

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



Visual Studio Team Foundation Server 2012 Adopting Agile Software Practices

From Backlog to Continuous Feedback



 Visual Studio

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.”

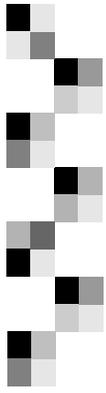
—**Dr. James A. Whittaker**, Software Engineering Director, Google

“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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The Visual Studio and .NET logos are either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries and is used under license from Microsoft.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

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

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

Microsoft, Windows, Visual Studio, Team Foundation Server, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

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

<i>Forewords</i>	<i>xii</i>
<i>Preface</i>	<i>xvi</i>
<i>Acknowledgments</i>	<i>xxiii</i>
<i>About the Authors</i>	<i>xxiv</i>
1 The Agile Consensus	1
The Origins of Agile	2
Agile Emerged to Handle Complexity	2
Empirical Process Models	4
A New Consensus	5
Scrum	6
An Example	12
Self-Managing Teams	14
Summary	15
Endnotes	16
2 Scrum, Agile Practices, and Visual Studio	19
Visual Studio and Process Enactment	20
Process Templates	21
Process Cycles and TFS	24
Inspect and Adapt	37
Task Boards	37
Kanban	38
Fit the Process to the Project	39
Summary	42
Endnotes	43



3	Product Ownership	45
	What Is Product Ownership?	46
	Scrum Product Ownership	50
	Release Planning	51
	Qualities of Service	69
	How Many Levels of Requirements	73
	Summary	75
	Endnotes	75
4	Running the Sprint	77
	Empirical over Defined Process Control	78
	Scrum Mastery	80
	Use Descriptive Rather Than Prescriptive Metrics	86
	Answering Everyday Questions with Dashboards	91
	Choosing and Customizing Dashboards	98
	Using Microsoft Outlook to Manage the Sprint	100
	Summary	101
	Endnotes	101
5	Architecture	103
	Architecture in the Agile Consensus	104
	Exploring Existing Architectures	107
	Summary	124
	Endnotes	126
6	Development	129
	Development in the Agile Consensus	130
	The Sprint Cycle	131
	Keeping the Codebase Clean	132
	Staying “in the Groove”	139
	Detecting Programming Errors Early	143
	Catching Side Effects	154
	Preventing Version Skew	162
	Making Work Transparent	170
	Summary	171
	Endnotes	173
7	Build and Lab	175
	Cycle Time	176
	Defining <i>Done</i>	177

Continuous Integration	179
Automating the Build	181
Automating Deployment to Test Lab	186
Elimination of Waste	199
Summary	203
Endnotes	204
8 Test	207
Testing in the Agile Consensus	208
Testing Product Backlog Items	211
Actionable Test Results and Bug Reports	215
Handling Bugs	223
Which Tests Should Be Automated?	223
Automating Scenario Tests	224
Load Tests, as Part of the Sprint	228
Production-Realistic Test Environments	234
Risk-Based Testing	236
Summary	238
Endnotes	239
9 Lessons Learned at Microsoft Developer Division	241
Scale	242
Business Background	243
Improvements after 2005	247
Results	256
Acting on the Agile Consensus	256
Lessons Learned	258
The Path to Visual Studio 2012	262
Endnotes	263
10 Continuous Feedback	265
Agile Consensus in Action	266
Continuous Feedback Allows Build/Measure/Learn	267
There's No Place Like Production	269
Summary	271
Endnotes	274
Index	275



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.

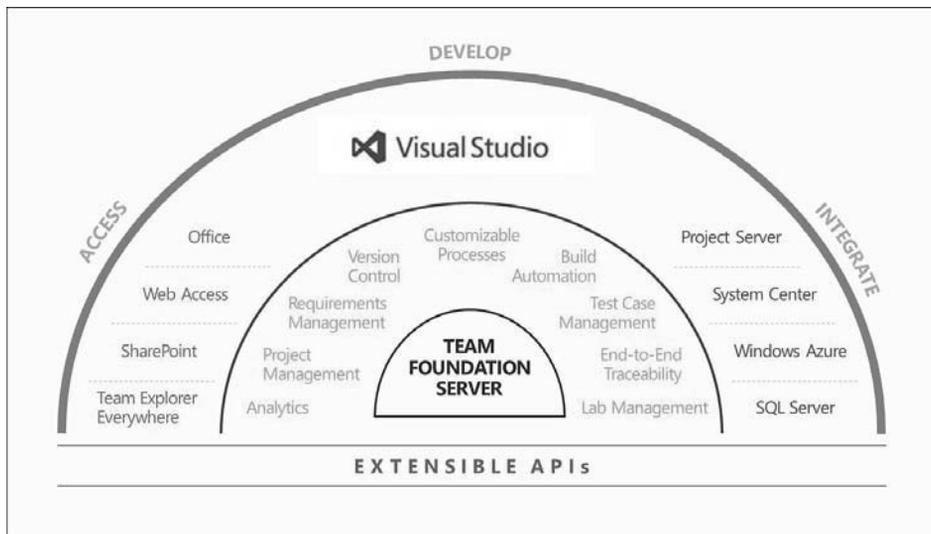


FIGURE P.1: Team Foundation Server forms the collaboration hub of VS 2012. The client components are available in VS Ultimate.

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/.

