*.

Tip Another way to minimize the pain of manually merging code is to *listen* to the other developers during the Daily Scrum. Remember that the purpose of the Daily Scrum is to synchronize and create a plan for the next 24 hours. This means that each developer verbally shares their planned tasks with the other developers. If a developer hears another mention a task that will be in the same file or files that he or she was planning on working on, they should consider pairing up and working on their overlapping tasks together. This should alleviate any need to merge the code manually, as well as increase knowledge and productivity in general.

CI is about reducing risk. When a developer defers integration until late in the day, the week, or the Sprint, the risk of failure (i.e., features not working, side effects, bugs) increases. By integrating code changes with others regularly throughout the day, the Development Team will identify these problems early and be able to fix them sooner because the offending code is fresh in everyone's mind. The practice of CI is a must for any high-performance Scrum Development Team.

A Development Team shouldn't be afraid to break the build and work together to fix it. Refactoring, restructuring classes and methods, and changing internal interfaces can be messy work. There may be times that you want to check in your not-yet-finished code so that another developer can begin working with a part of it. You may also want to see how many errors and warnings and failed tests occur when your changes are integrated with others. This is fine. You are in the middle of the Sprint. This is not production code. Just as a surgeon may need to make some cuts in order to fix a critical problem, so might you have to break some code in the development process. These cuts are temporary, and the CI build and failing tests will illuminate them until they are all healed.

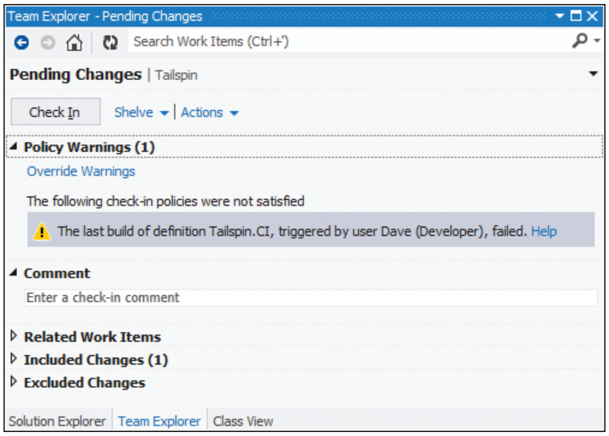
Tailspin Toys case study The Development Team has invested in a very powerful, very fast build server. They keep it quite busy, integrating code changes, building, and testing on every check-in. The developers aren't afraid to break the build, but they are disciplined to check the results of every build to ensure that they don't miss a broken one. Some have installed the build notification tool and others use email notifications to stay connected to the build status.

Builds check-in policy

Scrum Development Teams stay busy. They will work on a design task and when it's done, switch to a coding or testing task, and repeat, and repeat. As they finish a task, they usually check in their work. When CI builds are enabled, the check-in triggers an automated build. The developer then starts working on another task and, hopefully, remembers to go back and check that build's status and quality. Unfortunately, the developer may become focused on the new task or get sidetracked by something else. He or she may forget to evaluate the results of that CI build. Compound several builds on top of each other, and you might have a tangle of build results to work through.

The Builds check-in policy was created as a solution for just such a situation. When you configure a CI build in Team Build, every check-in operation starts a build. When one of these builds breaks, it is important for the Development Team to fix the problem that broke the build before making additional, unrelated changes. You can use the Builds check-in policy as a tool to limit additional check-ins until the broken build is fixed. When this policy is enabled, it literally blocks anyone else from adding new files to any source control folder that is a working folder in/under the build definition. When the policy fails, the developer who is attempting to check in will receive a message like the one shown in Figure 8-5.

When a developer runs into this warning message, the expected behavior is that he or she can query the other team member who "broke" the build. Remember, it could just be that a single test failed, and not some catastrophic system error. Once the Development Team has been consulted, the developer who received the warning can then choose to override it by clicking the Override Warnings hyperlink on the Pending Changes page and providing a comment. All developers will be blocked like this until the CI build completes without errors and all tests pass.


Team Explorer - Pending Changes

Search Work Items (Ctrl+')

Pending Changes | Tailspin

Check In | Shelve | Actions

Policy Warnings (1)

Override Warnings

The following check-in policies were not satisfied

The last build of definition Tailspin.CI, triggered by user Dave (Developer), failed. Help

Comment

Enter a check-in comment

Related Work Items

Included Changes (1)

Excluded Changes

Solution Explorer | Team Explorer | Class View

FIGURE 8-5 Builds check-in policy warning when the last CI build failed.

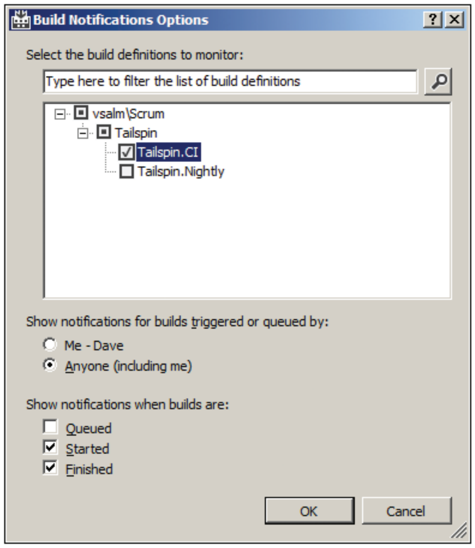
Tailspin Toys case study The Development Team tried the Builds check-in policy for a few Sprints and then, after discussing it during a Sprint Retrospective meeting, decided to disable it. What had happened was, as they improved their CI practice, they got better at proactively watching and analyzing the build results. In addition, some developers have opted to enable the Build Notifications tool.

Build Notifications tool

Rather than having TFS block check-ins when a build fails, some developers would rather just be notified when the build completes and then check the results manually. Fortunately, Microsoft includes a notification tool that does exactly this.

The Build Notifications tool is installed by default, but not configured. The developer will have to start it manually at first. It can be found under the Start menu by pointing to Microsoft Visual Studio 2012 > Team Foundation Server Tools. Each developer can choose the build(s) that they want to monitor. They can also choose to be notified when each build gets queued, starts, or completes. They can also choose to monitor only builds that *they* have started or that anyone on the team has started. You can see an example of the settings in Figure 8-6.

Note The Build Notification tool used to be part of the Team Foundation Server Power tools, but starting with TFS 2010, Microsoft now includes it "in the box," as part of a Visual Studio or Team Explorer installation.


Build Notifications Options

Select the build definitions to monitor:

Type here to filter the list of build definitions

- vsalm\Scrum
  - Tailspin
    - [✓] Tailspin.CI
    - [ ] Tailspin.Nightly

Show notifications for builds triggered or queued by:

- ○ Me - Dave
- ● Anyone (including me)

Show notifications when builds are:

- [ ] Queued
- [✓] Started
- [✓] Finished

FIGURE 8-6 Enabling build notifications for the Tailspin.CI build.

The notifications will appear in the system notification area (otherwise known as the *system tray*), in the lower-right corner of the Windows desktop. You can see an example of this in Figure 8-7. The notification will appear for a few seconds and then fade away. You can click hyperlinks on the notification to allow you to view the details of the build. If the notification has since disappeared, right-click the Build Notifications icon, in the system notification area, to view a build's status or replay a recent notification.

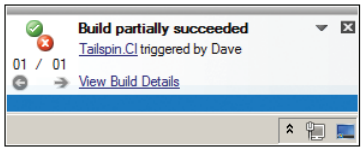

Build partially succeeded
Tailspin.CI triggered by Dave
01 / 01
View Build Details

FIGURE 8-7 A notification that a build has finished, but only partially succeeded.

Tailspin Toys case study There is not a team-level practice or requirement to use the Build Notification tool. Some developers on the team, however, have configured it and use it. Others have since disabled it. The most common reason for disabling it is to avoid the "noise" that it generates. As previously mentioned, the Development Team is quite good at watching the CI builds and responding to any problems.

Gated check-in builds

Gated check-in builds are a type of private build, triggered by a check-in, but built using shelvesets in order to ensure that there are no errors prior to checking in. The purpose of a gated check-in build is to verify that the developer's code integrates with the other team members and that tests pass before committing the changes to the main source control repository. This feature was introduced in TFS 2010.

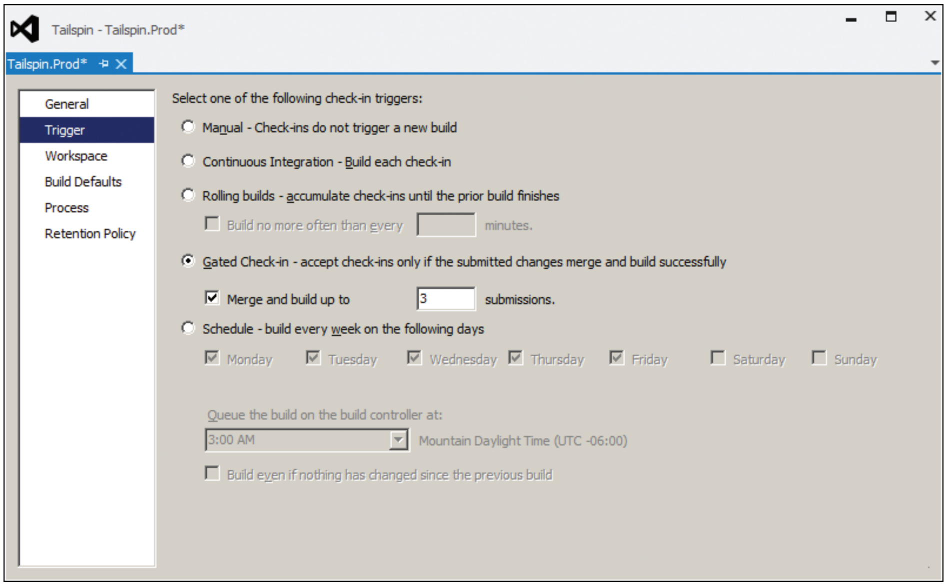
One of the problems plaguing the gated check-in build feature is performance. Even in TFS 2012, the slow performance of the UI notifications become apparent—very fast. Also, each gated check-in definition can have only one running build at a time. Therefore, active teams doing lots of check ins and builds are more likely to develop a large queue of gated check-in builds. Fortunately, there's a new feature in TFS 2012 that helps with performance, which I'll get to in a moment.

