Foreword by **Brian Harry,** Microsoft Technical Fellow

Visual Studio Team Foundation Server 2012 Adopting Agile Software Practices

From Backlog to Continuous Feedback

Sam **Guckenheimer** & Neno **Loje**

Praise for
Visual Studio Team Foundation Server 2012: Adopting Agile Software Practices

"Agile dominates projects increasingly from IT to product and business development, and Sam Guckenheimer and Neno Loje provide pragmatic context for users seeking clarity and specifics with this book. Their knowledge of past history and current practice, combined with acuity and details about Visual Studio's Agile capabilities, enable a precise path to execution. Yet their voice and advice remain non-dogmatic and wise. Their examples are clear and relevant, enabling a valuable perspective to those seeking a broad and deep historical background along with a definitive understanding of the way in which Visual Studio can incorporate Agile approaches."

—**Melinda Ballou**, Program Director, Application Lifecycle Management and Executive Strategies Service, International Data Corporation (IDC)

"Sam Guckenheimer and Neno Loje have forgotten more about software development processes than most development 'gurus' ever knew, and that's a good thing! In *Visual Studio Team Foundation Server 2012*, Sam and Neno distill the essence of years of hard-won experience and hundreds of pages of process theory into what really matters—the techniques that high-performance software teams use to get stuff done. By combining these critical techniques with examples of how they work in Visual Studio, they created a de-facto user guide that no Visual Studio developer should be without."

—**Jeffrey Hammond**, Principal Analyst, Forrester Research

"If you employ Microsoft's Team Foundation Server and are considering Agile projects, this text will give you a sound foundation of the principles behind its Agile template and the choices you will need to make. The insights from Microsoft's own experience in adopting Agile help illustrate challenges with scale and the issues beyond pure functionality that a team needs to deal with. This book pulls together into one location a wide set of knowledge and practices to create a solid foundation to guide the decisions and effective transition, and will be a valuable addition to any team manager's bookshelf."

—**Thomas Murphy**, Research Director, Gartner

"This book presents software practices you should want to implement on your team and the tools available to do so. It paints a picture of how first-class teams *can* work, and in my opinion, is a must-read for anyone involved in software development. It will be mandatory reading for all our consultants."

—**Claude Remillard**, President, InCycle

"This book is the perfect tool for teams and organizations implementing Agile practices using Microsoft's Application Lifecycle Management platform. It proves disciplined engineering and agility are not at odds; each needs the other to be truly effective."

—**David Starr**, Scrum.org

"Sam Guckenheimer and Neno Loje have written a very practical book on how Agile teams can optimize their practices with Visual Studio. It describes not only how Agile and Visual Studio work, but also the motivation and context for many of the functions provided in the platform. If you are using Agile and Visual Studio, this book should be a required read for everyone on the team. If you are not using Agile or Visual Studio, then reading this book will describe a place that perhaps you want to get to with your process and tools."

—**Dave West**, Analyst, Forrester Research

"Sam Guckenheimer and Neno Loje are leading authorities on Agile methods and Visual Studio. The book you are holding in your hand is the authoritative way to bring these two technologies together. If you are a Visual Studio user doing Agile, this book is a must-read."

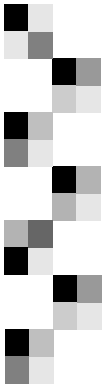
—**Dr. James A. Whittaker**, Software Engineering Director, Google

"Agile development practices are a core part of modern software development. Drawing from our own lessons in adopting Agile practices at Microsoft, Sam Guckenheimer and Neno Loje not only outline the benefits, but also deliver a hands-on, practical guide to implementing those practices in teams of any size. This book will help your team get up and running in no time!"

—**Jason Zander**, Corporate Vice President, Microsoft Corporation

Visual Studio Team Foundation Server 2012: Adopting Agile Software Practices

This page intentionally left blank



Visual Studio Team Foundation Server 2012: Adopting Agile Software Practices

From Backlog to Continuous Feedback

Sam Guckenheimer
Neno Loje

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The Library of Congress cataloging-in-publication data is on file.

ISBN-13: 978-0-321-86487-1
ISBN-10: 0-321-86487-5

Text printed in the United States on recycled paper at R.R. Donnelly in Crawfordsville, Indiana.

First printing September 2012

To Monica, Zoe, Grace, Eli, and Nick,
whose support made this book possible.
—Sam

This page intentionally left blank

Contents

Foreword to Third Edition

Sam and I met in 2003 over a storyboard. We were on this new team with a mission to take Visual Studio—the world's leading individual development environment—and turn it into the world's leading team development environment. He had just joined Microsoft and wanted to convince me that we would succeed not just by building the best tools, but by creating the best end-to-end integration, and he used a storyboard to show the ideas.[1] He convinced me and the rest of the team.

He also persuaded us that we should think of enabling Agile process flow from the beginning. A key idea was that we minimize time spent in transitions. We would build our tools to make all their data transparent and squeeze as much waste and overhead as possible out of the software process. That way, the team could focus on delivering a flow of value to its customers.

That vision, which we now call the Agile Consensus, informed the first versions of Team Foundation Server and Visual Studio Team System, as our product line was called in 2005. The first edition of this book was the explanation of the reasons we made the leap.

Neno was one of our first customers and consultants. He quickly became one of our strongest advocates and critics, seizing the vision and identifying the gaps that we would need to fill release after release. He is now one of the clear experts in the product line, knowing as much about how our customers use the Visual Studio product line as just about anyone.

Since we released v1, we've also been on our own journey of Agile transformation across Microsoft Developer Division. Our products have helped

[1] In VS 2012, we made storyboarding a part of the product too.

make that change possible. First, in the wave to VS 2008, we set out to apply Agile software engineering practices at scale. You'll recognize these in our product capabilities today like unit testing, gated check-in, parallel development, and test lab management. These helped us reduce waste and create trustworthy transparency. Once we achieved those goals, we could set out to really increase the flow of customer value, in the second wave leading to VS 2010. And most recently, with VS 2012, we have really addressed our cycle time. The clearest example of this is the hosted Team Foundation Service, which is now deploying to customers every three weeks. Chapter 9 tells this story well.

Together, Sam and Neno have written the *why* book for modern software practices and their embodiment in the Visual Studio product line. This book does not attempt to replace the product documentation by telling you which button to click. Rather, it covers your software life cycle holistically, from the backlog of requirements to deployment, and shows examples of how to apply modern best practices to develop the right thing.

If you are an executive whose business depends on software, then you'll want to read this book. If you're a team lead trying to improve your team's velocity or the fit of your software to your customers' needs, read this book. If you're a developer or tester, and you want to work better, with more time on task, and have more fun, read this book.

Brian Harry
Microsoft Technical Fellow
General Manager, Team Foundation Server

Foreword to Second Edition

It is my honor to write a foreword for Sam's book, *Agile Software Engineering with Visual Studio*. Sam is both a practitioner of software development and a scholar. I have worked with Sam for the past three years to merge Scrum with modern engineering practices and an excellent toolset, starting with Microsoft's VS 2010. We are both indebted to Aaron Bjork of Microsoft, who developed the Scrum template that instantiates Scrum in Visual Studio through the Scrum template.

I do not want Scrum to be prescriptive. I left many holes, such as what is the syntax and organization of the product backlog, the engineering practices that turned product backlog items into a potentially shippable increment, and the magic that would create self-organizing teams. In his book, Sam has superbly described one way of filling in these holes. He describes the techniques and tooling, as well as the rationale of the approach that he prescribes. He does this in detail, with scope and humor. As I have worked with Microsoft since 2004 and Sam since 2009 on these practices and tooling, I am delighted. Our first launch was a course, the Professional Scrum Developer .NET course, that taught developers how to use solid increments using modern engineering practices on VS (working in self-organizing, cross-functional teams). Sam's book is the bible to this course and more, laying it all out in detail and philosophy. If you are on a Scrum team building software with .NET technologies, this is the book for you. If you are using Java, this book is compelling enough to read anyway, and may be worth switching to .NET.

When we devised and signed the Agile Manifesto in 2001, our first value was "Individuals and interactions over processes and tools." Well, we have

the processes and tools nailed for the Microsoft environment. In Sam's book, we have something developers, who are also people, can use to understand the approach and value of the processes and tools. Now for the really hard work, people. After 20 years of being treated as resources, becoming accountable, creative, responsible people is hard. Our first challenge will be the people who manage the developers. They could use the metrics from the VS tooling to micromanage the processes and developers, squeezing the last bit of creativity out and leaving agility flat. Or, they could use the metrics from the tools to understand the challenges facing the developers. They could then coach and lead them to a better, more creative, and more productive place. This is the challenge of any tool. It may be excellent, but how it is used will determine its success.

Thanks for the book, Sam and Neno.

Ken Schwaber
Co-Creator of Scrum

Preface

Seven years ago, we extended Microsoft Visual Studio to include Application Lifecycle Management (ALM). This change made life easier and more productive for hundreds of thousands of our users and tens of thousands of our Microsoft colleagues. In 2010, when we shipped Visual Studio 2010 Premium, Ultimate, Test Professional, and Team Foundation Server, we achieved our goal of being widely recognized as the industry leader.[1] In 2012, we complemented the Server with the public preview of the hosted Team Foundation Service and started delivering even more value more frequently to software teams.