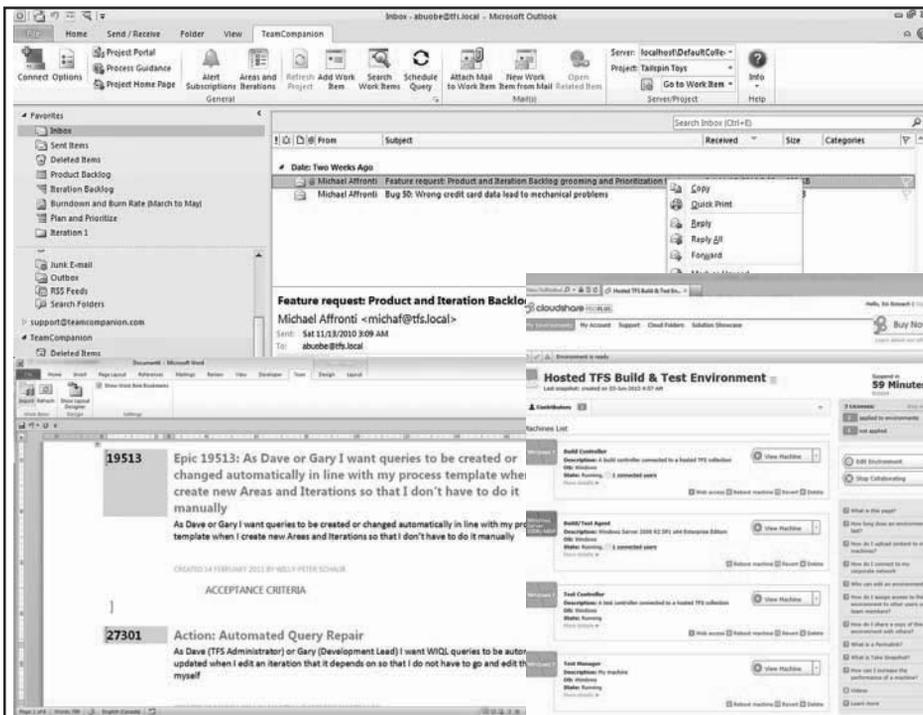


FIGURE P.2: A broad catalog of partner products extend TFS. Shown here are Ekobit TeamCompanion, CloudShare hosted dev/test labs, and the open source TFS Word Add-in available on CodePlex.

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 through 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.

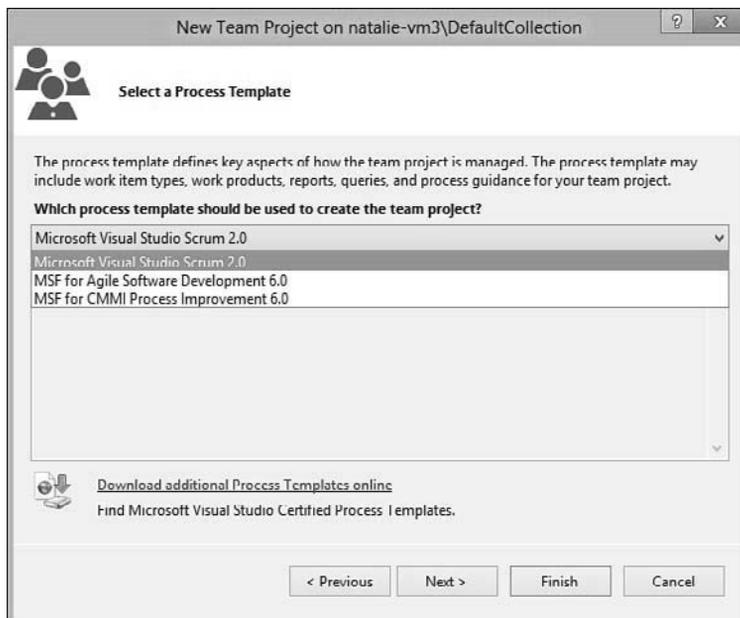


FIGURE 2.2: The Project Creation Wizard lets you create a team project based on any of the currently installed process templates.

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

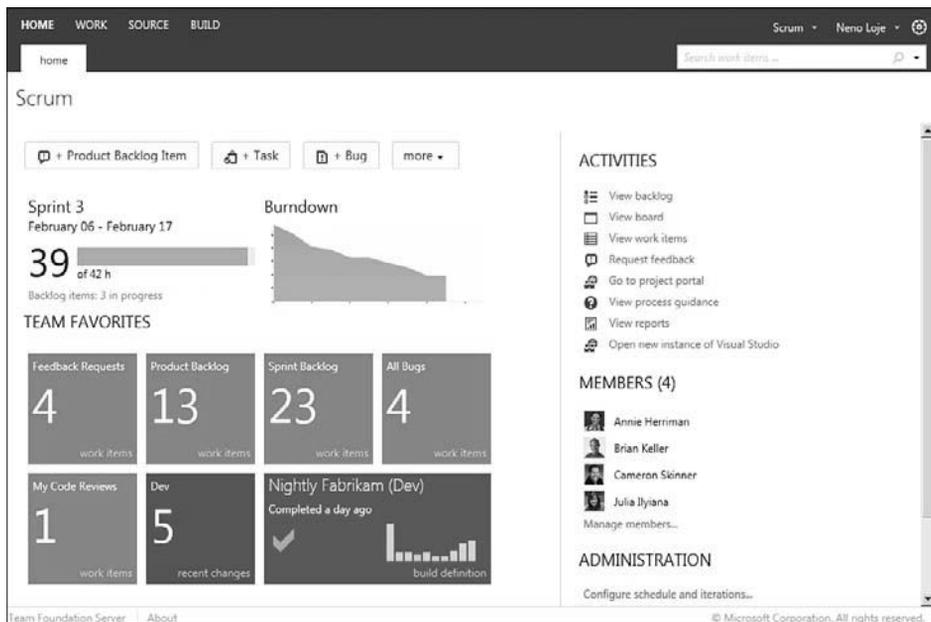


FIGURE 2.3: The tiles on the team’s home page represent the team’s current progress as well as favorite metrics, including work items, testing, source control, and the automated build and test results. Additionally, you can jump to all important views from this page.