Smell It's a smell if I see a team using gated check-in builds on their development codeline. Ideally, the Development Team practices lots of small, frequent check-ins. If one breaks the build, a few minutes later, they should know about it, fix it, and keep going.

Gated check-ins are a solution to a misunderstanding. When the original authors of XP said, "Don't break the build," they didn't mean it literally. They meant that if a developer ever does break the build, it is their responsibility to fix it immediately. Really, the authors should have said, "Don't ever leave the build broken." When I've seen teams use gated check-in builds, they often do so because they are unable to meet the requirement of never leaving it broken. It could be that their build takes too long or they simply don't have the discipline to follow the practice. It also may be because the organization has a low tolerance for broken builds. We need to recognize that these are all dysfunctions of some type.

Tip If your production gated check-in build takes a long time to complete, even without running tests, consider creating a second CI build that builds and runs the tests. The CI build would kick off in parallel with the gated check-in build and provide a measure of quality. It might take a really long time to finish the CI build, but at least the gated check-in build wouldn't get any slower, and you'd still get a sense of the code quality.

For teams running long gated check-in builds, TFS 2012 offers a helpful new feature. Gated check-in builds can now batch together multiple shelvesets into a single build. For example, a team might configure a production build to build up to three submissions simultaneously, as shown in Figure 8-8. These submissions (shelvesets) would get merged and built together on one build agent. If the build succeeds, and all tests pass, each shelveset would be committed (checked in) separately. If the build fails or any tests fail, then each shelveset is built, one at a time, to determine which one caused the failure.


Tailspin - Tailspin.Prod*

Tailspin.Prod*

- General
- Trigger
- Workspace
- Build Defaults
- Process
- Retention Policy

Select one of the following check-in triggers:

( ) Manual - Check-ins do not trigger a new build

( ) Continuous Integration - Build each check-in

( ) Rolling builds - accumulate check-ins until the prior build finishes

☐ Build no more often than every [ ] minutes.

(●) Gated Check-in - accept check-ins only if the submitted changes merge and build successfully

☑ Merge and build up to [3] submissions.

( ) Schedule - build every week on the following days

☑ Monday ☑ Tuesday ☑ Wednesday ☑ Thursday ☑ Friday ☐ Saturday ☐ Sunday

Queue the build on the build controller at:

3:00 AM Mountain Daylight Time (UTC -06:00)

☐ Build even if nothing has changed since the previous build

FIGURE 8-8 Configuring a gated check-in build to merge and build up to three submissions.

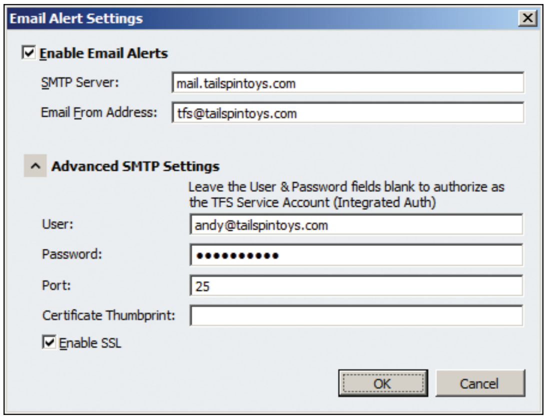
Tailspin Toys case study The Development Team makes heavy use of CI builds. They were selected over gated check-in builds for development in the DEV folder. Everyone agreed that the practice of CI promotes healthier team behaviors than relying on a tool. The team does use gated check-in builds when fixing bugs in code (in the PROD folder) that has been released.

Email alerts

A developer can also monitor builds by enabling an email alert. He or she can register an email address with TFS to receive an email that alerts them to the fact that a build has completed or a build's quality has changed. In fact, alerts can be established to notify when changes occur to work items, code reviews, and source control files, as well as builds. These are just standard emails (in either plaintext or HTML format) that are sent from TFS to a user's inbox using an intermediary Simple Mail Transfer Protocol (SMTP server). Developers can subscribe to alerts for themselves, for others, or for the entire team.

The body of these email alerts contain hyperlinks that can be clicked to take the reader to the respective information in Team Web Access. Emails pertaining to source control, such as check-ins, will display information about the changeset when clicked. Emails about work items will take the reader to the respective work item when clicked. Emails about builds will direct the user to the build in question.

Before TFS can send any alert email, a TFS administrator must configure the server to use an existing SMTP server. This can be accomplished in the Team Foundation Server Administrative Console in the Application Tier section, as shown in Figure 8-9. At a minimum, the administrator must specify the SMTP Server and the Email From Address. In TFS 2012, Microsoft added the ability to specify optional advanced SMTP settings, including User, Password, Port, and additional security information directly from the console. This was a long-anticipated feature in the product.


Email Alert Settings

[✓] Enable Email Alerts

SMTP Server: mail.tailspintoys.com

Email From Address: tfs@tailspintoys.com

Advanced SMTP Settings

Leave the User & Password fields blank to authorize as the TFS Service Account (Integrated Auth)

User: andy@tailspintoys.com

Password: ••••••••••

Port: 25

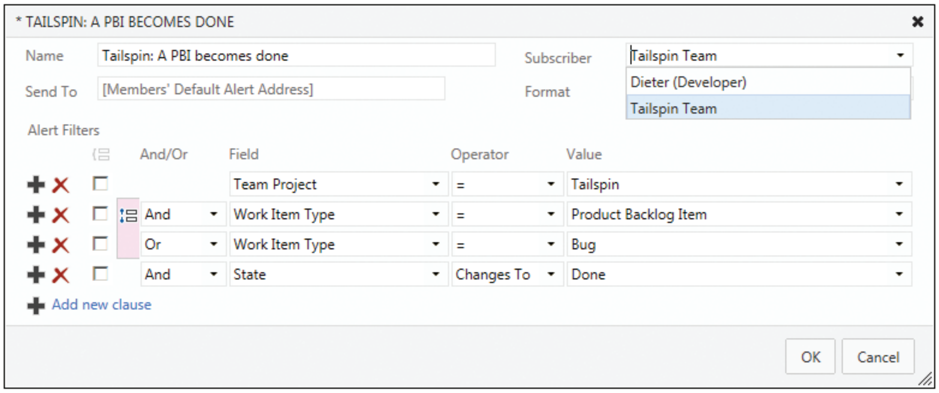
Certificate Thumbprint:

[✓] Enable SSL

FIGURE 8-9 Configuring SMTP settings so that TFS can send alert emails.

Tip The out-of-the-box emails are very plain. They convey the basic information, and not much else. There are no fancy colors or graphics. If a team wanted to, they could add some style or missing functionality to the content and format of the base email alerts by customizing the associated .xsl transform files. The event service uses these files to transform the XML data for an event into a human-readable email message. Editing the respective .xsl file would provide a different format for the email. You should make a backup copy of the transform files before attempting any customization. Better yet, consider creating a separate team project for such an effort so that you can manage changes to those files. For more information check out *http://msdn.microsoft.com/en-us/library/bb552337.aspx*.

Alert subscriptions are stored on the server and organized by team project. A developer can add different alerts for each team project or team that they are a member of. A developer can also configure a *team alert*, which is new in Visual Studio 2012. Team alerts simplify the administration of setting up the same alert for everyone on the team. For example, if all Scrum Team members, including the Product Owner and Scrum Master, want to be informed when a build has completed or when a PBI is "done," someone can create a team alert such as the one shown in Figure 8-10.