We've learned a lot from our customers in the past seven years. Visual Studio enables a high-performance Agile software team to release higher-quality software more frequently. It is broadly recognized as the market-leading, innovative solution for software teams, regardless of technology choice. We set out to enable a broad set of scenarios for our customers. We systematically attacked major root causes of waste in the application life cycle, elevated transparency for the broadly engaged team, and focused on flow of value for the end customer. We have eliminated unnecessary silos among roles, to focus on empowering a multidisciplinary, self-managing team. Here are some examples:

No more no repro. One of the greatest sources of waste in software development is a developer's inability to reproduce a reported defect. Traditionally, this is called a "no repro" bug. A tester or user files a bug and later receives a response to the effect of "Cannot reproduce," or "It works on my machine," or "Please provide more information," or something of

the sort. Usually this is the first volley in a long game of Bug Ping-Pong, in which no software gets improved but huge frustration gets vented. Bug Ping-Pong is especially difficult for a geographically distributed team. As detailed in Chapters 1, "The Agile Consensus," and 8, "Testing," VS 2012 shortens or eliminates this no-win game.

No more waiting for build setup. Many development teams have mastered the practice of continuous integration to produce regular builds of their software many times a day, even for highly distributed Web-based systems. Nonetheless, testers regularly wait for days to get a new build to test because of the complexity of getting the build deployed into a production-realistic lab. By virtualizing the test lab and automating the deployment as part of the build, VS 2012 enables testers to take fresh builds daily or intraday with no interruptions. Chapter 7, "Build and Lab," describes how to work with build and lab automation.

No more UI regressions. The most effective user interface (UI) testing is often exploratory, unscripted manual testing. However, when bugs are fixed, it is often hard to tell whether they have actually been fixed or if they simply haven't been found again. VS 2012 removes the ambiguity by capturing the action log of the tester's exploration and allowing it to be converted into an automated test. Now fixes can be retested reliably and automation can focus on the actually observed bugs, not the conjectured ones. Chapter 8 covers both exploratory and automated testing.

No more performance regressions. Most teams know the quickest way to lose a customer is with a slow application or Web site. Yet teams don't know how to quantify performance requirements and accordingly, don't test for load capacity until right before release, when it's too late to fix the bugs that are found. VS 2012 enables teams to begin load testing early. Performance does not need to be quantified in advance because the test can answer the simple question, "What has gotten slower?" And from the end-to-end result, VS profiles the hot paths in the code and points the developer directly to the trouble spots. Chapter 6, "Development," and Chapter 8 cover profiling and load testing.

No more missed changes. Software projects have many moving parts, and the more iterative they are, the more the parts move. It's easy for developers and testers to misunderstand requirements or overlook the impact

of changes. To address this, Visual Studio Test Professional introduces test impact analysis. This capability compares the changes between any two builds and recommends which tests to run, both by looking at the work completed between the builds and by analyzing which tests cover the changed code based on prior coverage. Chapters 3, "Product Ownership," and 4, "Running the Sprint," describe the product backlog and change management, and Chapters 6 through 8 show test impact analysis and the corresponding safety nets from unit testing, build automation, and acceptance testing.

No more planning black box. In the past, teams have often had to guess at their historical velocity and future capacity. VS 2012 draws these directly from the Team Foundation Server database and builds an Excel worksheet that allows the team to see how heavily loaded every individual is in the sprint. The team can then transparently shift work as needed. Examples of Agile planning are discussed in Chapter 2, "Scrum, Agile Practices, and Visual Studio," and Chapter 4.

No more late surprises. Agile teams, working iteratively and incrementally, often use burndown charts to assess their progress. Not only does VS 2012 automate the burndowns, but project dashboards go beyond burndowns to provide a real-time view of quality and progress from many dimensions: requirements, tasks, tests, bugs, code churn, code coverage, build health, and impediments. Chapter 4 introduces the "happy path" of running a project and discusses how to troubleshoot project "smells."

No more legacy fear. Very few software projects are truly "greenfield," developing brand-new software on a new project. More frequently, teams extend or improve existing systems. Unfortunately, the people who worked on earlier versions are often no longer available to explain the assets they have left behind. VS 2012 makes it much easier to work with the existing code by introducing tools for architectural discovery. VS 2012 reveals the patterns in the software and enables you to automatically enforce rules that reduce or eliminate unwanted dependencies. These rules can become part of the check-in policies that ensure the team's definition of *done* to prevent inadvertent architectural drift. Architectural changes can also be tied to bugs or work, to maintain transparency. Chapter 5, "Architecture," covers

the discovery of existing architecture, and Chapter 7 shows you how to automate the definition of *done.*

No more distributed development pain. Distributed development is a necessity for many reasons: geographic distribution, project complexity, release evolution. VS 2012 takes much of the pain out of distributed development processes both proactively and retrospectively. Gated check-in proactively forces a clean build with verification tests before accepting a check-in. Branch visualization retrospectively lets you see where changes have been applied. The changes are visible both as code and work item updates (for example, bug fixes) that describe the changes. You can visually spot where changes have been made and where they still need to be promoted. Chapters 6 and 7 show you how to work with source, branches, and backlogs across distributed teams.

No more technology silos. More and more software projects use multiple technologies. In the past, teams often have had to choose different tools based on their runtime targets. As a consequence, .NET and Java teams have not been able to share data across their silos. Visual Studio Team Foundation Server 2012 integrates the two by offering clients in both the Visual Studio and Eclipse IDEs, for .NET and Java, respectively. This changes the either-or choice into a both-and, so that everyone wins. Again, Chapters 6 and 7 include examples of working with your Java assets alongside .NET.

These scenarios are not an exhaustive list, but a sampling of the motivation for VS 2012. All of these illustrate our simple priorities: reduce waste, increase transparency, and accelerate the flow of value to the end customer. This book is written for software teams considering running a software project using VS 2012. This book is more about the *why* than the *how*.

This book is written for the team as a whole. It presents information in a style that will help all team members get a sense of each other's viewpoint. I've tried to keep the topics engaging to all team members. I'm fond of Einstein's dictum "As simple as possible, but no simpler," and I've tried to write that way. I hope you'll agree and recommend the book to your colleagues (and maybe your boss) when you've finished with it.

Enough about Visual Studio 2012 to Get You Started

When I write about Visual Studio (or VS) I'm referring to the full product line. As shown in Figure P.1, the VS 2012 family is made up of a server and a small selection of client-side tools, all available as VS Ultimate.

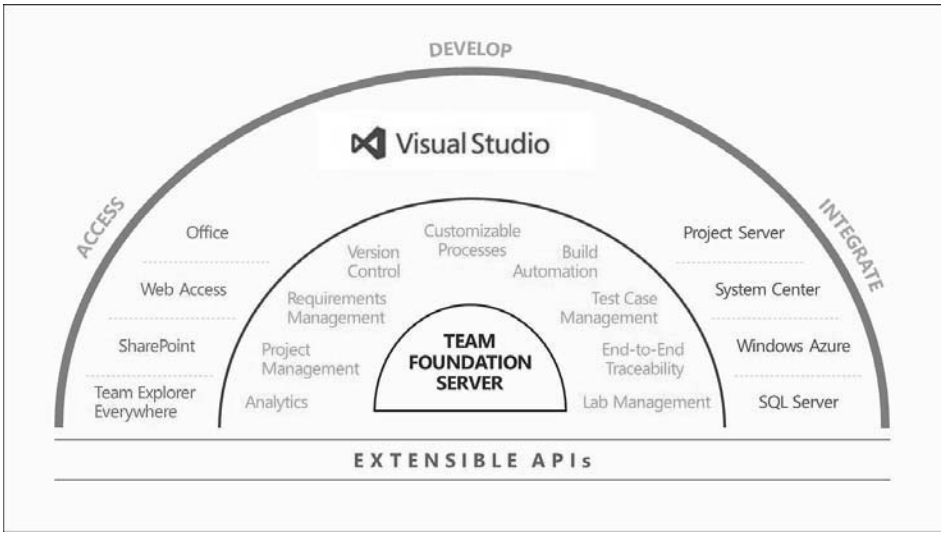


Figure P.1: Team Foundation Server forms the collaboration hub of VS 2012. The client components are available in VS Ultimate.

Team Foundation Server (TFS) is the ALM backbone, providing source control management, build automation, work item tracking, test case management, reporting, and dashboards. Part of TFS is Lab Management, which extends the build automation of TFS to integrate physical and virtual test labs into the development process.

If you just have TFS, you get a client called Team Explorer that launches either stand-alone or as a plug-in to the Visual Studio Professional IDE. Team Explorer Everywhere, a comparable client written in Java, launches as an Eclipse plug-in. You also get Team Web Access and plug-ins that let you connect from Microsoft Excel or Project. SharePoint hosts the dashboards.

Visual Studio Premium adds the scenarios that are described in Chapter 6 around working with the code. Visual Studio Test Professional, although it bears the VS name, is a separate application outside the IDE, designed with the tester in mind. You can see lots of Test Professional examples in Chapter 8. VS Ultimate, which includes Test Professional, adds architectural modeling and discovery, discussed in Chapter 5.

There is also a rich community of partner products that use the extensibility to provide additional client experiences on top of TFS. Figure P.2 shows examples of third-party extensions that enable MindManager, Microsoft Word, and Microsoft Outlook as clients of TFS. You can find a directory at www.visualstudiowidgets.com/.