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

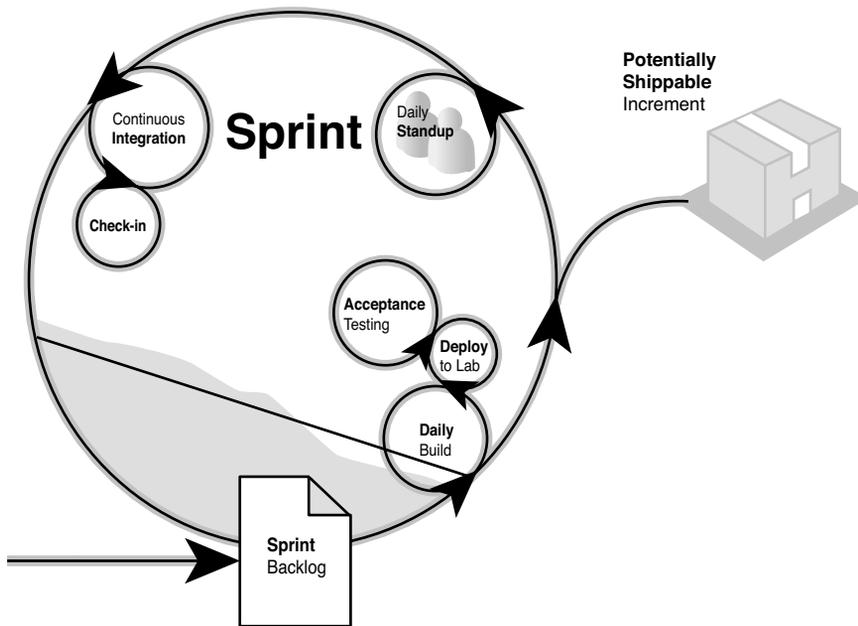


FIGURE 2.4: Software projects proceed on many interlocking cycles, ranging from the “code-edit-test-debug-check in” cycle, measured in minutes, to continuous integration, to daily testing cycles, to the sprint. These are views of both the process and the flow of data, automated by the process tooling.

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

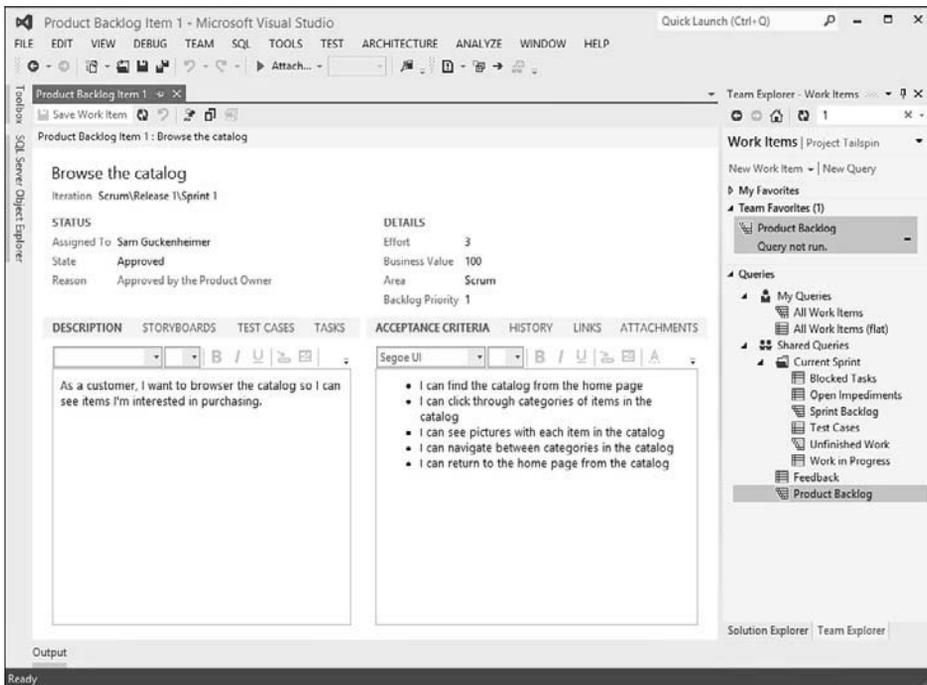


FIGURE 2.5: A product backlog item, shown here as accessed inside the VS IDE, can also be viewed from the Web Portal, Microsoft Excel, Microsoft Project, and many third-party plug-in tools available for TFS.

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).

The screenshot shows the Scrum Backlog interface. At the top, there are tabs for 'backlog', 'board', and 'work items'. The 'backlog' tab is active. On the left, there is a 'Product Backlog' section with a tree view showing 'Current' (Sprint 1) and 'Future' (Sprints 2-6). The main area displays 'Sprint 1' with a date range of 'May 21 - June 08' and '10 work days remaining'. Below this, there are two sub-sections: 'contents' and 'capacity'. The 'contents' section shows a list of tasks grouped by PBIs. The tasks are listed in a table with columns: ID, Title, State, Work Item..., Assigned To, Rema..., Activity, and Effort.