* TAILSPIN: A PBI BECOMES DONE

Name: Tailspin: A PBI becomes done

Send To: [Members' Default Alert Address]

Subscriber: Tailspin Team (options: Dieter (Developer), Tailspin Team)

Format

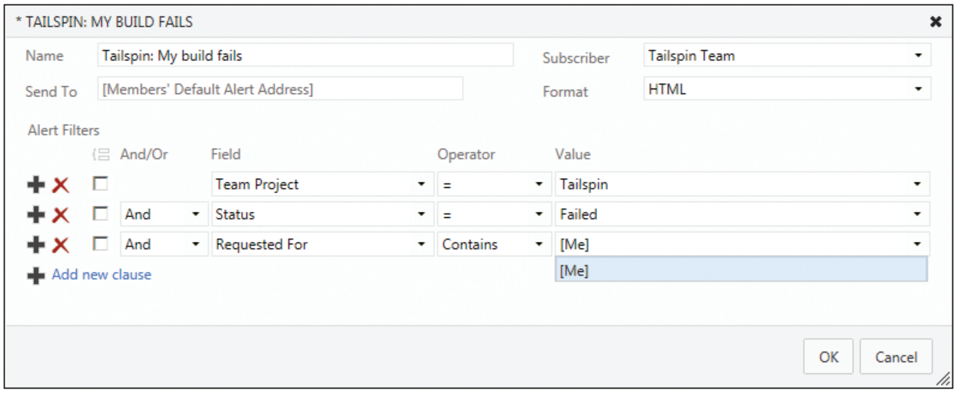
Alert Filters

And/Or	Field	Operator	Value
	Team Project	=	Tailspin
And	Work Item Type	=	Product Backlog Item
Or	Work Item Type	=	Bug
And	State	Changes To	Done

FIGURE 8-10 Creating a team alert so that everyone on the Tailspin Team is notified when a PBI is done.

For build-related alerts, there are two fields that you should be aware of: Requested By and Requested For. The Requested By field is populated by TFS and is always the account that actually queues the build. For manual builds, it contains the user that queues the build, but for CI and scheduled builds, it contains the build service account. If you are interested in knowing who requested the CI build, this won't work. Instead, you should reference the Requested For field. Its behavior is very similar to the Requested By field, except that for CI builds it contains the user who performed the check-in.

Email alerts can also be configured to let a developer, or a whole team, know when a build has completed or failed. This would be an alternative to using the Build Notifications tool previously mentioned. In addition, by using the Requested For field, a single team alert can be created that is smart enough to only email the developer who requested the build, and nobody else. This is done by creating a team alert with the criteria Requested For = [Me], as shown in Figure 8-11. This criteria establishes a behavior which causes an email to be sent to only the person who requested the build. You'll be happy to know that the *[Me]* macro, and its related behavior, is available for all types of alerts.


* TAILSPIN: MY BUILD FAILS

Name: Tailspin: My build fails

Send To: [Members' Default Alert Address]

Subscriber: Tailspin Team

Format: HTML

Alert Filters

And/Or	Field	Operator	Value
	Team Project	=	Tailspin
And	Status	=	Failed
And	Requested For	Contains	[Me]

FIGURE 8-11 Creating a team alert with the *[Me]* macro to email only the developer who broke the build.

Shelving

Shelving lets you set aside a batch of pending changes for whatever reason. It could be that you were interrupted by some more important work, or you want to queue a private build, back up your work, or hand something off to another developer. Shelving can also be used when you want to have your code reviewed.

Shelving produces an artifact called a *shelveset*. Shelvesets exist outside of the normal TFS source control repository and are identified by a unique name provided by the developer who created it. Some point in time later, that developer (or another) can unshelve those pending changes into a local workspace and continue working or review the code.

When a developer shelves his or her code, anybody on the team with the appropriate permissions can view and unshelve those pending changes. In other words, to unshelve a pending change, you must have the *Read* and *Check out* permissions set to *Allow*. For a Scrum Development Team practicing my recommended security configuration, this means that any developer can unshelve another developer's pending changes. This is how it should be on a high-performance Scrum Development Team. Any developer can review any other developer's code without being limited by the tool.

If the developer reviewing the code makes any changes, he or she must create a new shelveset or check in the code. This is because the second developer cannot change the first developer's shelveset. What I have seen happen is that the reviewer will create a second shelveset with the proposed changes and comments, and then the first developer will create a third shelveset, and so on. If the two developers are not diligent about cleaning up their shelvesets as they iterate, there will be a big housekeeping task at the end. For this reason, as well as for general efficiency reasons, code reviews should be performed in person, where all developers look at the same screen at the same time.

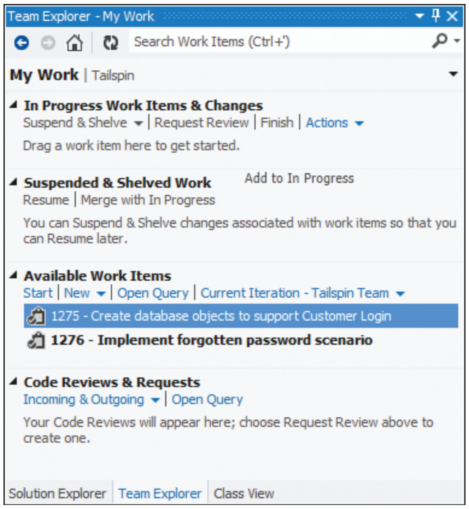
Smell It's a smell if I see a team using shelvesets as a mechanism for code reviews. If it turns out that they are used sparingly for situations where the coder or reviewer are remote, then that is OK. I will suggest, however, that the developers use a screen-sharing utility such as Microsoft Lync or *http://join.me*. By sharing a screen, you avoid the back-and-forth of shelveset creation or the administrivia of creating them or cleaning them up after the exercise.

Tailspin Toys case study The Development Team primarily uses shelvesets for interruptions and private builds. They also use them indirectly with the new suspend and resume features in Visual Studio 2012.

My Work

Visual Studio 2012 Premium and Ultimate edition users can use *My Work* as a way to see and manage their current, in-progress work. As a developer works through his or her tasks in the Sprint Backlog, they can be started in the My Work page. Work can also be suspended and resumed as needed. Code reviews can be requested and managed. Check-ins can be performed. It's a very powerful page within Team Explorer, and every developer on the team should consider using it.

To begin working on a new task, drag it from the Available Work Items section to the In Progress section, as shown in Figure 8-12. You can also right-click the task and add it to In Progress or click the *Start* link. If the task you want to start isn't visible, you may have to run a different query or refresh the results. It may be that you have to create the Task work item first.


Team Explorer - My Work
Search Work Items (Ctrl+')
My Work | Tailspin
In Progress Work Items & Changes
Suspend & Shelve | Request Review | Finish | Actions
Suspended & Shelved Work
Add to In Progress
Resume | Merge with In Progress
Available Work Items
Start | New | Open Query | Current Iteration - Tailspin Team
Code Reviews & Requests
Incoming & Outgoing | Open Query
Solution Explorer
Team Explorer
Class View

Drag a work item here to get started.

You can Suspend & Shelve changes associated with work items so that you can Resume later.

1275 - Create database objects to support Customer Login

1276 - Implement forgotten password scenario

Your Code Reviews will appear here; choose Request Review above to create one.

FIGURE 8-12 Getting started on a new task by dragging it to the In Progress section.

Note Microsoft provides two default queries to get you started in the Available Work Items window: "All Iterations - <project team>" and "Current Iteration - <project team>". The second query is the interesting one. It contains some behind-the-scenes magic to determinate what the current iteration is. If your team has set up the iterations (Sprints) and specified start and end dates, then TFS knows what the current Sprint is. This value is looked up and hard-coded into this query to return Task work items from the current Sprint. This is very convenient, but unfortunately the magic cannot be bottled and reused on other custom queries. Hopefully, Microsoft will give us a *CurrentIteration* macro, or something like it, to use in our queries some day soon.

Dragging the work item to the In Progress section will also change the State to In Progress. More important, it gives you context on what you're doing. For example, even taking 60 seconds to answer a phone call can generate a lengthy "Now where was I?" pause. Being able to see what item you are working on will help return that focus more quickly. If you exit Visual Studio without finishing an In Progress task, it will still be there later when you return.

Smell It's a smell if I see two or more tasks In Progress. Just because Team Explorer allows it, doesn't mean that it makes sense from a work management perspective. Are you really working on two things at once? Perhaps you didn't create the right tasks in the first place. Maybe you switched context and didn't know how to suspend your existing work before starting something new.

Later, when you are done with your task, you can check in your changes and resolve (rather than associate) the task. You can also just click the Finish link in My Work. Both of these methods will transition the work item to the Done state and set any Remaining Hours to zero. If you click Finish, Visual Studio may prompt you with a warning that you haven't checked anything in. For tasks that don't require a check-in, you can dismiss the warning. The task will be removed from My Work, as will the pending changes when you check them in, allowing you to move to your next task.

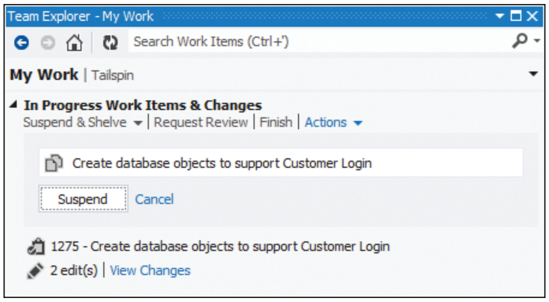
Code reviews can also be requested and managed from the My Work page. I will discuss them later in this chapter.

Suspending and resuming work

From time to time throughout the Sprint (or the day on some dysfunctional teams), a developer will experience an interruption. In a perfect world, this never happens, but in the real world, it does. A high-performance Scrum Development Team works to marginalize this reality by either reducing the number of interruptions or making them less painful. When an interruption does occur, switching context to the new problem can be difficult and wasteful.

For example, let's say that you are deep in thought, implementing a complex scenario within a PBI and the Product Owner drops into the team area with an emergency. It's obvious that an urgent bug fix is required and, as this is critical to the business, you *should* drop what you're doing and fix it. Forget your forecast. It's about saving money and customers at this point. What most developers do is to shelve their code, undo pending changes, and close their solution. Others will just start a new instance of Visual Studio, get the specific version of code that's running in production, and go about locating, verifying, and fixing the bug. Visual Studio 2012 now offers a better way.