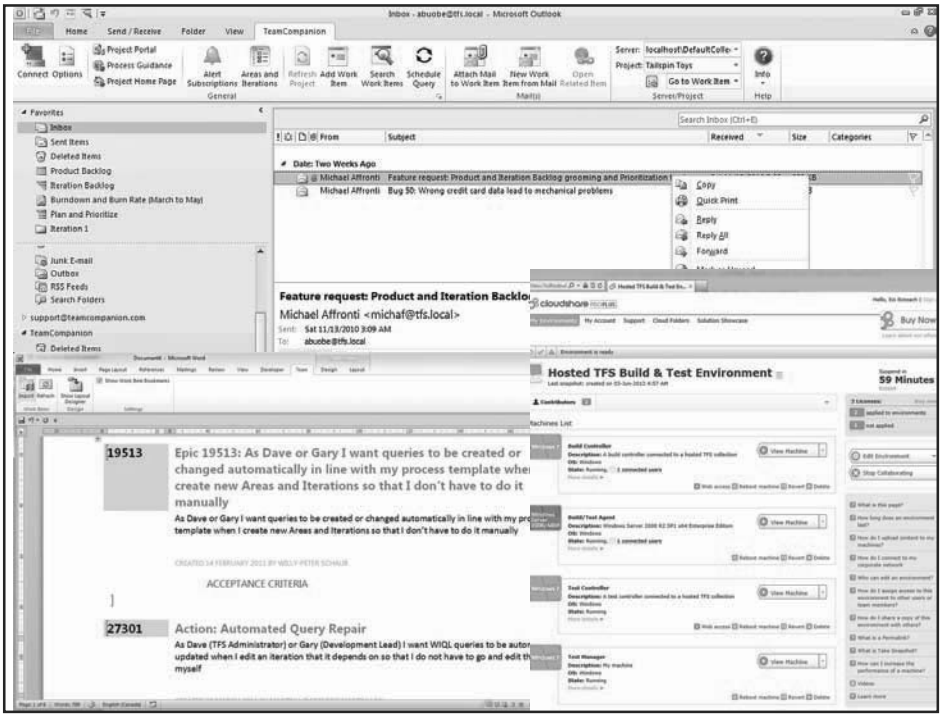


Figure P.2: A broad catalog of partner products extend TFS. Shown here are Ekobit TeamCompanion, CloudShare hosted dev/test labs, and the open source TFS Word Add-in available on CodePlex.

Of course, all the clients read and feed data into TFS, and their trends surface on the dashboards, typically hosted on SharePoint. Using Excel Services or SQL Server Reporting Services, you can customize these dashboards. Dashboard examples are the focus of Chapter 4.

Of course, there's plenty more to learn about VS at the Developer Center of http://msdn. microsoft.com/vstudio/.

A Note to Readers about Editions of the Book

We are pleased to bring you the third edition of the book. If you have been with the book since the first edition, *Software Engineering with Microsoft Visual Studio Team System* (published May 2006), you will notice significant changes and essentially an entire rewrite of the book. If you purchased the second edition, *Agile Software Engineering with Visual Studio: From Concept to Continuous Feedback*, published last year, you will notice little has changed. This third edition has been revised with a focus on Visual Studio 2012 and is recommended if you are working with Visual Studio 2012.

Acknowledgments

Hundreds of colleagues and millions of customers have contributed to shaping Visual Studio. In particular, the roughly two hundred "ALM MVPs" who relentlessly critique our ideas have enormous influence. Regarding this book, there are a number of individuals who must be singled out for the direct impact they made. Ken Schwaber convinced me that this book was necessary. The inexhaustible Brian Harry and Cameron Skinner provided detail and inspiration. Jason Zander gave me space and encouragement to write. Tyler Gibson illustrated the Scrum cycles to unify the chapters. Natalie Wells, Martin Woodward, and Amit Chopra helped us with builds, virtual machines, and prerelease logistics to get the work done in time. Among our reviewers, David Starr, Claude Remillard, Aaron Bjork, David Chappell, and Adam Cogan stand out for their thorough and careful comments. And a special thanks goes to Joan Murray, our editor at Pearson, whose patience was limitless.

About the Authors

Sam Guckenheimer

When I wrote the predecessor of this book, I had been at Microsoft less than three years. I described my history like this:

I joined Microsoft in 2003 to work on Visual Studio Team System (VSTS), the new product line that was just released at the end of 2005. As the group product planner, I have played chief customer advocate, a role that I have loved. I have been in the IT industry for twenty-some years, spending most of my career as a tester, project manager, analyst, and developer.

As a tester, I've always understood the theoretical value of advanced developer practices, such as unit testing, code coverage, static analysis, and memory and performance profiling. At the same time, I never understood how anyone had the patience to learn the obscure tools that you needed to follow the right practices.

As a project manager, I was always troubled that the only decent data we could get was about bugs. Driving a project from bug data alone is like driving a car with your eyes closed and only turning the wheel when you hit something. You really want to see the right indicators that you are on course, not just feel the bumps when you stray off it. Here, too, I always understood the value of metrics, such as code coverage and project velocity, but I never understood how anyone could realistically collect all that stuff.

As an analyst, I fell in love with modeling. I think visually, and I found graphical models compelling ways to document and communicate. But the models always got out of date as soon as it came time to implement anything. And the models just didn't handle the key concerns of developers, testers, and operations.

In all these cases, I was frustrated by how hard it was to connect the dots for the whole team. I loved the idea in Scrum (one of the Agile processes) of a "single product backlog"—one place where you could see all the work—but the tools people could actually use would fragment the work every which way. What do these requirements have to do with those tasks, and the model elements here, and the tests over there? And where's the source code in that mix?

From a historical perspective, I think IT turned the corner when it stopped trying to automate manual processes and instead asked the question, "With automation, how can we reengineer our core business processes?" That's when IT started to deliver real business value.

They say the cobbler's children go shoeless. That's true for IT, too. While we've been busy automating other business processes, we've largely neglected our own. Nearly all tools targeted for IT professionals and teams seem to still be automating the old manual processes. Those processes required high overhead before automation, and with automation, they still have high overhead. How many times have you gone to a 1-hour project meeting where the first 90 minutes were an argument about whose numbers were right?

Now, with Visual Studio, we are seriously asking, "With automation, how can we reengineer our core IT processes? How can we remove the overhead from following good process? How can we make all these different roles individually more productive while integrating them as a high-performance team?"

Obviously, that's all still true.

Neno Loje

I started my career as a software developer—first as a hobby, later as profession. At the beginning of high school, I fell in love with writing software because it enabled me to create something useful by transforming an idea into something of actual value for someone else. Later, I learned that this was generating customer value.

However, the impact and value were limited by the fact that I was just a single developer working in a small company, so I decided to focus on helping and teaching other developers. I started by delivering pure technical training, but the topics soon expanded to include process and people,

because I realized that just introducing a new tool or a technology by itself does not necessarily make teams more successful.

During the past six years as an independent ALM consultant and TFS specialist, I have helped many companies set up a team environment and software development process with VS. It has been fascinating to watch how removing unnecessary, manual activities makes developers and entire projects more productive. Every team is different and has its own problems. I've been surprised to see how many ways exist (both in process and tools) to achieve the same goal: Deliver customer value faster though great software.

When teams look back at how they worked before, without VS, they often ask themselves how they could have survived without the tools they use now. However, what had changed from the past were not only the tools, but also the way they work as a team.

Application Lifecycle Management and practices from the Agile Consensus help your team to focus on the important things. VS and TFS are a pragmatic approach to implement ALM (even for small, nondistributed teams). If you're still not convinced, I urge you to try it out and judge for yourself.

Endnotes

[1] See, for example: Thomas E. Murphy and Jim Duggan, "Magic Quadrant for Application Life Cycle Management," 5 June 2012 ID:G00218016, available at http://www.gartner.com/technology/reprints.do?id=1-1ASCXON&ct=120606&st=sb.

2

Scrum, Agile Practices, and Visual Studio

One methodology cannot possibly be the "right" one, but… there is an appropriate, different way of working for each project and project team.[1]

—Alistair Cockburn

Figure 2.1: The rhythm of a crew rowing in unison is a perfect example of flow in both the human and management senses. Individuals experience the elation of performing optimally, and the coordinated teamwork enables the system as a whole (here, the boat) to achieve its optimum performance. It's the ideal feeling of a "sprint."

THE PRECEDING CHAPTER DISCUSSED the Agile Consensus of the past decade. That chapter distinguished between complicated projects, with well-controlled business or technical risk, and complex ones, where the technology and business risks are greater. Most new software projects are complex; otherwise, the software would not be worth building.

This chapter covers the next level of detail—the characteristics of software engineering and management practices, the "situationally specific" contexts to consider, and the examples that you can apply in Visual Studio (VS). In this chapter, you learn about the mechanisms that VS (primarily Team Foundation Server [TFS]) provides to support the team enacting the process. Whereas Chapter 1, "The Agile Consensus," gave an outside-in view of what a team needs, this chapter provides an inside-out overview of the tooling that makes the enactment possible.

Visual Studio and Process Enactment

Through three classes of mechanisms, VS helps the team follow a defined software process:

1. As illustrated in Chapter 1, TFS captures backlogs, workflow, status, and metrics. Together, these keep the work transparent and guide the users to the next appropriate actions. TFS also helps ensure the "doneness" of work so that the team cannot accrue technical debt without warning and visibility.
2. Each team project tracked by TFS starts off with a process template that defines the standard workflows, reports, roles, and artifacts for the process. These are often changed later during the course of the team project as the team learns and tunes its process, but their initial defaults are set according to the chosen process template.
3. On the IDE clients (VS or Eclipse), there are user experiences that interact with the server to ensure that the policies are followed and that any warnings from policy violations are obvious.

Process Templates

The process template supports the workflow of the team by setting the default work item types, reports, queries, roles (i.e., security groups), team portal, and artifacts. Work item types are the most visible of these because they determine the database schema that team members use to manage the backlog, select work, and record status as it is done. When a team member creates a team project, the Project Creation Wizard asks for a choice of process template, as shown in Figure 2.2.