ID	Title	State	Work Item...	Assigned To	Rema...	Activity	Effort
28	New customer	Committed	Product...	Sam Gucken...	29		5
31	Verify user details - validation	To Do	Task		3	Developm...	
32	Save to database	In Progress	Task	Phil Hodgson	3	Developm...	
35	New user screen	In Progress	Task	Aaron Bjork	1	Developm...	
33	Email informing customer of credentials	To Do	Task		5	Developm...	
34	Testing of validation	To Do	Task		2	Testing	
36	Customer validation after receiving email	To Do	Task	Aaron Bjork	5	Developm...	
37	Exploratory testing	In Progress	Task	Gregg Boer	10	Testing	
29	Login options - username and password	Committed	Product...	Sam Gucken...	29		3
38	Logging to track multiple failed attempts	To Do	Task		2	Developm...	
39	Login validation	To Do	Task		5	Developm...	
40	Build the database auditing	To Do	Task		5	Developm...	
41	Login visuals	To Do	Task		6	Design	
42	Exploratory testing	To Do	Task		3	Testing	
43	Build the login authentication mechanism	To Do	Task		8	Developm...	
30	Login with home page landing	Approved	Product...	Sam Gucken...	15		3
44	Test the home page navigation	To Do	Task		3	Testing	
45	Test the login UI	To Do	Task		5	Testing	
46	Implement back/forward features	To Do	Task		2	Developm...	
47	Implement all error handling	To Do	Task		2	Developm...	
48	Implement visuals from design	To Do	Task		1	Developm...	
49	Test the landing page	To Do	Task		1	Testing	
50	Exploratory Testing	To Do	Task		1	Testing	

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.

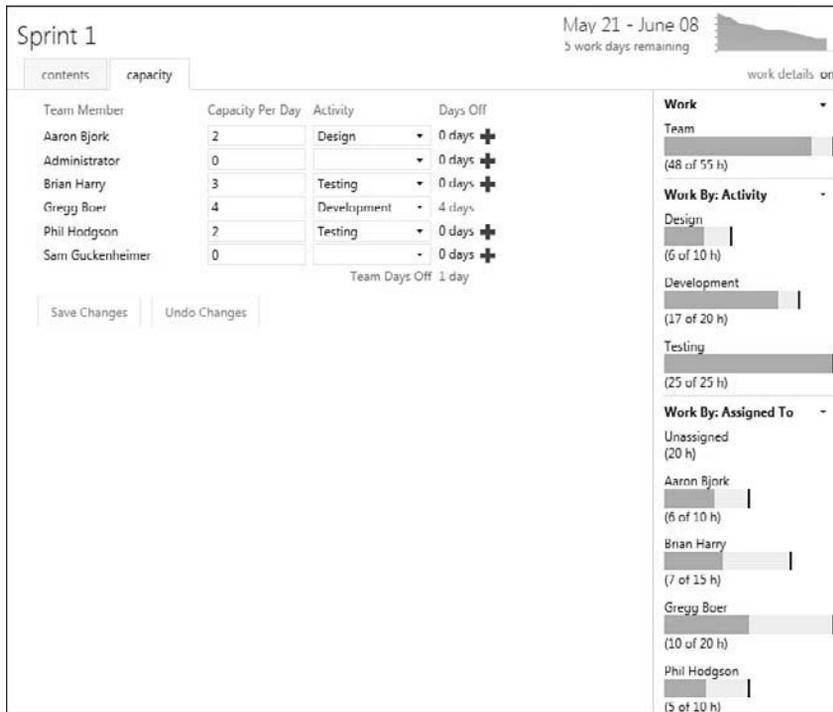


FIGURE 2.7: For each sprint the team sets its capacity. Each individual can identify primary activity and identify planned days off. The work pane compares available time against the estimated hours for the team, both for each team member and grouped at the level of activities like development or testing.

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).

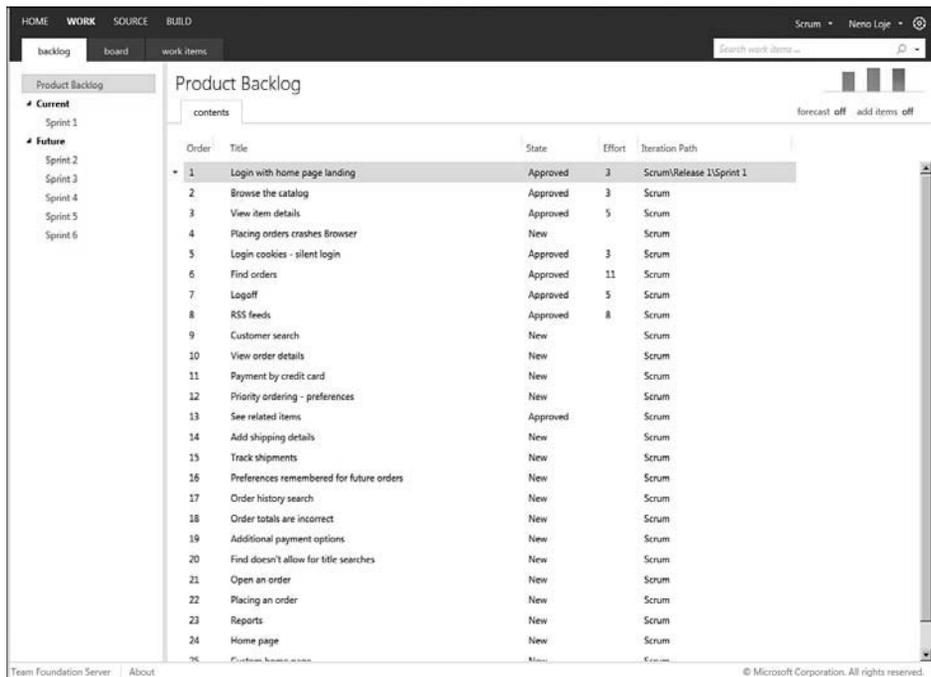


FIGURE 2.8: The product backlog contains the PBIs that express requirements and the bugs that are not handled in the current sprint. This can be accessed from any of the TFS clients; here it is shown in the VS IDE.