From the My Work page, the developer can suspend the current work he or she is doing. Behind the scenes, a shelveset is created to save any pending changes to code, tests, and other files. Important elements of Visual Studio are also saved, such as open windows, breakpoints, and other debug states. The developer assigns the suspended work a friendly name in order to find it easily at some point in the future. By default, this name is the title of the In Progress work item, as you can see in Figure 8-13.


Team Explorer - My Work

Search Work Items (Ctrl+')

My Work | Tailspin

In Progress Work Items & Changes

Suspend & Shelve | Request Review | Finish | Actions

Create database objects to support Customer Login

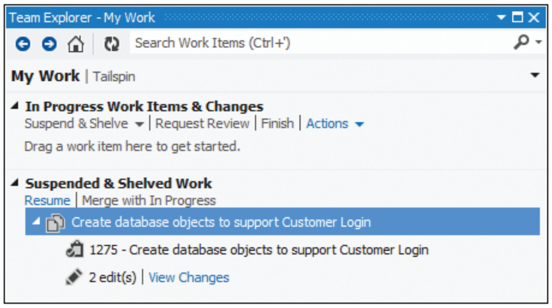
Suspend Cancel

1275 - Create database objects to support Customer Login

2 edit(s) | View Changes

FIGURE 8-13 Suspending in-progress work and giving it a friendly name.

Suspending will shelve any pending changes and then undo the local changes, putting your workspace back into a clean state. It is now ready to handle the new crisis by dragging the new task into the In Progress section. You can see the suspended work listed in the Suspended & Shelved Work section of the My Work page, as shown in Figure 8-14.


Team Explorer - My Work

Search Work Items (Ctrl+')

My Work | Tailspin

In Progress Work Items & Changes

Suspend & Shelve | Request Review | Finish | Actions

Drag a work item here to get started.

Suspended & Shelved Work

Resume | Merge with In Progress

Create database objects to support Customer Login

1275 - Create database objects to support Customer Login

2 edit(s) | View Changes

FIGURE 8-14 Suspended work is given a friendly name and persisted as a shelveset.

Smell It's a smell if I see more than one piece of suspended work listed. Maybe you are the kind of developer who spends more time helping, mentoring, and supporting others. This could explain the various pieces of suspended work. Maybe your organization is so chaotic that even the interruptions get interrupted? Maybe you are just the kind of developer who leaves a bunch of half-eaten sandwiches sitting around your house. That's a different kind of smell.

Later, when the crisis has passed and the developer is able to return to the planned work, he or she can select the suspended work and click the Resume link. The original pending changes will be unshelved, and the task will be put back in the In Progress section. Other IDE settings and behaviors will be restored as well.

In the event that your interruption gets interrupted, you may want to suspend it, and return to your original work. If this is the case, then instead of a Resume link, there will be a Switch link. Clicking Switch will suspend the interruption work, and return context to the original task. Alternatively, you can choose to Merge With In Progress and bring all the pending changes from the two tasks together.

PowerPoint Storyboarding

Visualizations allow the Development Team to elicit feedback more easily. This feedback can come from other members of the Scrum Team, as well as stakeholders, especially domain experts. Shapes and lines drawn on a whiteboard to represent components and actions enable ideas to be vetted by the right people. It's faster to sketch out the high-level concepts and their interactions than it is to try to design or code anything in Visual Studio. It's also cheaper to fix a bug in a drawing than later in code.

As previously mentioned, feedback is important when brainstorming how to tackle a problem such as developing a particular feature or scenario. This is especially true when the developers are not familiar with the domain or the workflow is complex. Having the right people involved in the conversation is critical. The more eyes you can put on a problem, the better the chances of finding an optimal solution. Unfortunately, this is not always possible when the required people are geographically distributed.

Smell It's a smell when the Development Team doesn't have access to the people who know the domain. It's the Product Owner's responsibility to either know the domain or collaborate with experts who do. The Scrum Master might have to get involved to make sure the introductions, communication, and collaboration occur effectively. It's also a smell when the opposite occurs, and the Development Team becomes the domain experts. This is natural in an organization that encapsulates its critical business processes into software. The developers know the software, and thus the domain behind it. This is fine so long as the organization doesn't start using their technical staff as the business help desk.

For Development Teams that love their whiteboards, I recommend setting up a laptop with a webcam in the meeting room or team area. By aiming the webcam at the whiteboard, and then strategically standing out of the way after drawing on it, remote attendees can be part of the design session and discussion. This is less ideal than collaborating in person, but still allows for rapid design with a dry erase pen, rather than fumbling with a software design tool.

When the Development Team has a need to present their ideas to (and gather feedback from) remote stakeholders, then the new PowerPoint Storyboarding feature in Visual Studio 2012 can be beneficial. Users of Visual Studio 2012 Premium, Ultimate, or Test Professional editions can install and use PowerPoint Storyboarding, which allows a developer to illustrate a PBI or a specific feature or scenario using Microsoft PowerPoint. The illustration is created by dragging and dropping predefined, inline images and adding formatted text. It can then be linked to a work item, such as the PBI that it describes, and shared with other TFS users.

Smell It's a smell when I see storyboards created *before* the Sprint in which the PBI gets forecast for development. It could be that the Development Team had to iterate on the design of a complex feature or scenario with a remote stakeholder or two before they were able to estimate it. It could also be that the Development Team started working on this

PBI in a previous Sprint and didn't finish it. From my experiences, the more likely reason is that someone on the team got bored, fired up PowerPoint, and started designing something. When a developer has spare time, he or she should help the rest of the Development Team complete their forecast work. If the whole Development Team has spare time, they should meet with the Product Owner to discuss working on an additional PBI.

Tip Fellow Professional Scrum Developer Jose Luis Soria Teruel has experimented with using storyboards while grooming the Product Backlog. Wary of generating waste, he and the other developers keep them to the right (rough) level of detail. This was a practice that they opted into as a team.

To create a PowerPoint storyboard, there are a few simple steps to follow:

1. Open the PBI work item, choose the Storyboarding tab, and then choose the Start Storyboarding link. You can also start the tool from the Start menu under Microsoft Visual Studio 2012 or by starting PowerPoint directly.
2. Add slides, shapes, and text to the blank presentation to illustrate a PBI, feature, or scenario, as shown with the Customer Login storyboard in Figure 8-15.

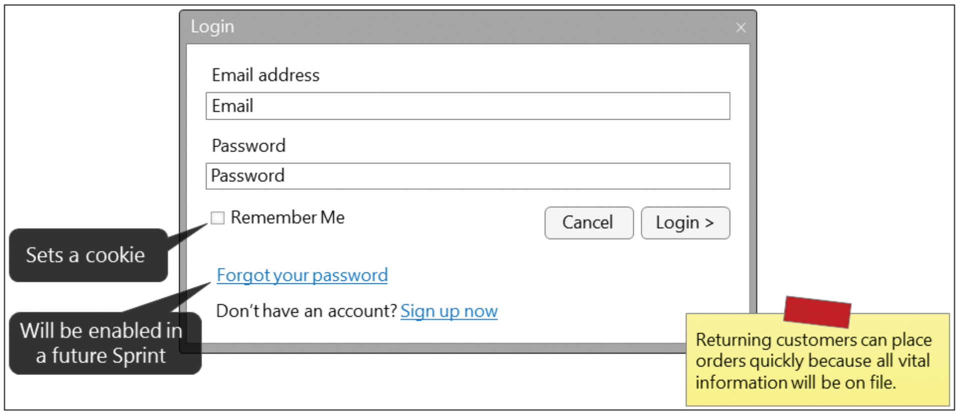
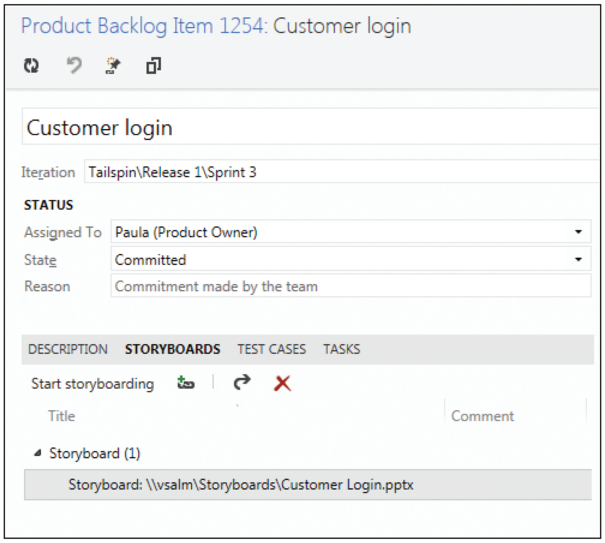

Login
Email address
Email
Password
Password
Remember Me
Cancel
Login >
Forgot your password
Don't have an account? Sign up now
Sets a cookie
Will be enabled in a future Sprint
Returning customers can place orders quickly because all vital information will be on file.

FIGURE 8-15 An example PowerPoint storyboard with annotations.

3. Save the storyboard presentation to a network share or Microsoft SharePoint.
4. (Optional) Link the presentation to the PBI work item that it describes, as shown in Figure 8-16.
5. Share the storyboard with others.