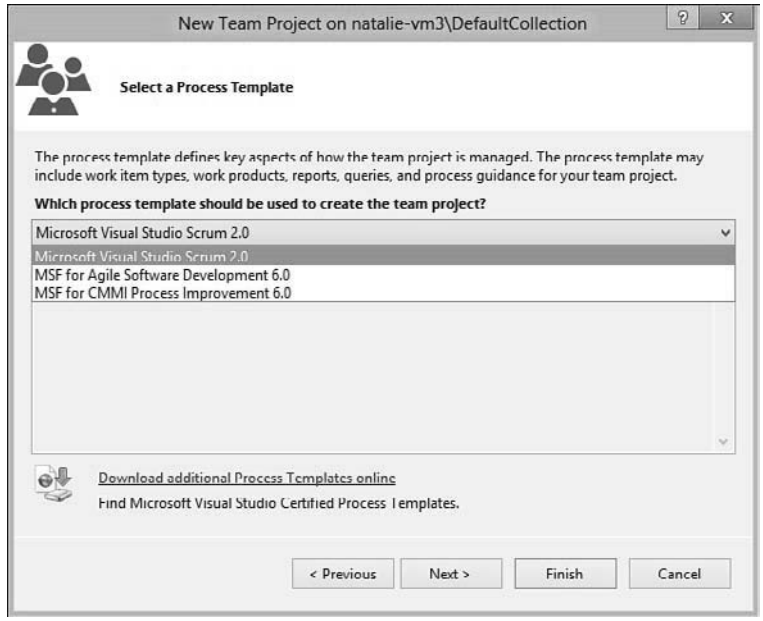


Figure 2.2: The Project Creation Wizard lets you create a team project based on any of the currently installed process templates.

Microsoft provides three process templates as standard:

1. **Microsoft Visual Studio Scrum:** This process template directly supports Scrum, and was developed in collaboration with Ken Schwaber based on the *Scrum Guide*.[2] The Scrum process template defines work item types for Product Backlog Item, Bug, Task,

Impediment, Test Case, Shared (Test) Steps, Code Review Request/Response, and Feedback Request/Response. The reports are Backlog Overview, Release Burndown, Sprint Burndown, and Velocity.

2. **MSF for Agile Software Development:** MSF Agile is also built around an agile base but incorporates a broader set of artifacts than the Scrum process template. In MSF Agile, product backlog items (PBIs) are called *user stories* and impediments are called *issues.* The report shown in Figure 1.4 in Chapter 1 is taken from MSF Agile.
3. **MSF for CMMI Process Improvement:** This process template is also designed for iterative work practices, but with more formality than the other templates. This one is designed to facilitate a team's practice of Capability Maturity Model Integration (CMMI) Level 3 as defined by the Software Engineering Institute.[3] Accordingly, it extends MSF Agile with more formal planning, more documentation and work products, more sign-off gates, and more time tracking. Notably, this process template adds Change Request and Risk work item types and uses a Requirement work item type that is more elaborate than the user stories of MSF Agile.

Other companies provide their own process templates and can have these certified by Microsoft. For example, Sogeti has released a version of its Test Management Approach (TMap) methodology as a certified process template, downloadable from the Visual Studio Developer Center at http://msdn.microsoft.com/vstudio/aa718795.aspx.

When you create a team project with TFS, you choose the process template to apply, as shown in Figure 2.2.

Teams

Processes tend to prescribe team structure. Scrum, for example, has three roles. The Product Owner is responsible for the external definition of the product, captured in the product backlog, and the management of the stakeholders and customers. The Team of Developers is responsible for the implementation. And the Scrum Master is responsible for ensuring that the Scrum process is followed.

In Scrum, the team has three to nine dedicated members. Lots of evidence indicates that this is the size that works best for close communication. Often, one of the developers doubles as the Scrum Master. If work is larger than can be handled by one team, it should be split across multiple teams, and the Scrum Masters can coordinate in a scrum of scrums. A Product Owner can serve across multiple scrum teams but should not double as a Scrum Master.

In TFS, each team has a home page with data from the current sprint of its project, like an up-to-date burndown chart and the remaining, incomplete PBIs, as shown in Figure 2.3. Additionally, the team gets its own product backlog and a task board—both are available using the Web browser. To support larger projects with multiple teams, TFS enables the concept of master backlogs that consolidate each team's product backlog into a single view.[4]

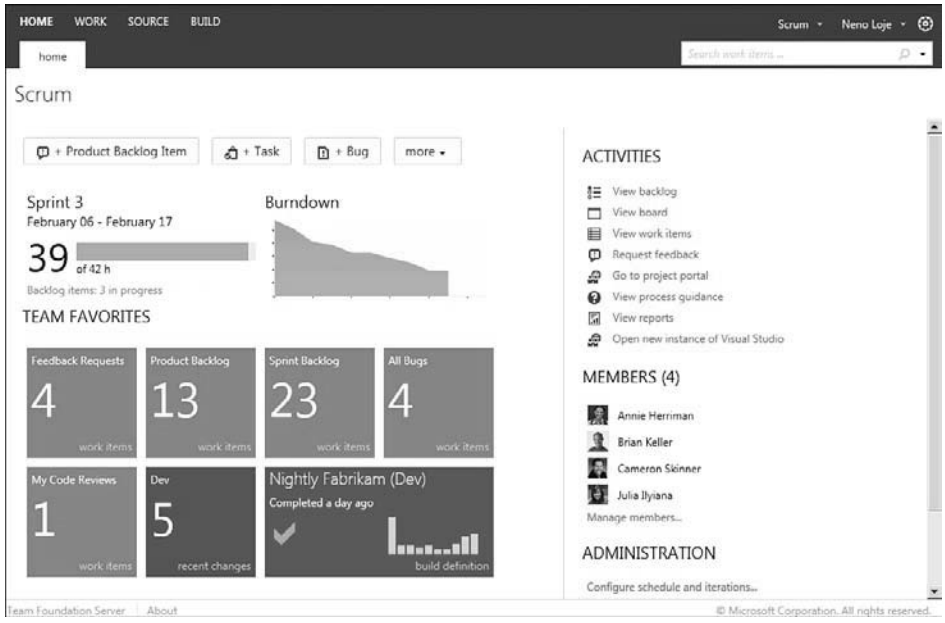


Figure 2.3: The tiles on the team's home page represent the team's current progress as well as favorite metrics, including work items, testing, source control, and the automated build and test results. Additionally, you can jump to all important views from this page.

In most cases, it is bad Scrum to use tooling to enforce permissions rather than to rely on the team to manage itself. Instead, it is generally better to assume trust, following the principle that "responsibility cannot be assigned; it can only be accepted."[5] TFS always captures the history of every work item change, thereby making it easy to trace any unexpected changes and reverse any errors.

Nonetheless, sometimes permissions are important (perhaps because of regulatory or contractual circumstances, for example). Accordingly, you can enforce permissions in a team project in four ways:

1. By role
2. By work item type down to the field and value
3. By component of the system (through the area path hierarchy of work items and the folder and branch hierarchy of source control)
4. By builds, reports, and team site

For example, you can set a rule on the PBI work item type that only a Product Owner can update PBIs. In practice, this is rarely done.

Process Cycles and TFS

A core concept of the convergent evolution discussed in Chapter 1 is iterative and incremental development. Scrum stresses the basis of iteration in empirical process control because through rapid iteration the team reduces uncertainty, learns by doing, inspects and adapts based on its progress, and improves as it goes.[6] Accordingly, Scrum provides the most common representation of the main macro cycles in a software project: the *release* and the *sprint* (a synonym for *iteration*), as shown in Figure 2.4. Scrum provides some simple rules for managing these.

Release

The release is the path from vision to delivered software. As Ken Schwaber and Jeff Sutherland explain in the *Scrum Guide:*

Release planning answers the questions, "How can we turn the vision into a winning product in the best possible way? How can we meet or exceed the desired customer satisfaction and Return on Investment?" The release plan establishes the goal of the release, the highest priority Product Backlog, the major risks, and the overall features and functionality that the release will contain. It also establishes a probable delivery date and cost that should hold if nothing changes.[7]

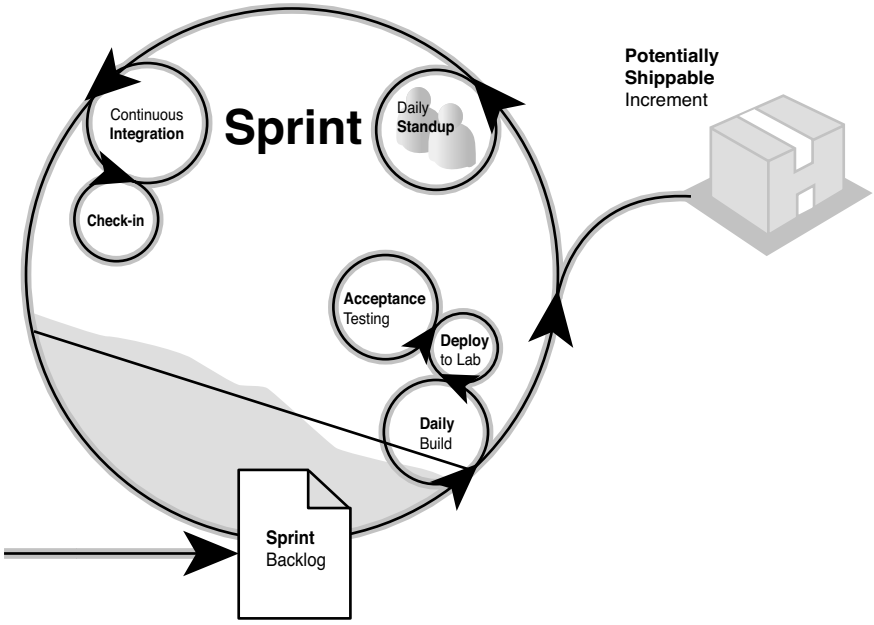


FIGURE 2.4: Software projects proceed on many interlocking cycles, ranging from the "code-edit-test-debug-check in" cycle, measured in minutes, to continuous integration, to daily testing cycles, to the sprint. These are views of both the process and the flow of data, automated by the process tooling.