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 shelve set 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.

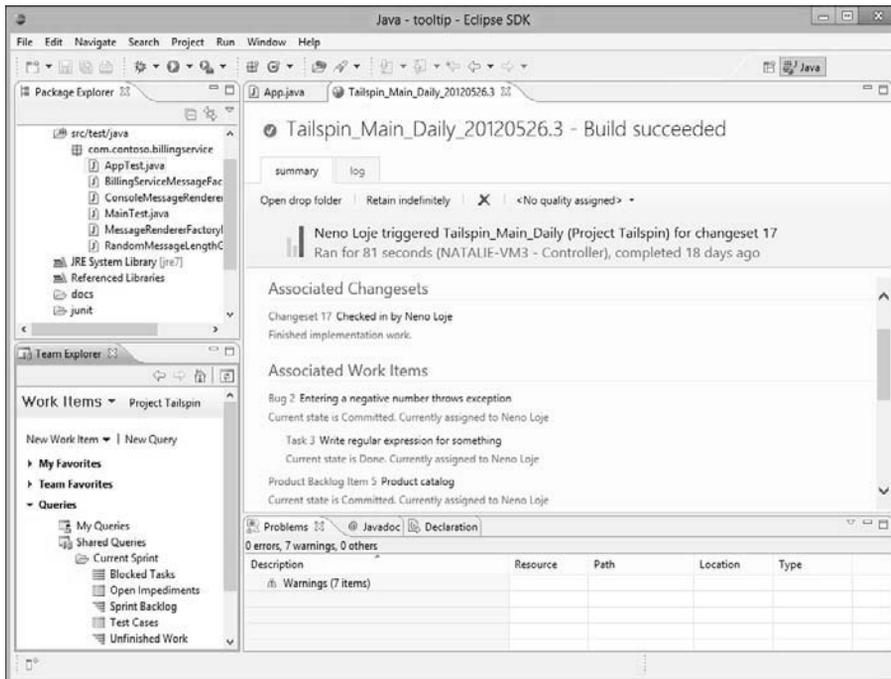


FIGURE 2.9: Every build has a “build details” page that serves as an automated release note, accessible from the dashboard or inside the IDE clients. In this case, it is shown inside Eclipse, as a team working with Java code would see.

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.)

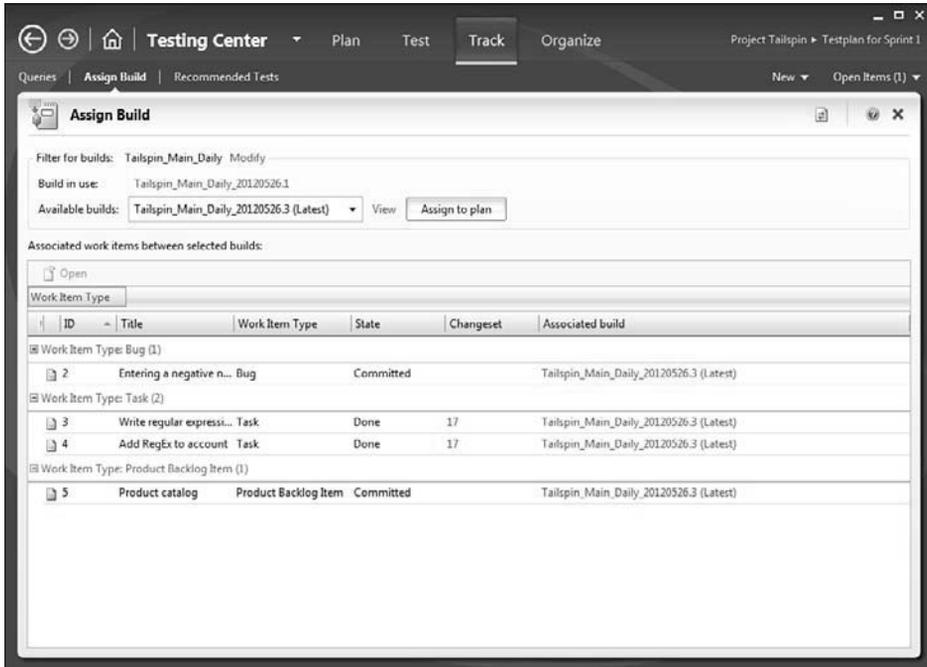


FIGURE 2.10: This build assignment in Microsoft Test Manager is a great way to start the test cycle because it shows the new work delivered since the last tested build and can recommend tests accordingly.

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.