Product Backlog Item 1254: Customer login

Customer login

Iteration: Tailspin\Release 1\Sprint 3

STATUS

Assigned To: Paula (Product Owner)

State: Committed

Reason: Commitment made by the team

DESCRIPTION | STORYBOARDS | TEST CASES | TASKS

Start storyboarding

Title	Comment
Storyboard (1)	
Storyboard: \\vsalm\Storyboards\Customer Login.pptx	

FIGURE 8-16 A PBI work item with a linked PowerPoint storyboard.

6. Others may provide feedback by annotating the PowerPoint document or by using the Feedback client.

As the stakeholders review the storyboard, they can add comments or even make changes to the illustrations using the built-in features of PowerPoint. If the presentation is stored on SharePoint, it can enjoy the dual benefit of broad availability and revision control. Users can check out the presentation and check in any changes. Feedback can also be provided out-of-band, via email, voice, or using the Feedback client, which is covered in the next section.

Tip I'm often asked if a Development Team should use PowerPoint Storyboarding or SketchFlow. On the surface, they appear to be very similar in functionality. SketchFlow is a feature of Expression Studio Ultimate and has a new UI to learn. PowerPoint Storyboarding runs inside PowerPoint, so the learning curve isn't as steep. While SketchFlow is more sophisticated (with a richer set of user controls for designing UIs), it's not as nicely integrated into a development process that uses TFS. Another important difference is that SketchFlow is able to convert (forward-engineer) the prototypes into starter projects. PowerPoint storyboards don't support that. They will always just be illustrations.

Creating a storyboard

To create a storyboard, a developer can select from several layouts that support common user interfaces, such as web and Windows Phone backgrounds. Images can be dragged and dropped from the Storyboard Shapes pane in addition to using all the features available within PowerPoint. These features include clipping and inserting screenshots, hyperlinking from one page to another, animation, inserting images and shapes, and aligning and grouping objects. For example, a developer might create two slides to illustrate the UI for a particular PBI. She might add information about upcoming service appointments to the customer's account page and add buttons that customers can use to schedule, reschedule, and cancel those appointments.

Tip You can save a custom shape to MyShapes and then use it in the same way that you use the predefined storyboard shapes. Also, you can export shapes to share with other developers on the team or import shapes that others have created. Microsoft has also created a Storyboard Shapes Authoring tool to help make storyboard shapes that can be used with PowerPoint Storyboarding. It is available for free at *http://visualstudiogallery.msdn.microsoft.com/75f32d63-8ff2-49f3-b86e-70297d300858*.

Before you can link a storyboard to a work item, you must save it to a shared location. The shared location can be any shared folder on the network or a SharePoint site (such as the team project portal). By linking the storyboard to a work item, you are essentially inviting the rest of your team to access this shared file, so be sure they have the appropriate permissions. They can open the presentation, review it, and add their comments. You can link storyboards only to certain types of work items based on the process template from which your team project was created. In the Visual Studio Scrum process template, you can only link storyboards to Product Backlog Item work items. It is possible to link a storyboard to more than one work item.

Note You cannot create work items from PowerPoint, but you can link to them. This means that if you create the storyboard first, you will have to switch to Visual Studio or Team Web Access to create the PBI so that you may link it. This situation is less likely to occur for a Scrum Team, who should be creating and grooming PBIs a long time before the Sprint.

To create and modify storyboards by using PowerPoint Storyboarding, a developer must have installed either PowerPoint 2007 or later, and one of the following versions: Visual Studio Premium, Visual Studio Ultimate, or Visual Studio Test Professional. Storyboarding is not available in Visual Studio Professional or Express edition. To view storyboards that were created by using the PowerPoint Storyboarding template, users must have PowerPoint 2007 or later installed. They do not need Visual Studio 2012 installed.

Tailspin Toys case study In the past, some developers on the team have used Balsamiq to mock up complex UIs. Over time, the Development Team realized that in-person conversations at a whiteboard provide the most value. They take this approach whenever possible. Occasionally, however, it's not possible because a stakeholder or expert is not available for an in-person discussion. When this happens, they will usually generate and send the storyboard over email or even store them on SkyDrive, allowing the stakeholder to review and comment. Once the feature or scenario is done, the storyboards are deleted.

Feedback client

As you read in the last section, the PowerPoint Storyboarding tool enables a team to create rapidly a UI mockup or illustration of a feature that can be shared with other team members or stakeholders. It's important to close that loop by collecting rich feedback about what those users think of a feature, and whether it is still being brainstormed, under development, or has been released. Feedback should always be welcomed, and even encouraged. If the feature has been released and valid feedback is given, it can be captured in the Product Backlog to be considered for future development.

One of the new features of Visual Studio 2012 is the ability to capture rich stakeholder feedback on features being implemented and bugs being fixed. This is good for distributed organizations who want stakeholders to evaluate the emerging Increment or a design that may still be in flux. The Feedback client is used to gather this type of feedback. It is versatile enough that it can be used to provide feedback on anything the user can see and interact with on the desktop.

Note Users submitting feedback using the Feedback client *do not need* a TFS Client Access License (CAL). A Windows Server CAL may still be required, however. Please refer to the latest version of the Visual Studio 2012 licensing white paper at *http://go.microsoft.com/fwlink/?LinkID=246172*.

This type of feedback can either be formally requested via a work item and email sent from Visual Studio, or it can be provided voluntarily, without solicitation. We will look at both scenarios shortly.

Requesting feedback

The first feedback scenario occurs when a member of the Scrum Team, preferably the Product Owner, solicits feedback from one or more stakeholders. These stakeholders will receive a feedback request through an email that is constructed from the feedback request form. From the email, the stakeholders can install and launch the Feedback client tool, which guides them in providing and capturing their feedback. TFS stores this feedback as a Feedback Response work item.

In order to request feedback, TFS must be configured to use an existing SMTP server in order to send emails. This requirement was mentioned earlier in the chapter in the context of setting up email alerts and, hopefully, it is already configured. To begin, click the Request Feedback link on the Team Web Access home page, as shown in Figure 8-17.

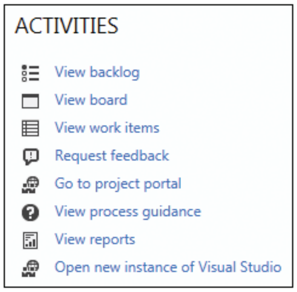
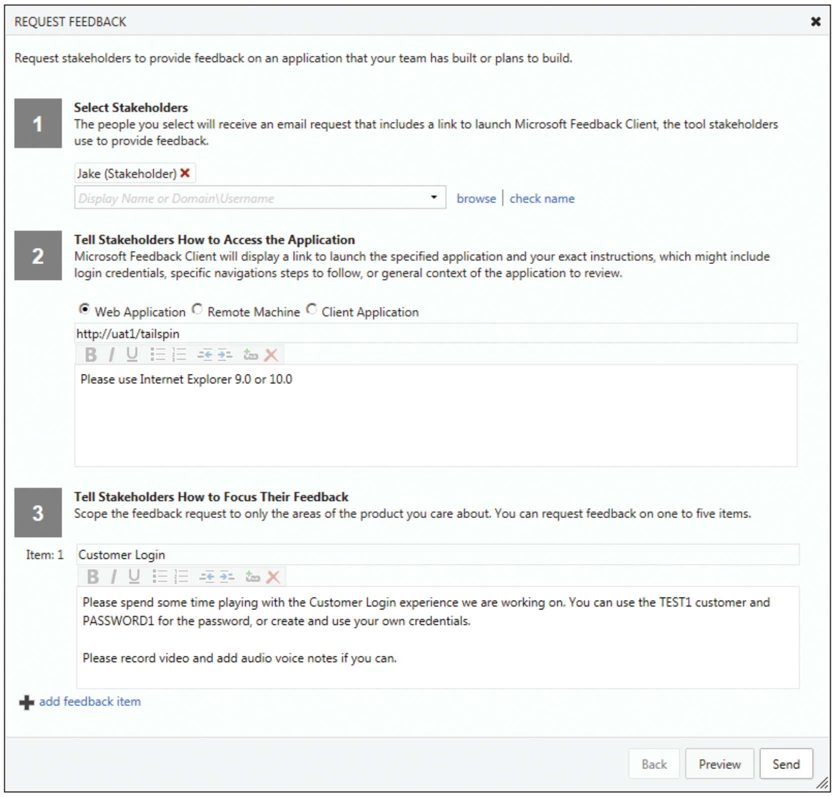


FIGURE 8-17 The Request Feedback link on the Team Web Access home page.

Feedback can be requested on any aspect of the product, from the entire application down to a specific scenario within a feature. Because the feedback request is essentially an email, the requester can be as ambiguous or as specific as he or she wants to be. In addition, one request can be partitioned to ask for feedback on up to five discrete items. For example, if the Development Team is code-complete on three scenarios within a PBI, a request could be created that contains three items—one for each scenario the Product Owner desires feedback on.

Note Regardless of the size and scope of the request, the stakeholders must be able to access physically the application and feature(s) in question, and they must have the time and know-how to do it. This should be considered as the feedback request is created.