The release definition is contained in the *product backlog*, which consists of requirements, unsurprisingly named *product backlog items*, as shown in Figure 2.5. Throughout the release, the Product Owner keeps the PBIs stack ranked to remove ambiguity about what to do next. As DeMarco and Lister have put it:

Rank-ordering for all functions and features is the cure for two ugly project maladies: The first is the assumption that all parts of the product are equally important. This fiction is preserved on many projects because it assures that no one has to confront the stakeholders who have added their favorite bells and whistles as a price for their cooperation. The same fiction facilitates the second malady, piling on, in which features are added with the intention of overloading the project and making it fail, a favorite tactic of those who oppose the project in the first place but find it convenient to present themselves as enthusiastic project champions rather than as project adversaries.[8]

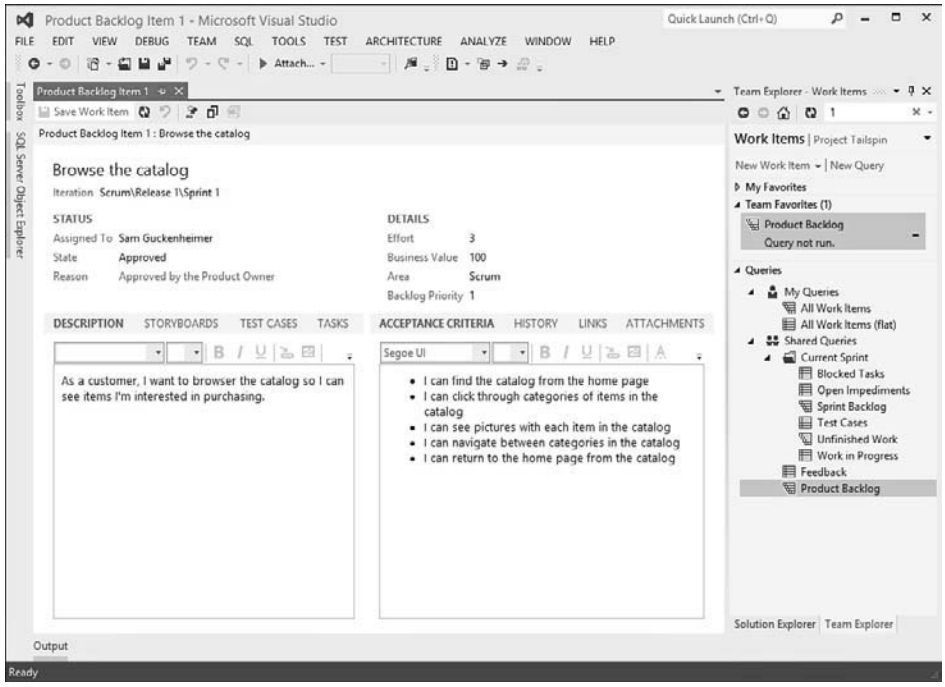


FIGURE 2.5: A product backlog item, shown here as accessed inside the VS IDE, can also be viewed from the Web Portal, Microsoft Excel, Microsoft Project, and many third-party plug-in tools available for TFS.

A common and useful practice is stating the PBIs, especially the functional requirements, as *user stories.* User stories take the form *As a <target*

customer persona>, *I can <achieve result> in order to <realize value>*. Chapter 3, "Product Ownership," goes into more detail about user stories and other forms of requirements.

Sprint

In a Scrum project, every sprint has the same duration, typically two to four weeks. Prior to the sprint, the team helps the Product Owner groom the product backlog, estimating a rough order of magnitude for the top PBIs. This estimation has to include all costs associated with completing the PBI according to the team's agreed-upon definition of *done.* The rough estimation method most widely favored these days is *Planning Poker,* adapted by Mike Cohn as a simple, fast application of what had been described by Barry Boehm as the Wideband Delphi Method.[9] Planning Poker is easy and fast, making it possible with minimal effort to provide estimates that are generally as good as those derived from much longer analysis. Estimates from Planning Poker get entered as story points in the PBI work item. Planning Poker is discussed further in Chapter 4, "Running the Sprint."

Another great practice is to define at least one acceptance test for each PBI. These are captured in TFS as test cases, a standard work item type. Defining acceptance tests early has three benefits:

1. They clarify the intent of the PBI.
2. They provide a *done* criterion for PBI completion.
3. They help inform the estimate of PBI size.

At the beginning of the sprint, the team commits to delivering a *potentially shippable increment* of software realizing some of the top-ranked product backlogs. The commitment factors the cumulative estimate of the PBIs, the team's capacity, and the need to deliver customer value in the potentially shippable increment. Then, only the PBIs committed for the current sprint are broken down by the team into tasks. These tasks are collectively called the *sprint backlog* (see Figure 2.6).

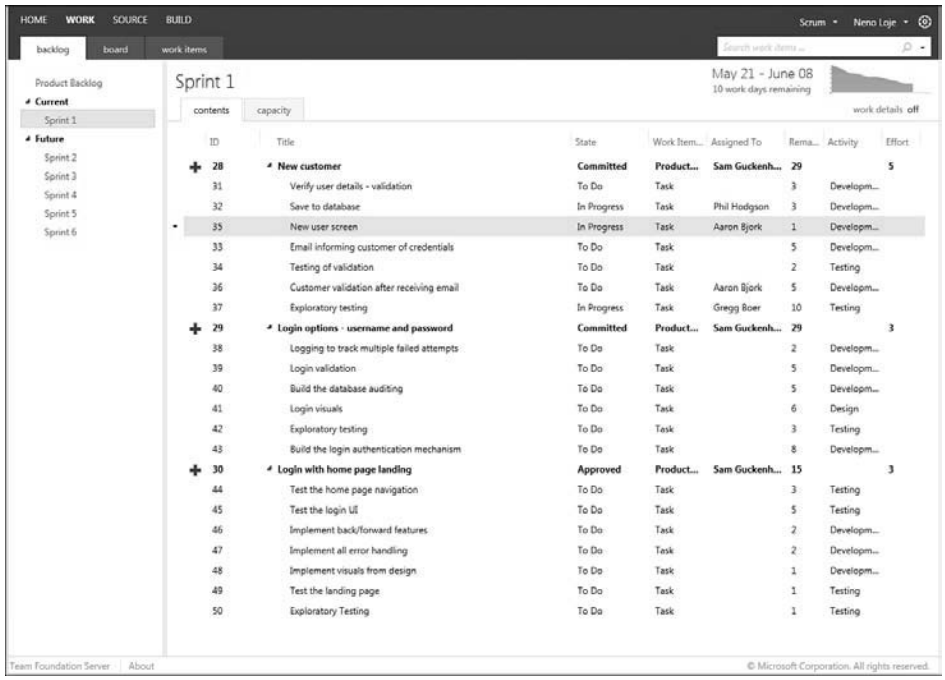


Figure 2.6: The sprint backlog, shown here as accessed from the Web Portal, consists of the tasks for the current sprint, grouped under the PBIs to which the team has committed.

Don't Confuse Product Backlog and Sprint Backlog

In our experience, the most common confusion around Scrum terminology is the use of the word *backlog* in two different instances. To some extent, the confusion is a holdover from earlier project management techniques. The product backlog holds only requirements and bugs deferred to future sprints and is the interface between the Product Owner, representing customers and other stakeholders, and the team. PBIs are assessed in story points only.

The *sprint backlog* consists of implementation tasks, test cases, bugs of the current sprint, and impediments and is for the implementation team. When working on a task, a team member updates the remaining hours on these tasks, but typically does not touch the PBI, except to mark it as ready for test or completed. Stakeholders should not be concerned with the sprint backlog, only with the PBIs.

Using tasks, the team lays out an initial plan for how to transform the selected PBIs into working software. Estimating each task's remaining hours helps the team verify the effort is not exceeding their capacity, as shown in Figure 2.7.

Figure 2.7: For each sprint the team sets its capacity. Each individual can identify primary activity and identify planned days off. The work pane compares available time against the estimated hours for the team, both for each team member and grouped at the level of activities like development or testing.

Handling Bugs

Bugs should be managed according to context. Different teams view bugs differently. Product teams tend to think of anything that detracts from customer value as a bug, whereas contractors stick to a much tighter definition.

In either case, do not consider a PBI done if there are outstanding bugs because doing so would create technical debt. Accordingly, treat bugs that

are found in PBIs of the current sprint as simply undone work and manage them in the current iteration backlog.

In addition, you often discover bugs unrelated to the current PBIs, and these can be added to the product backlog, unless you have spare capacity in the current sprint. (The committed work of the sprint should normally take precedence, unless the bug found is an impediment to achieving the sprint goal.) This can create a small nuisance for grooming the product backlog, in that individual bugs are usually too fine-grained and numerous to be stack ranked against the heftier PBIs. In such a case, create a PBI as a container or allotment for a selection of the bugs, make it a "parent" of them in TFS, and rank the container PBI against its peers (see Figure 2.8).