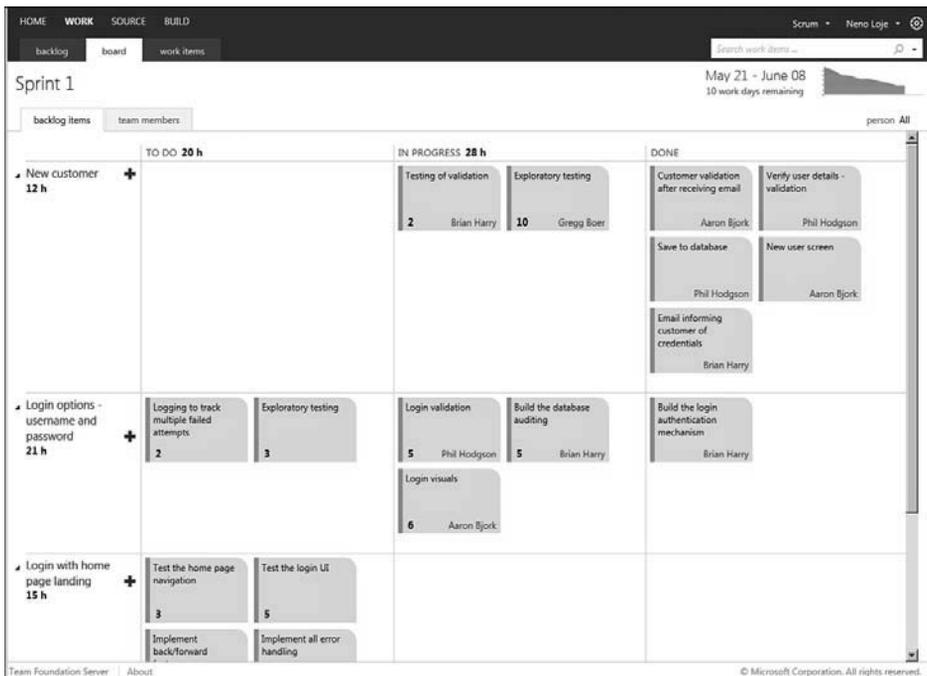


FIGURE 2.11: The TFS task board shows the progress of the sprint backlog visually grouped by PBI. Alternative queries allow different groupings, such as by team member.

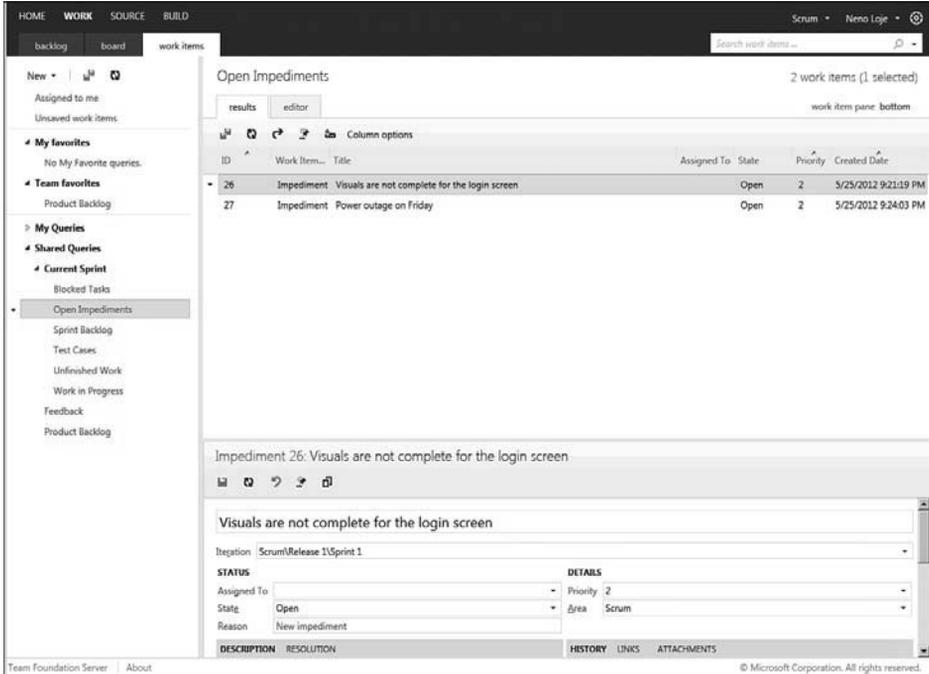


FIGURE 2.12: The Open Impediments query shows the current state of blocking issues as of the daily scrum.

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 “sign-board”) 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.

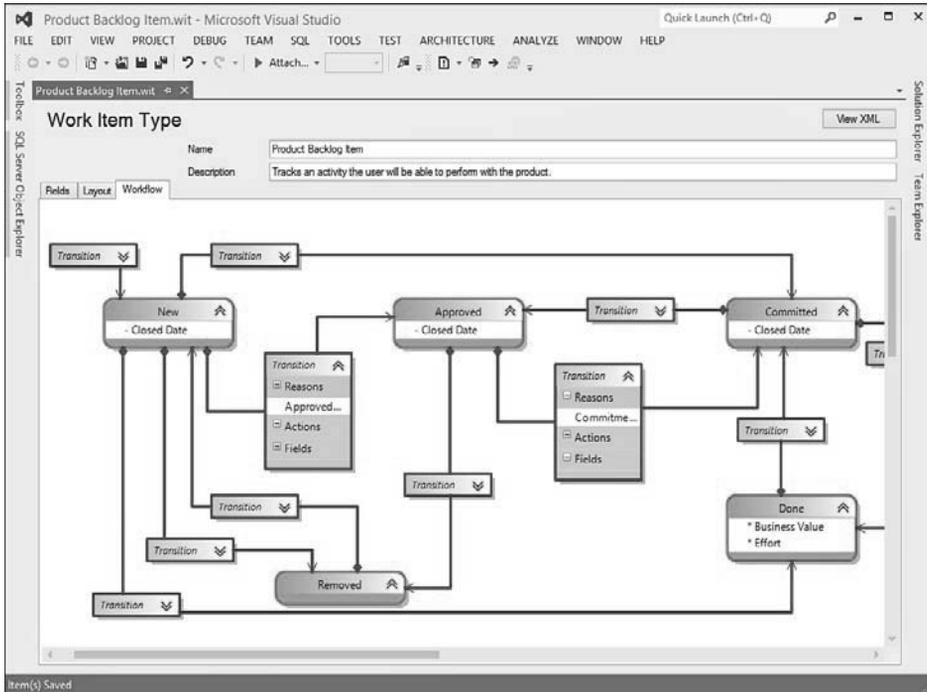


FIGURE 2.13: The Process Template Editor (in the TFS Power Tools on the VS Gallery) enables you to customize work item types, form design, and workflows.