When creating the feedback request, one or more stakeholders must be selected. These users must have an email address associated with their user name. Users without email addresses won't be sent a request. The stakeholders should also be told *how* to access the application in question. An address and instructions can be provided for a web application, (rich) client application, or a remote machine. Finally, the item(s) to be evaluated and any related notes are added to the request. Figure 8-18 shows a feedback request ready to be sent to a stakeholder to evaluate the Customer Login feature of a web application.


REQUEST FEEDBACK

Request stakeholders to provide feedback on an application that your team has built or plans to build.

1 **Select Stakeholders**
The people you select will receive an email request that includes a link to launch Microsoft Feedback Client, the tool stakeholders use to provide feedback.

Jake (Stakeholder)

Display Name or Domain\Username

2 **Tell Stakeholders How to Access the Application**
Microsoft Feedback Client will display a link to launch the specified application and your exact instructions, which might include login credentials, specific navigations steps to follow, or general context of the application to review.

[✓ Web Application] Remote Machine Client Application

http://uat1/tailspin

Please use Internet Explorer 9.0 or 10.0

3 **Tell Stakeholders How to Focus Their Feedback**
Scope the feedback request to only the areas of the product you care about. You can request feedback on one to five items.

Item: 1 Customer Login

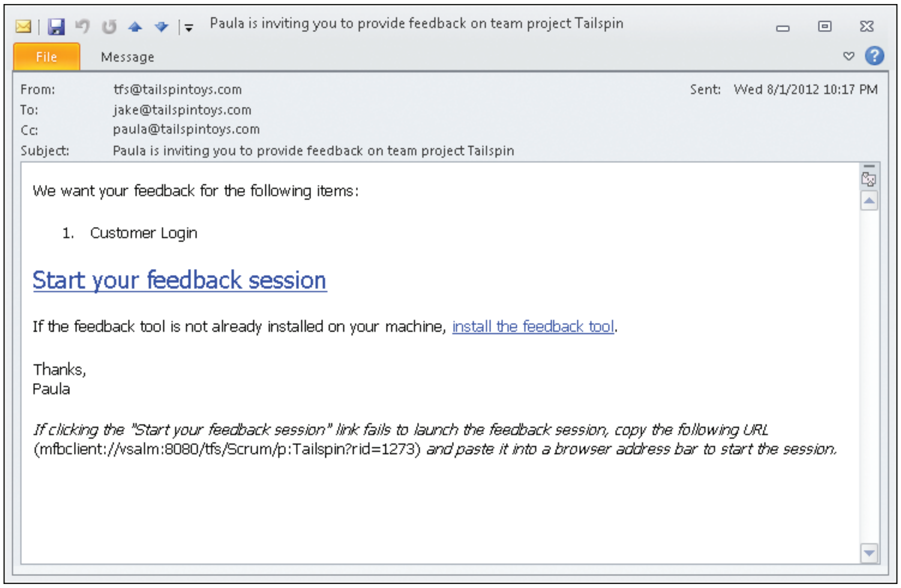
Please spend some time playing with the Customer Login experience we are working on. You can use the TEST1 customer and PASSWORD1 for the password, or create and use your own credentials.

Please record video and add audio voice notes if you can.

FIGURE 8-18 Creating a request for feedback on the Customer Login feature.

Tip Consider previewing the request before sending it. It will show what the email that the stakeholder(s) receive will look like and allow you to customize it. It will also show the email addresses rather than the user names, so you can see if there are any discrepancies, such as wrong or missing email addresses associated with the user names. For example, if you add a stakeholder by user name and that user doesn't have an email address associated with his or her account, you will receive an error message like this: *TF400596: Cannot find email addresses for the following recipient(s): 'Chuck'*. If this occurs, you can just add the email address manually and continue with the request. However, you should ask the stakeholder to update his or her profile and provide a valid email address to avoid this error in the future.

As the feedback requester, you will receive a copy of the email submission automatically when you send it. You can also add other email addresses in the To box when previewing the email. Figure 8-19 shows a sample email requesting feedback. If an administrator has not granted permissions to the accounts of those stakeholders that you add, they will not be able to provide feedback through the Feedback client.


Paula is inviting you to provide feedback on team project Tailspin
File
Message

From: tfs@tailspintoys.com
To: jake@tailspintoys.com
Cc: paula@tailspintoys.com
Subject: Paula is inviting you to provide feedback on team project Tailspin
Sent: Wed 8/1/2012 10:17 PM

We want your feedback for the following items:

1. Customer Login

Start your feedback session

If the feedback tool is not already installed on your machine, install the feedback tool.

Thanks,
Paula

If clicking the "Start your feedback session" link fails to launch the feedback session, copy the following URL (mfbclient://vsalm:8080/tfs/Scrum/p:Tailspin?rid=1273) *and paste it into a browser address bar to start the session.*

FIGURE 8-19 A sample email sent to a stakeholder requesting feedback.

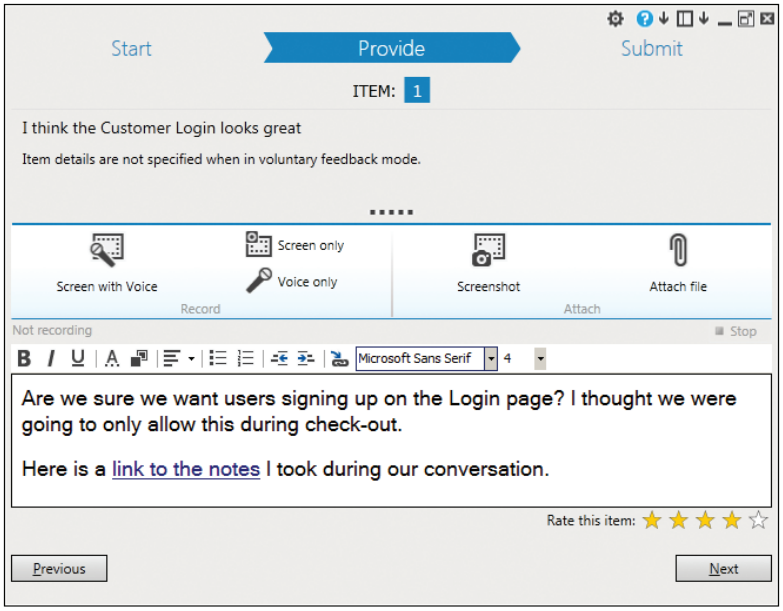
Providing feedback

When the stakeholder receives the request, he or she should first make sure that the Feedback client is installed. If this is the first time providing feedback, it will need to be installed. The email contains a hyperlink to download it, if necessary. Next, the stakeholder starts the feedback session by clicking the large hyperlink in the email, or copying and pasting the supplemental URL into the web browser.

As the stakeholder reviews the new feature, he or she is able to perform the following tasks using the Feedback client:

- Record video of the interaction with the application.
- Record voice comments.
- Capture a screenshot.
- Annotate a screenshot using a program such as Microsoft Paint.
- Type comments.
- Attach a file.
- Rate each item of feedback on a scale of 1–5 stars.

On the Provide page of the Feedback client, one or more items appear for the user to provide feedback. For each item, he or she can get context on what's being asked and then provide free-form feedback through any of the aforementioned methods of input. Figure 8-20 shows the various recording options. If there are multiple items, clicking Next will advance to the next item for which to provide feedback. Recordings appear as images within the Feedback client's text box.



I think the Customer Login looks great

Item details are not specified when in voluntary feedback mode.

Are we sure we want users signing up on the Login page? I thought we were going to only allow this during check-out.

Here is a link to the notes I took during our conversation.

FIGURE 8-20 The Feedback client provides many ways to record and attach your feedback.

By annotating screenshots, the reviewer can indicate corrections or improvements by adding text or images to the screenshot that was captured. By default, Paint opens automatically when the user opens a screenshot image that was captured within the Feedback client. Another annotation tool, such as Paint.NET or Snagit, can be configured instead by clicking on the cog icon at the top of the feedback tool, as shown in Figure 8-21. After feedback has been provided for each item, the user can review, make corrections or additions, and then submit the feedback to the requesting user via TFS.

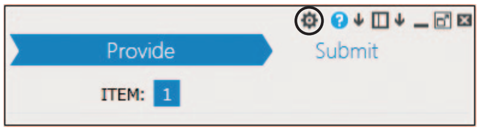


FIGURE 8-21 Click the cog icon to configure your annotation tool.

Tip Be careful when recording sensitive data, such as user names, passwords, account numbers, etc. If the recording is going, everything will be captured. If you do record sensitive data, you can delete the recording by deleting its representative image in the text box and then record it again.

In order for stakeholders to be able to provide feedback, an administrator must grant them specific permissions in TFS. They can either be added to the Limited license group in Team Web Access or a custom group with specific permissions. The Limited group is provided specifically to support access to TFS for users who do not need a CAL. If the stakeholders have a CAL and you are not going to use the Limited group, then make sure to grant the minimum permissions required, which are project-level permissions to create and view test runs and view project-level information, as well as area path permissions to view and edit work items in the respective nodes.

Regardless of which permissions approach you take, you should try to group the feedback stakeholders together in their own Windows group. Because providing feedback is probably the only way that they will interact with TFS, keeping them grouped together will simplify management and allow the Scrum Team to know exactly who their feedback stakeholders are.

Feedback requests generate a Feedback Request work item assigned to the creator of the request. The Description field contains the body of the email that was sent. Feedback Response work items are created to hold the feedback provided by the stakeholder using the tool. Remember that both Feedback Request and Feedback Response work item types are designated as Hidden types. This means that they cannot be created directly from Visual Studio or Team Web Access. Instead, they are created using the appropriate tool, such as the Request Feedback link and Feedback client respectively.