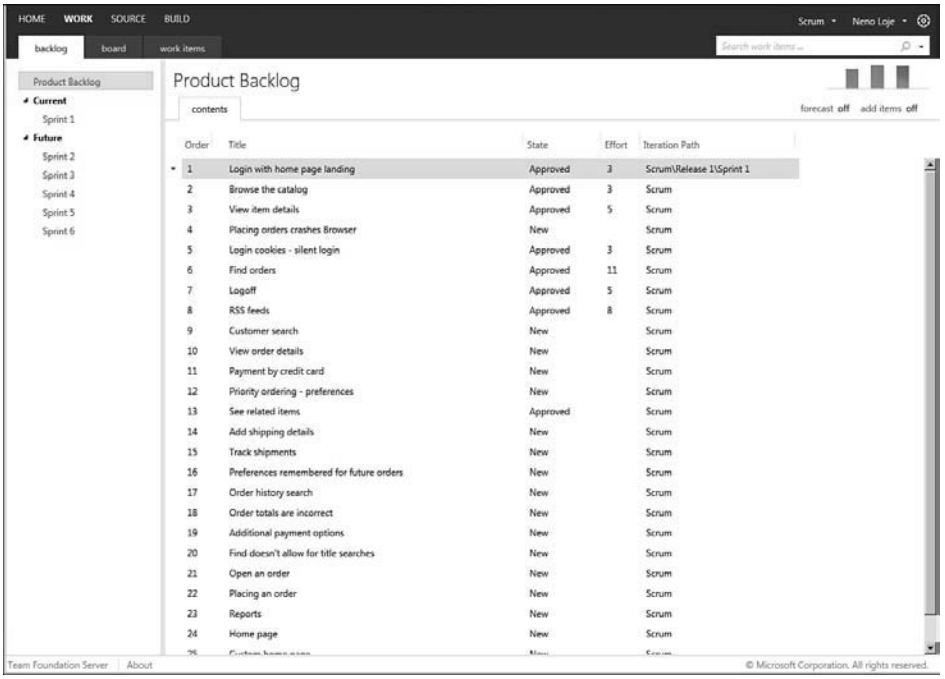


Figure 2.8: The product backlog contains the PBIs that express requirements and the bugs that are not handled in the current sprint. This can be accessed from any of the TFS clients; here it is shown in the VS IDE.

Avoiding Analysis Paralysis

A great discipline of Scrum is the strict timeboxing of the sprint planning meeting, used for commitment of the product backlog (the "what") and for initial task breakdown of the sprint backlog (the "how"). For a one-month sprint, the sprint planning meeting is limited to a day before work begins on the sprint. For shorter sprints, the meeting should take a proportionally shorter length of time.

Note that this does not mean that all tasks are known on the first day of the sprint. On the contrary, tasks may be added to the sprint backlog whenever necessary. Rather, timeboxing sprint planning means that the team needs to understand the committed PBIs *well enough to start* work. In this way, only 5% of the sprint time is consumed by planning before work begins. (Another 5% of the calendar, the last day of a monthly sprint, is devoted to review and retrospective.) In this way, 90% of the sprint is devoted to working through the sprint backlog.

Bottom-Up Cycles

In addition to the two macro cycles of release and sprint, TFS uses the two finer-grained cycles of check-in and test to collect data and trigger automation. In this way, with no overhead for the users, TFS can provide mechanisms to support both automating definitions of *done* and transparently collecting project metrics.

Personal Development Preparation

As discussed in Chapter 6, "Development," VS provides continuous feedback to the developer to practice test-driven development; correct syntax suggestions with IntelliSense; and check for errors with local builds, tests, and check-in policy reviews. These are private activities, in the sense that VS makes no attempt to persist any data from these activities before the developer decides to check in.

Check-In Cycle

The finest-grained coding cycle at which TFS collects data and applies workflow is the check-in (that is, any delivery of code by the developer from

a private workspace to a shared branch). This cycle provides the first opportunity to measure *done* on working code. The most common Agile practice for the check-in cycle is continuous integration, in which every check-in triggers a team build from a TFS build definition. The team build gets the latest versions of all checked-in source from all contributors, provisions a build server, and runs the defined build workflow, including any code analysis, lab deployment, or build verification tests that have been defined in the build. (See Chapter 7, "Build and Lab," for more information.)

Continuous integration is a great practice, if build breaks are rare. In that case, it is a great way to keep a clean, running drop of the code at all times. The larger the project, however, the more frequent build breaks can become. For example, imagine a source base with 100 contributors. Suppose that they are all extraordinary developers, who make an average of only one build break per three months. With continuous integration, their build would be broken every day.

To avoid the frequent broken builds, TFS offers a form of continuous integration called *gated check-in*. Gated check-in extends the continuous integration workflow, in that it provisions a server and runs the team build *before* check-in. Only if the full build passes, *then* the server accepts the code as checked in. Otherwise, the check-in is returned to the developer as a shelveset with a warning detailing the errors. Chapter 9, "Lessons Learned at Microsoft Developer Division," describes how we use this at Microsoft.

In addition, prior to the server mechanisms of continuous integration or gated check-in, TFS runs *check-in policies* on the clients. These are the earliest and fastest automated warnings for the developer. They can validate whether unit tests and code analysis have been run locally, work items associated, check-in notes completed, and other "doneness" criteria met before the code goes to the server for either continuous integration or gated check-in.

Test Cycle

Completed PBIs need to be tested, as do bug fixes. Typically, team members check in code in small increments many times before completing a PBI. However, when a PBI is completed, a test cycle may start. In addition, many PBIs and bug fixes are often completed in rapid succession, and these can

be combined into a single test cycle. Accordingly, a simple way to handle test cycles is to make them daily.

TFS allows for multiple team build definitions, and a good practice is to have a daily build in addition to the continuous integration or gated check-in build. When you do this, every daily "build details" page shows the increment in functionality delivered since the previous daily build, as shown in Figure 2.9.

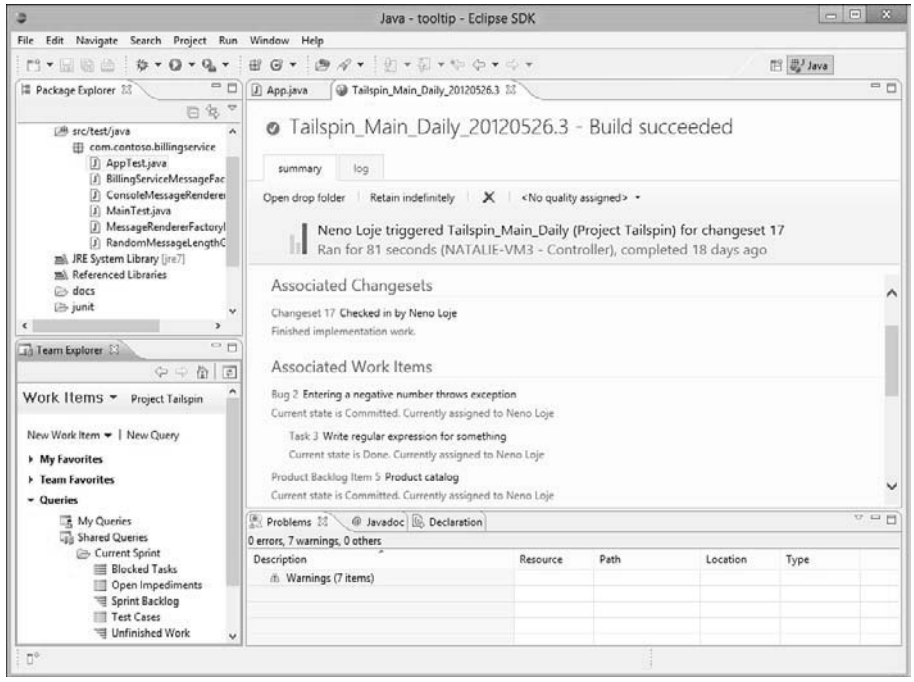


FIGURE 2.9: Every build has a "build details" page that serves as an automated release note, accessible from the dashboard or inside the IDE clients. In this case, it is shown inside Eclipse, as a team working with Java code would see.

In addition, Microsoft Test Manager (MTM, part of the VS product line) enables you to compare the current build against the last one tested to see the most important tests to run based on both backlog changes and new or churned code, as shown in Figure 2.10. (See Chapter 8, "Test," for more information.)

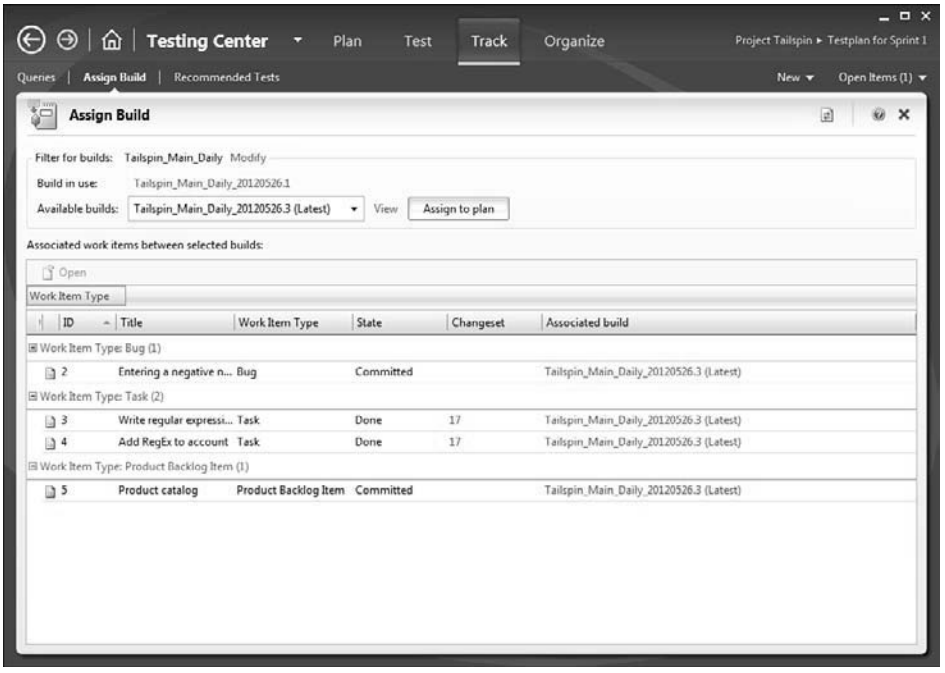


FIGURE 2.10: This build assignment in Microsoft Test Manager is a great way to start the test cycle because it shows the new work delivered since the last tested build and can recommend tests accordingly.