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

7-Day Bug Trend Rates chart, 95
7-Day Issue Trend Rates chart, 93

A

acceptance tests, 27
Active Bugs by Assignment chart, 95
Active Bugs by Priority chart, 95
activity diagrams, 118
Agile Alliance, 2
Agile Consensus
 back to basics, 15
 flow of value
 defined, 5
 Scrum, 8-9
 principles, 5-6
 transparency
 defined, 5
 Scrum, 8
 self-managing teams, 11
 waste reduction
 defined, 5
 Taiichi Ohno's Taxonomy of Waste,
 9-11
Agile Manifesto, 2
ALM (Application Lifecycle
 Management), 258
analysis
 automated code, 152
 paralysis, 31

Anderson, David J., 38
application performance problems,
 diagnosing, 233-234
architecture
 ball of mud, 113
 dependency graphs, creating, 107
 broad coverage, 107
 code indexing, 108
 Quick Cluster layout, 109
 sequence of interactions between,
 110-113
 Solution Explorer, 109
 top-down, 107
designing just enough, 104
documenting, 119
emergent, 105
existing, 107
Explorer, 114
maintainability, 106-107
structures, controlling, 113-117
 code mapping, 114
 existing codebases, 117
 intended dependencies, defining, 114
 layer validation, 115
transparency, 105-106
UML diagrams, 118
 activity, 118
 artifacts, sharing, 120
 component/class/sequence, 119

- extending, 122-124
 - use case, 118
 - work item links, creating, 122
- artifacts, sharing, 120
- Austin, Robert, 86
- automating
 - builds, 181
 - agents, maintaining, 185
 - definitions, maintaining, 184
 - BVTs, 182-183
 - configuring, 181
 - daily builds, 182
 - deployment to test labs, 192, 196
 - reports, 183-184
 - code analysis, 152
 - definition of done, 179
 - scenario tests, 224-227
 - task boards, 37
 - tests, 223-224, 260

B

- back to basics, 15
- back-alley tours, 210
- backlog
 - iteration, 253-254
 - product. *See* product backlog
 - sprint, 27-28
- ball of mud, 113
- baseless merges, 167
- behaviors
 - distorting, 88
 - unexpected, isolating, 154-155
- Beizer, Boris, 219
- Blank, Steve, 267
- bottom-up cycles, 31
- branching, 164
 - benefits, 165
 - by release, 165
 - changes, 167
 - viewing, 167
 - work isolation, 165
- broken windows theory, 90-91
- Brown, Tim, 58

- brownfield projects
 - defined, 107
 - dependency graphs, creating, 107
 - broad coverage, 107
 - code indexing, 108
 - Quick Cluster layout, 109
 - sequence of interaction between, 110-113
 - Solution Explorer, 109
 - top-down, 107
 - layer diagrams, 117
- bugs
 - charts, 94-95
 - dashboard, 95
 - deferral, 245
 - handling, 29, 223
 - Ping-Pong, 12-13
 - Progress Chart, 94-95
 - reactivations, 95
 - reproducing, 218
 - DDAs, 215
 - evolving tests, 219-221
 - immersive app testing on Windows 8, 221
 - solving, 218
 - trends chart, 95
- Build/Measure/Learn, 267-269
- builds
 - agents, 185
 - automated, 181
 - agents, maintaining, 185
 - BVTs, 182-183
 - definitions, maintaining, 184
 - reports, 183-184
 - configuring, 181
 - daily builds, 182
 - waste reduction, 12
 - check-in policies, 137
 - daily, 33
 - chart, 94
 - failures, 202
 - testing, 196
 - dashboard, 98

- definitions, 184
- failures, 202
- process templates, 184
- Quality Indicators report, 171, 203
- test lab deployment, automating, 192, 196
- reports, 183-184
- Status chart, 94
- Success Over Time report, 202
- verification tests, 149, 182-183
- Burndown dashboard, 92-93
- business goals (DevDiv), 243-244
 - company culture, 244-245
 - debt crisis, 246
 - waste, 245
- business value
 - problems, 47
 - release planning, 51
- BVTs (build verification tests), 149, 182-183

C

- Capability Maturity Model Integration (CMMI), 22
- celebrating successes, 261
- Change by Design* (Brown), 58
- changesets, 133, 167
- chaos theory, 273
- chaotic management situations, 3
- charts
 - 7-Day Bug Trend Rates, 95
 - 7-Day Issue Trend Rates, 93
 - Active Bugs by Assignment, 95
 - Active Bugs by Priority, 95
 - Bug Progress, 94-95
 - Bug Reactivations, 95
 - Build Status, 94
 - Code Churn, 95
 - Code Coverage, 95
 - Manual Test Activity, 97
 - Recent Builds, 98
 - Sprint Burndown, 93
 - Task Burndown, 93
 - Task Progress, 93