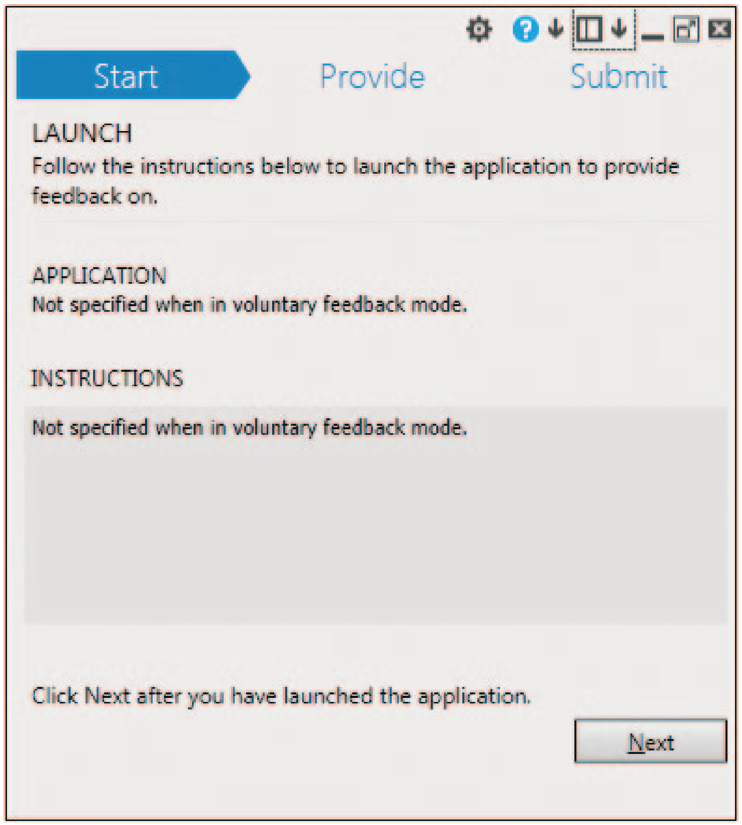
Smell It's a smell when the Development Team solicits stakeholder feedback directly. Gathering feedback from stakeholders is the responsibility of the Product Owner, not the Development Team. If the developers want to seek feedback from stakeholders or other domain experts, they should do it with the blessing of the Product Owner. If necessary, the Scrum Master can help facilitate this. Visual Studio, however, doesn't know about the rules of Scrum, and it allows anyone to request or provide feedback. If the Feedback client is being used inconsistently with the rules of Scrum, the Scrum Team should discuss it during the next Sprint Retrospective meeting and adapt accordingly.

Tailspin Toys case study The Development Team does not use this feature, but Paula (the Product Owner) does. She will sometimes send a request for feedback to a stakeholder in a remote office, along with a link to the test website. Andy (the TFS administrator) will add any new stakeholders to the Limited license group in Team Web Access so that they have adequate permissions.

Voluntary feedback

Another way to use the Feedback client to provide feedback is for a stakeholder to start it directly. It can be found on the Start menu under Visual Studio 2012. If it's missing, it can be downloaded from Microsoft.

When started, the client will be in *voluntary feedback mode*. There won't be any associated request or instructions, as you can see in Figure 8-22. Hopefully, the stakeholder will already know what application to start, what features or scenarios to evaluate and provide feedback on, and what team project to submit the feedback response to.



LAUNCH
Follow the instructions below to launch the application to provide feedback on.

APPLICATION
Not specified when in voluntary feedback mode.

INSTRUCTIONS
Not specified when in voluntary feedback mode.

Click Next after you have launched the application.

FIGURE 8-22 Feedback client running in voluntary feedback mode.

Feedback that has been submitted voluntarily like this can be found in TFS by running the Feedback shared query. This query returns work items that are in the Microsoft.FeedbackResponse Category work item type category. In the Visual Studio Scrum process template, this would only include Feedback Response work items.

When viewing a Feedback Response work item, you will see many of the standard fields, such as *title, created by, state, rating, area*, and *iteration*. The more interesting data will be in the Notes field, as it contains the comments typed by the stakeholder and any references to audio, video, screenshots, or attached files. There won't be any linked stories (PBIs), but the developer can add them as needed. Be aware that any files attached in the Feedback client will appear as Result Attachments links on the All Links tab and not as true work item attachments.

Note Currently, the Feedback client doesn't capture and persist system information. It was available during the beta version of Visual Studio 2012, but it was later removed. Microsoft is considering enabling it in a future update, along with the ability to disable it selectively for organizations that are sensitive to this kind of data being collected.

Tailspin Toys case study During the Sprint, completed features are deployed to a dedicated acceptance testing environment where the Scrum Team (as well as stakeholders) can use the system and provide feedback. Once Paula's remote stakeholders know how to use the Feedback client, they may drop in on the deployed website periodically and provide feedback. The Development Team has created an email alert that watches for new Feedback Response work items being created to let everyone know when an unsolicited, voluntarily provided piece of feedback arrives.

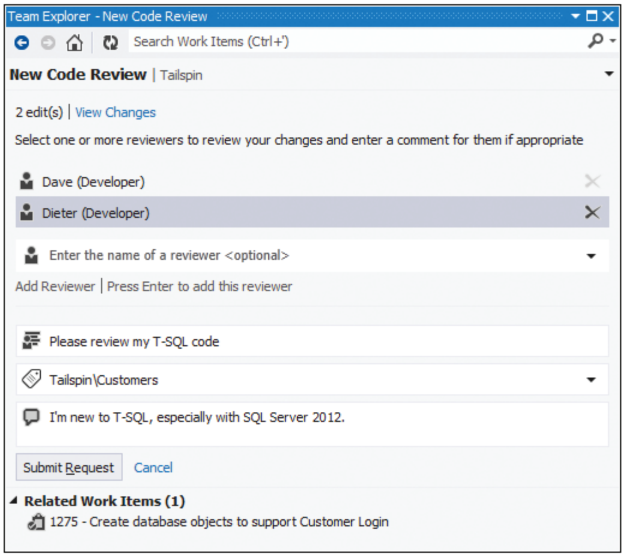
Code reviews

As we discussed earlier in this chapter, code reviews and pair programming are two ways that developers can collaborate to help assure higher code quality. These practices also reduce the risk of creating bugs, technical debt, and gold plating. Visual Studio Premium and Ultimate edition users can use Visual Studio to facilitate code reviews.

Smell It's a smell when I see a *collocated* Development Team using tools to facilitate code reviews. They should be able to practice these reviews in person. Excuses are usually to the effect of "But the developers are busy right now" or "It would be rude to interrupt them." It's obvious that they want to use the asynchronous behavior that the tool provides. I understand that there's a cost to interruptions, and that instant messaging (IM) and Short Message Service (SMS) texts are good for quick questions. Code reviews are not quick interruptions. They require a full stop and context shift in order for the review to have everyone's full attention. As I've mentioned several times in this chapter, conversations that take place face to face are more efficient, reduce ambiguity and misunderstanding, and provide more value than anything facilitated by a tool.

Tip Fellow Professional Scrum Developer Jose Luis Soria Teruel sometimes uses the Code Review tool to ask people *outside* the Development Team to review the code. It's useful to get the opinion of someone not working directly on the code, especially where new technologies are concerned. The Code Review tool provides the opportunity to involve an expert in the matter being reviewed.

From the My Work page, you can request a code review of work that currently has a state of In Progress or that has been suspended. You can also request a code review on a shelveset or changeset. Code reviews can be requested from various other pages and menus as well. Let's focus on the scenario where we want another developer to review some code that is currently In Progress. Assuming that there are pending changes on one or more files, the coder will click the Request Review link from the My Work page. Next, he or she selects one or more reviewers to send the request to. He or she can specify a friendly name for the code review, the area path, and a helpful comment, as shown in Figure 8-23. Unlike sending a request for feedback, this feature just assigns work items to the other TFS users. No email is sent.


Team Explorer - New Code Review
Search Work Items (Ctrl+')
New Code Review | Tailspin
2 edit(s) | View Changes
Add Reviewer | Press Enter to add this reviewer
Submit Request
Cancel
Related Work Items (1)

Select one or more reviewers to review your changes and enter a comment for them if appropriate

Dave (Developer)

Dieter (Developer)

Enter the name of a reviewer <optional>

Please review my T-SQL code

Tailspin\Customers

I'm new to T-SQL, especially with SQL Server 2012.

1275 - Create database objects to support Customer Login

FIGURE 8-23 Creating a new code review request for two other developers.

Tip Each code review recipient must have access to the files in TFS. In other words, if some files are off limits to a particular developer, don't ask her to review your changes to those files. She won't get very far. Also, it is possible to add yourself as a reviewer. Microsoft enabled this particular workflow so that you could add comments on your own code to explain the context before the review is sent to others. When those reviewers receive the code review request, they can read your comments first to obtain context and understanding.

When the request is submitted, a Code Review Request work item is created and assigned to the requester. In addition, one or more Code Review Response work items are created and assigned to the individual developers being asked to review the code. All of these work items are in the Requested state. Code Review Request and Code Review Response work item types are designated as Hidden types. This means that they cannot be created directly from Visual Studio or Team Web Access. Instead, they are created and managed using the appropriate tooling in Team Explorer. While you can query and open one of these work item types in Team Explorer or Team Web Access, the data in the form is read only.