Daily Cycle

The Scrum process specifies a *daily scrum*, often called a "daily stand-up meeting," to inspect progress and adapt to the situation. Daily scrums should last no more than 15 minutes. As the *Scrum Guide* explains, during the meeting, each team member explains the following:

1. What has the team member accomplished since the last meeting?
2. What will the team member accomplish before the next meeting?
3. What obstacles currently impede the team member?

Daily scrums improve communications, eliminate other meetings, identify and remove impediments to development, highlight and promote quick decision making, and improve everyone's level of project knowledge.

Although TFS does not require daily builds, and the process rules do not mandate combining the daily and testing cycles, treating the daily cycle and test cycle as the same is certainly convenient. TFS helps considerably with preparation for the Scrum questions:

- As Figures 2.9 and 2.10 show, the automated release note of the "build details" page and the test recommendations of MTM help resolve any discrepancies in assumptions for question 1.
- The task board, shown in Figure 2.11, should align with question 2.
- The Open Impediments or Open Issues query, shown in Figure 2.12, should match question 3.

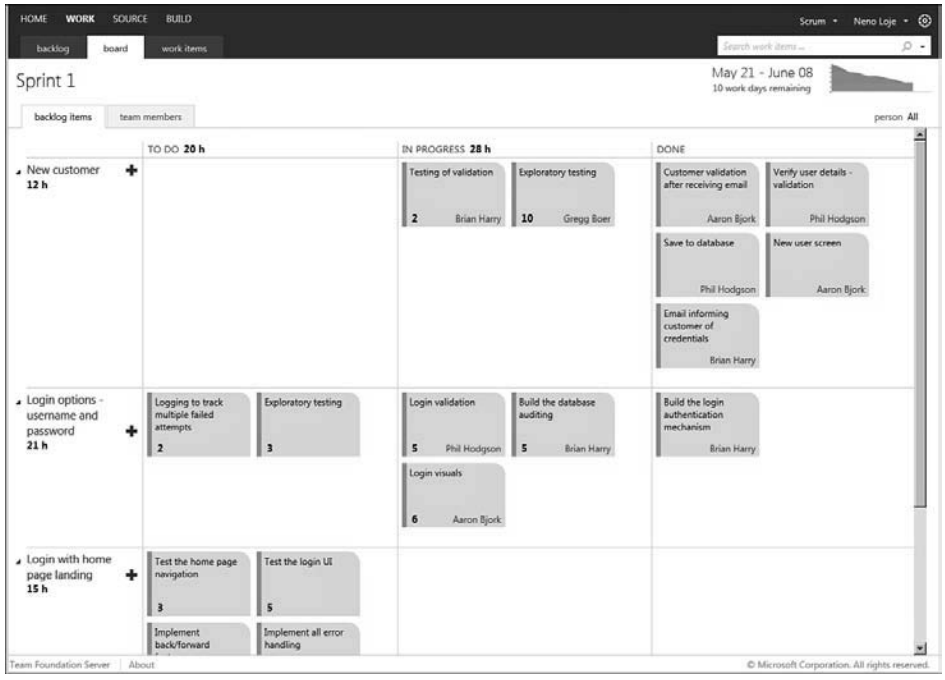


Figure 2.11: The TFS task board shows the progress of the sprint backlog visually grouped by PBI. Alternative queries allow different groupings, such as by team member.

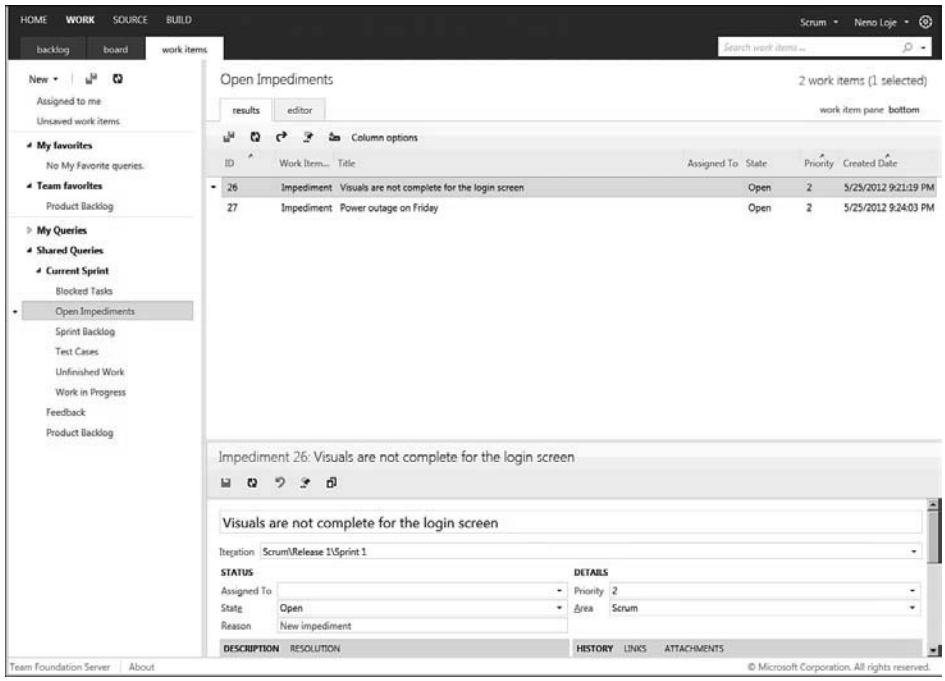


Figure 2.12: The Open Impediments query shows the current state of blocking issues as of the daily scrum.

These tools don't replace the daily scrum, but they remove any dispute about the data of record. In this way, the team members can focus the meeting on crucial interpersonal communication rather than on questions about whose data to trust.

Definition of *Done* at Every Cycle

For each of these cycles—check-in, test, release, and sprint—the team should have a common definition of *done* and treat it as a social contract. The entire team should be able to see the status of *done* transparently at all times. Without this social contract, it is impossible to assess technical debt and, accordingly, impossible to ship increments of software predictably.

With Scrum and TFS working together, every cycle has a done mechanism. Check-in has its policies and the build workflows, test has the test plans for the cycle, and sprint and release have work items to capture their done lists.

Inspect and Adapt

In addition to the daily 15 minutes, Scrum prescribes that the team have two meetings at the end of the sprint to inspect progress (the sprint review) and identify opportunities for process improvement (the sprint retrospective). Together, these should take about 5% of the sprint, or one day for a monthly sprint. Alistair Cockburn has described the goal of the retrospective well: "Can we deliver more easily or better?"[10] Retrospectives force the team to reflect on opportunities for improvement while the experience is fresh.

Based on the retrospective, the sprint end is a good boundary at which to make process changes. You can tune based on experience, and you can adjust for context. For example, you might increase the check-in requirements for code review as your project approaches production and use TFS check-in policies, check-in notes, and build workflow to enforce these requirements.

Task Boards

Scrum uses the sprint cadence as a common cycle to coordinate prioritization of the product backlog and implementation of the iteration backlog. The team manages its capacity by determining how much product backlog to take into the coming sprint, usually based on the story points delivered in prior sprints. This is an effective model for running an empirical process in complex contexts, as defined in Figure 1.3 in Chapter 1.

TFS includes an automated task board that visualizes the sprint backlog, as shown in Figure 2.11. It provides a graphical way to interact with TFS work items and an instant visual indicator of sprint status.

Automated task boards are especially useful for geographically distributed teams and scrums. You can hang large touch screens in meeting areas at multiple sites, and other participants can see the same images on their laptops. Because they all connect to the same TFS database, they are all current and visible. At Microsoft, we use these to coordinate Scrum teams across Redmond, Raleigh, Hyderabad, and many smaller sites.

Kanban

The history of task boards is an interesting study in idea diffusion. For Agile teams, they were modeled after the so-called *Kanban* (Japanese for "signboard") that Taiichi Ohno of Toyota had pioneered for just-in-time manufacturing. Ohno created his own model after observing how American supermarkets stocked their shelves in the 1950s.[11] Ohno observed that supermarket shelves were stocked not by store employees, but by distributors, and that the card at the back of the cans of soup, for example, was the signal to put more soup on the shelf. Ohno introduced this to the factory, where the card became the signal for the component supplier to bring a new bin of parts.

Surprisingly, only in the past few years have software teams discovered the value of the visual and tactile metaphor of the task board. And Toyota only recently looked to bring Agile methods into its software practices, based not on its manufacturing but on its observation again of Western work practices.[12] So, we've seen an idea move from American supermarkets to Japanese factories to American software teams back to Japanese software teams, over a period of 50 years.

In software practices, Kanban has become the name of more than the task board; it is also the name of an alternative process, most closely associated with David J. Anderson, who has been its primary proponent.[13] Where Scrum uses the team's commitments for the sprint to regulate capacity, Kanban uses work-in-progress (WIP) limits. Kanban models workflow more deterministically with finer state transitions on PBIs, such as Analysis Ready, Dev Ready, Test Ready, Release Ready, and so on. The PBIs in each such state are treated as a queue, and each queue is governed by a WIP limit. When a queue is above the WIP limit, no more work may be pulled from earlier states, and when it falls below, new work is pulled.

Kanban is more prescriptive than Scrum in managing queues. The Kanban control mechanism allows for continuous adjustment, in contrast to Scrum, which relies on the team commitment, reviewed at sprint boundaries. Kanban can readily be used inside the sprint boundaries to keep WIP low.