The prospective reviewers will see the incoming request in the Code Reviews & Requests section of their My Work window. You can see an example of this in Figure 8-24. A number in parentheses shows, at a glance, how many code reviews are being displayed in the view. This is a quick way to see if any code reviews need your attention. There are several available views that can be selected to show code reviews in different ways. If you are curious, you can click the Open Query link to see the work item query (WIQ) behind any of the views.

Here is a list of the built-in views:

- **Incoming Requests** Shows active code reviews in which you are a reviewer.
- **Outgoing Requests** Shows active code reviews that you have requested.
- **Incoming & Outgoing** Shows both incoming and outgoing code reviews. This is the default view.
- **Recently Finished** Shows code reviews that have been completed in the last seven days.

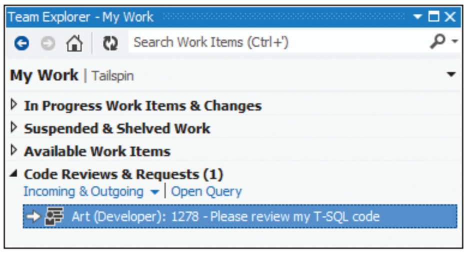
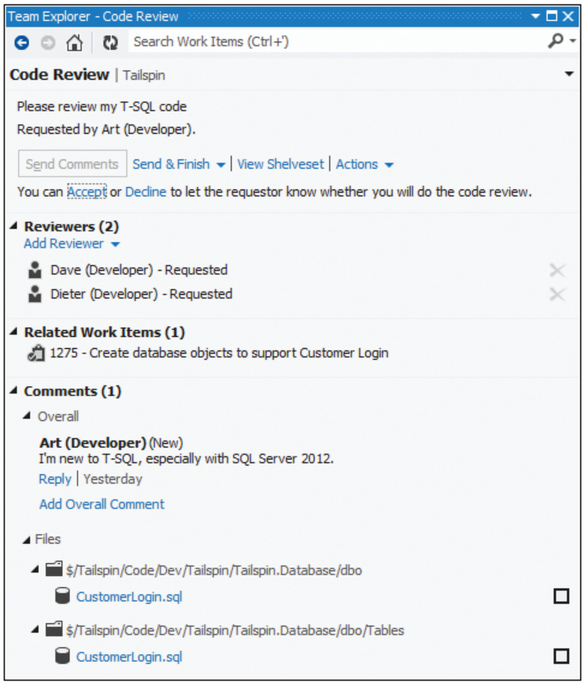

Team Explorer - My Work
Search Work Items (Ctrl+')
My Work | Tailspin
In Progress Work Items & Changes
Suspended & Shelved Work
Available Work Items
Code Reviews & Requests (1)
Incoming & Outgoing | Open Query
Art (Developer): 1278 - Please review my T-SQL code

FIGURE 8-24 My Work page showing an incoming code review request.

When a request appears, the prospective code reviewer should open it to learn more. This opens the Code Review page in Team Explorer, as you can see in Figure 8-25. On this page, the reviewer can accept or decline the request by clicking the respective link towards the top. If the developer chooses to decline the request, he or she can provide a reason for declining the request.


Team Explorer - Code Review

Search Work Items (Ctrl+')

Code Review | Tailspin

Please review my T-SQL code
Requested by Art (Developer).

Send Comments | Send & Finish | View Shelveset | Actions

You can Accept or Decline to let the requestor know whether you will do the code review.

Reviewers (2)
Add Reviewer
Dave (Developer) - Requested
Dieter (Developer) - Requested

Related Work Items (1)
1275 - Create database objects to support Customer Login

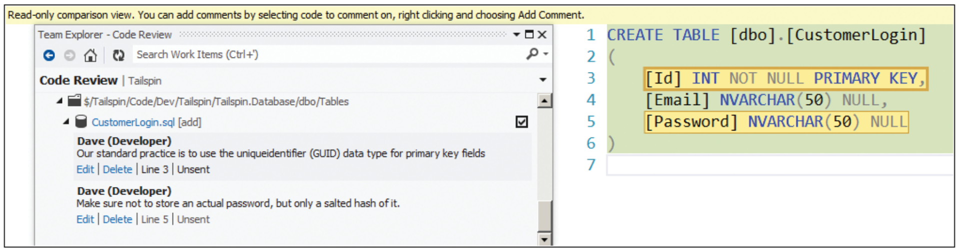
Comments (1)
Overall
Art (Developer) (New)
I'm new to T-SQL, especially with SQL Server 2012.
Reply | Yesterday
Add Overall Comment

Files
$/Tailspin/Code/Dev/Tailspin/Tailspin.Database/dbo
CustomerLogin.sql
$/Tailspin/Code/Dev/Tailspin/Tailspin.Database/dbo/Tables
CustomerLogin.sql

FIGURE 8-25 Opening a code review request in Team Explorer.

Reviewing code within Visual Studio consists of performing one or more of the following activities:

- View the associated shelveset or changeset that contains the code.
- Open and review the associated Task work items.
- Add additional reviewers or remove current reviewers.
- Add overall comments.
- Comment on another's overall comment.
- Review the individual files and add inline comments (as shown in Figure 8-26).
- Check the boxes next to each file to ensure everything is reviewed.
- Finish the review.


Read-only comparison view. You can add comments by selecting code to comment on, right clicking and choosing Add Comment.

Team Explorer - Code Review

Code Review | Tailspin

$/Tailspin/Code/Dev/Tailspin/Tailspin.Database/dbo/Tables

CustomerLogin.sql [add]

Dave (Developer)
Our standard practice is to use the uniqueidentifier (GUID) data type for primary key fields
Edit | Delete | Line 3 | Unsent

Dave (Developer)
Make sure not to store an actual password, but only a salted hash of it.
Edit | Delete | Line 5 | Unsent

```
1 CREATE TABLE [dbo].[CustomerLogin]
2 (
3     [Id] INT NOT NULL PRIMARY KEY,
4     [Email] NVARCHAR(50) NULL,
5     [Password] NVARCHAR(50) NULL
6 )
7
```

FIGURE 8-26 Adding two separate comments about the CustomerLogin table.

Once the reviewer is done with the review, he or she can complete and send it with an overall opinion. The opinion choices are Looks Good, With Comments, or Needs Work. At this point, the work item will be closed and the person named in the Assigned-To field will be removed. At any time, the code review requester can expand the outbound request in his or her My Work page and see if the prospective reviewers have accepted, declined, ignored, or finished the request. The requester can also complete the code review as a whole at any time by closing it or abandoning it.

Tailspin Toys case study The Development Team has been performing code reviews and pair programming for some time now. They don't make either practice mandatory, but let the individual developers decide which will serve them best. They rarely use the code review features in Visual Studio, opting for in-person reviews instead. Occasionally, however, they have used this tooling when team members are on the road or otherwise working remotely.

Chapter burndown

Here are the key concepts we covered in this chapter:

- **Collaboration is key** Software development is a team sport. The Scrum Team needs to communicate with each other, as well as stakeholders, effectively.
- **Active listening** Communication techniques that enable better, more effective dialogue.
- **Collocated teams** Development Teams working in close proximity are more productive and generate more business value than teams that are geographically distributed. Large, open-space team rooms can be particularly effective.
- **Meet effectively** Scrum has all the built-in events (meetings) that a Development Team needs. Limit attendance to other meetings, or send the Scrum Master instead.
- **Limit interruptions** Turn off or otherwise neutralize cell phones, email clients, and IM/chat clients. Limit Internet searches and attending non-essential meetings.

- **Collective code ownership** The entire Development Team owns every aspect of the code. Everyone can read, check out, or check in code for any assembly, namespace, or class. TFS will effectively track all changes made.
- **Comments** When commenting code, be sure to explain your actions to others, assuming the code and/or check-in comments can't do it for you.
- **Code reviews** Practice these in person, or consider pair programming as an alternative. Developers should be open to giving and receiving criticism. Use the code review features in Visual Studio only when in-person reviews are not possible.
- **Continuous integration** Merging is painful, so do it more often so it hurts less. Stay in touch with your builds, especially when they fail. Get them healthy again as soon as possible.
- **Builds check-in policy** This check-in policy requires that the last build was successful for each affected CI build definition.
- **Build Notification tool** Configure and use this to receive alerts from TFS in your notification area (system tray) when a build queues, starts, or completes.
- **Gated check-in build** Use this on production code to ensure that the codeline stays healthy. CI is a better practice for the active development codeline.
- **Email alerts** TFS can be configured to send individuals or the entire team an email when something interesting happens, like a build breaking.
- **My Work** A page in Team Explorer that enables a developer to see and manage their current, in-progress work. The page is available to Visual Studio 2012 Ultimate or Premium users. Work can be suspended and resumed as other priorities crop up.
- **PowerPoint storyboards** Mockups and illustrations can be created in a familiar environment and shared with remote stakeholders to obtain their feedback.
- **Feedback client** A freely downloadable, lightweight tool that enables desktop video, audio, screenshots, and notes to be recorded as a stakeholder evaluates a piece of software. This feedback can be requested, or it can be offered voluntarily, without solicitation.

Index