Fit the Process to the Project

Based on your project context and your retrospectives, you may choose to customize your process template. Ideally, this is a team decision, but certain stakeholders may have special influence. Even then, every team member should understand the rationale of the choice and the value of any practice that the process prescribes. If the value cannot be identified, it is unlikely that it can be realized. Sometimes the purpose might not be intuitive (certain legal requirements for example), but if understood can still be achieved.

As Barry Boehm and Richard Turner have described, it is best to start small:

> Build Your Method Up, Don't Tailor It Down
>
> Plan-driven methods have had a tradition of developing all-inclusive methods that can be tailored down to fit a particular situation. Experts can do this, but nonexperts tend to play it safe and use the whole thing, often at considerable unnecessary expense. Agilists offer a better approach of starting with relatively minimal sets of practices and only adding extras where they can be clearly justified by cost-benefit.[14]

Fortunately, TFS assumes that a team will "stretch the process to fit"—that is, take a small core of values and practices and add more as necessary (see Figure 2.13).

One of the tenets of the Agile Consensus is to keep overhead to a minimum. Extra process is waste unless it has a clear purpose whose return justifies the cost. Three common factors might lead to more steps or done criteria in the process than others: geographic distribution; required documentation; and governance, risk management, and compliance.

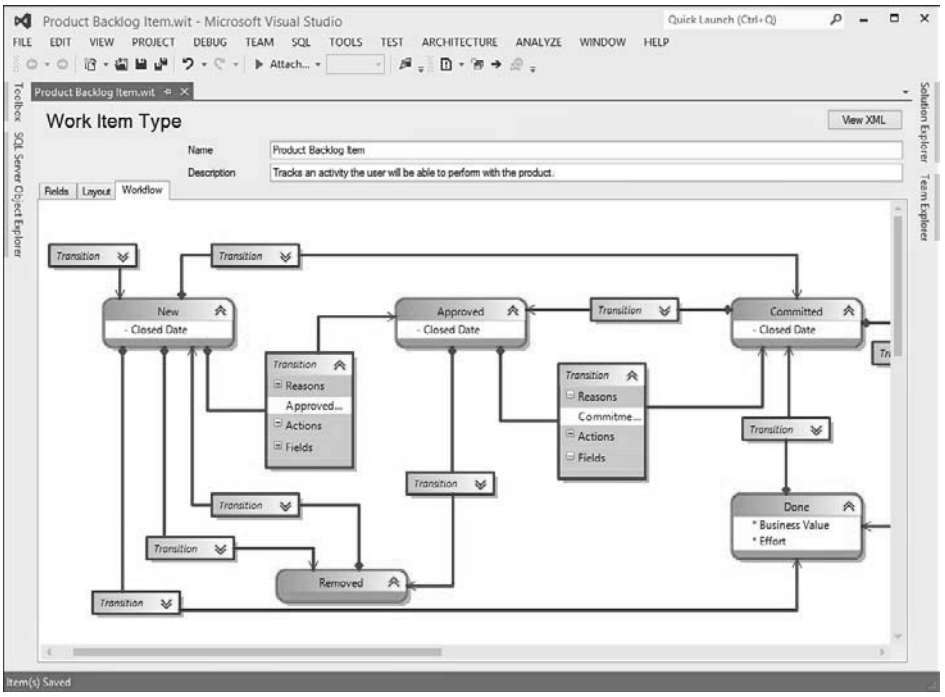


Figure 2.13: The Process Template Editor (in the TFS Power Tools on the VS Gallery) enables you to customize work item types, form design, and workflows.

Geographic Distribution

Most organizations are now geographically distributed. Individual Scrum teams of seven are best collocated, but multiple Scrum teams across multiple locations often need to coordinate work. For example, on VS, we are running scrums of scrums and coordinating sprint reviews and planning across Redmond, Raleigh, and Hyderabad, and several smaller sites, a spread of 12 time zones. In addition to TFS with large screens, we use Microsoft Lync for the video and screen sharing, and we record meetings and sprint review demos so that not everyone needs to be awake at weird hours to see others' work.

Tacit Knowledge or Required Documentation

When you have a geographically distributed team, it is harder to have spontaneous conversations than when you're all in one place, although instant messaging and video chat help a lot. When you're spread out, you

cannot rely just on tacit knowledge. You can also use internal documentation to record contract, consensus, architecture, maintainability, or approval for future audit. Whatever the purpose, write the documentation for its audience and to its purpose and then *stop* writing. Once the documentation serves its purpose, more effort on it is waste. Wherever possible, use TFS work items as the official record so that there is a "single source of truth." Third-party products such as Ekobit TeamCompanion, shown in Chapter 4, can help by converting email into TFS work items for a visible and auditable record.

Governance, Risk Management, and Compliance

Governance, risk management, and *compliance* (GRC) are closely related terms that are usually considered together since the passage of the Sarbanes-Oxley Act of 2002 (SOX) in the United States. For public and otherwise regulated companies, GRC policies specify how management maintains its accountability for IT. GRC policies may require more formality in documentation or in the fields and states of TFS work items than a team would otherwise capture.

One Project at a Time Versus Many Projects at Once

One of the most valuable planning actions is to ensure that your team members can focus on the project at hand without other commitments that drain their time and attention. Gerald Weinberg once proposed a rule of thumb to compute the waste caused by project switching, shown in Table 2.1.[15]

Table 2.1: Waste Caused by Project Switching

Number of Simultaneous Projects	Percent of Working Time Available per Project	Loss to Context Switching
1	100%	0%
2	40%	20%
3	20%	40%
4	10%	60%
5	5%	75%

That was 20 years ago, without suitable tooling. In many organizations today, it is a fact of life that individuals have to work on multiple projects, and VS is much easier to handle now than it was when Weinberg wrote. In Chapter 6, I discuss how VS is continuing to help you stay in the groove despite context switching, but it is still a cognitive challenge.

Summary

As discussed in Chapter 1, in the decade since the Agile Manifesto, the industry has largely reached consensus on software process. Scrum is at its core, complemented with Agile engineering practices, and based on Lean principles. This convergent evolution is the basis for the practices supported by VS.

This chapter addressed how VS, and TFS in particular, enacts process. Microsoft provides three process templates with TFS: Scrum, MSF for Agile Software Development, and MSF for CMMI Process Improvement. All are Agile processes, relying on iterative development, iterative prioritization, continuous improvement, constituency-based risk management, and situationally specific adaptation of the process to the project. Microsoft partners provide more process templates and you can customize your own.

Core to all the processes is the idea of work in nested cycles: check-in, test, sprint, and release. Each cycle has its own definition of *done,* reinforced with tooling in TFS. The definitions of *done* by cycle are the best guards against the accumulation of technical debt and, thus, are the best aids in maintaining the flow of potentially shippable software in every sprint.

Consistent with Scrum, it is important to inspect and adapt not just the software but also the process itself. TFS provides a Process Template Editor to adapt the process to the needs of the project. The process design should reflect meaningful business circumstances and what the team learns as it matures from sprint to sprint.

Finally, inspect and adapt. Plan on investing in process and tooling early to improve the economics of the project over its life span. By following an Agile approach, you can achieve considerable long-term benefits, such as the development of high-quality and modifiable software without a long

tail of technical debt. However, such an approach, and its attendant benefits, requires conscious investment.

The next chapter pulls back to the context around the sprint and discusses product ownership and the many cycles for collecting and acting on feedback. That chapter covers the requirements in their many forms and the techniques for eliciting them and keeping them current in the backlog.

Endnotes

1 Alistair Cockburn coined the phrase *stretch to fit* in his Crystal family of methodologies and largely pioneered this discussion of context with his paper "A Methodology per Project," available at http://alistair.cockburn.us/crystal/articles/mpp/methodologyperproject.html.

2 Ken Schwaber and Jeff Sutherland, *Scrum Guide*, February 2010, available at www.scrum.org/scrumguides/.

3 www.sei.cmu.edu

4 See Gregg Boer's blog, http://blogs.msdn.com/b/greggboer/archive/2012/01/27/tfs-vnext-configuring-your-project-to-have-a-master-backlog-and-sub-teams.aspx.

5 Kent Beck with Cynthia Andres, *Extreme Programming Explained: Embrace Change, Second Edition* (Boston: Addison-Wesley, 2005), 34.

6 Mentioned in the *Scrum Guide,* and discussed in somewhat greater length in Ken Schwaber and Mike Beedle, *Agile Software Development with Scrum* (Upper Saddle River, NJ: Prentice Hall, 2001), 25.

7 *Scrum Guide,* 9.

8 Tom DeMarco and Timothy Lister, *Waltzing with Bears: Managing Risk on Software Projects* (New York: Dorset House, 2003), 130.

9 Mike Cohn, *Agile Estimating and Planning* (Upper Saddle River, NJ: Prentice Hall, 2005).

10 Cockburn, *op. cit.*

11 Ohno, *op. cit.*, 26.

[12] Henrik Kniberg, "Toyota's Journey from Waterfall to Lean Software Development," posted March 16, 2010, at http://blog.crisp.se/henrikkniberg/2010/03/16/1268757660000.html.

[13] David J. Anderson, *Kanban, Successful Evolutionary Change for Your Technology Business* (Seattle: Blue Hole Press, 2010). This control mechanism is very similar to the drum-buffer-rope described by Eli Goldratt in *The Goal.*

[14] Barry Boehm and Richard Turner, *Balancing Agility with Discipline: A Guide for the Perplexed* (Boston: Addison-Wesley, 2004), 152.

[15] Gerald M. Weinberg, *Quality Software Management: Systems Thinking* (New York: Dorset House, 1992), 284.

Index

B

C

D

E

F

G–H

I

J–K–L

M

N–O

P

Q

R

S

T

U

V

W–Z