- Test Case Readiness, 96
- Test Failure Analysis, 98
- Test Plan Progress, 93, 96
- User Story Progress, 93
- User Story Test Status, 97
- check-ins
 - cycle, 31-32
 - error catching, 132-133
 - check-in policies, 135
 - gated check-ins, 136-137
 - policies, 32, 135
 - work items, 140
- CI (continuous integration), 179-180
- class diagrams, 119
- clean codebase
 - catching errors at check-in, 132-133
 - check-in policies, 135
 - gated check-ins, 136-137
 - shelving code, 138-139
- clean layering dependencies, 113
- code mapping, 114
- existing codebases, 117
- intended dependencies, defining, 114
- validation, 115
- clones (code), finding, 151
- cloud, 263
 - feedback, 270-271
 - test environments, 198
- CloudShare, 198
- CMMI (Capability Maturity Model Integration), 22
- code
 - automated analysis, 152
 - brownfield projects, 107
 - Churn chart, 95
 - clean
 - catching errors at check-in, 132-137
 - shelving code, 138-139
 - clones, finding, 151
 - coverage
 - chart, 95
 - monitoring, 203
 - unit test gaps, pinpointing, 147-148

- dependency graphs, creating, 107
 - broad coverage, 107
 - code indexing, 108
 - Quick Cluster layout, 109
 - sequence of interaction between, 110-113
 - Solution Explorer, 109
 - top-down, 107
- indexing, 108
- integrating frequently, 199
- maintenance
 - build verification tests, 149
 - data, varying, 148
 - redundant code, 151-152
 - unit test gaps, pinpointing, 147-148
 - without tests, 145
- metrics, 153
- redundant, 151-152
- reviews, 140-141
- sequence diagrams, 110-113
- shelving, 138-139
- UI tests (Web performance tests), 224-227
 - creating, 225
 - running, 226
 - test data, varying, 227
- Cohn, Mike, 53
- company culture, 244-245
- comparing quantities, 82
- compilers, versioning, 163
- completing PBIs, 199
- complex management situations, 3-4
- complicated management situations, 3
- component diagrams, 119
- configurations
 - automated builds, 181
 - testing
 - critical cases, 189
 - labs, 187-189
 - test machines, 189, 192
- continuous delivery, 177-178
- continuous deployment test labs
 - automating, 192, 196
 - cloud, 198
 - continuous feedback
 - after every sprint cycle, 268
 - Build/Measure/Learn, 267-269
 - cloud, 270-271
 - cycle, 266-269
 - developers, 269
 - priorities/execution, 268
 - production, 269-271
 - tests, 269
 - continuous flow, 262
 - continuous integration (CI), 179-180
 - controlling structures, 113, 117
 - code mapping, 114
 - existing codebases, 117
 - intended dependencies, defining, 114
 - layer validation, 115
 - Conway's law, 244
 - crowd wisdom, 83
 - culture, 244-245
 - cumulative flow diagram, 200-201
 - customers
 - clear goals, 52
 - paint points, 52
 - problems, 47
 - release planning, 53
 - sprint reviews, including, 263
 - user stories, 53
 - validation, 63-69
 - vision statements, 52
 - customizing
 - dashboards, 98-99
 - processes to projects, 39
 - documentation, 41
 - geographic distribution, 40
 - GRC, 41
 - Process Template Editor, 39
 - project switching, 41
 - cycles
 - continuous feedback, 266-269
 - daily. *See* daily cycles
 - process, 24
 - bottom-up, 31
 - check-in, 31-32

- done, defining, 36
- personal development, 31
- release, 24, 27
- sprint, 27
- test, 32-36
- Scrum, 7
- time
 - PBIs, 176-177
 - reducing, 199-203

D

- daily builds, 33
 - chart, 94
 - failures, 202
 - testing, 196
- daily cycles
 - antipatterns, 131-132
 - automated builds, 182
 - branching, 164
 - benefits, 165
 - by release, 165
 - merging changes, 167
 - tracking changes, 167
 - viewing, 167
 - work isolation, 165
- clean codebase
 - catching errors at check-in, 132-133
 - check-in policies, 135
 - gated check-ins, 136-137
 - shelving code, 138-139
- Eclipse/Windows shell, 169
- existing code maintenance
 - build verification tests, 149
 - data, varying, 148
 - gaps, pinpointing, 147-148
 - redundant code, 151-152
 - without tests, 145
- interruptions, minimizing, 139
 - checking in work, 140
 - code reviews, 140-141
 - My Work pane, 139
 - suspending work, 140
- programming errors, catching, 143
 - automated code analysis, 152
 - code metrics, calculating, 153
 - TDD, 143-145
- scrums, 34-36
- side effects, 154
 - operational issues, isolating, 157
 - performance, tuning, 159-162
 - unexpected behaviors, isolating, 154-155
- transparency, 170-171
- versioning, 162-163
- dashboards
 - Bugs, 95
 - Build, 98
 - Burndown, 92-93
 - customizing, 98-99
 - overview, 91
 - Quality, 93-95
 - Test, 96-98
- data, querying, 100
- database schema, versioning, 163
- DDAs (diagnostic data adapters), 215
- debt crisis, 246
- deferring bugs, 245
- defined process models, 3, 78-80
- defining done, 177-178
 - post-2005 improvements, 248-249
 - validating
 - build automation, 181-185
 - continuous integration (CI), 179-180
- delivering software continuously, 177-178
- dependencies
 - clean layering, 113
 - code mapping, 114
 - existing codebases, 117
 - intended dependencies, defining, 114
 - validation, 115
- graphs, creating, 107
 - broad coverage, 107
 - code indexing, 108

- format, 124
- Quick Cluster layout, 109
- sequence of interactions between, 110-113
- Solution Explorer, 109
- top-down, 107
- unwanted, viewing, 115
- deployment
 - builds to test labs, 192-196
 - continuous, 192-198
 - test machines, 192
- descriptive metrics, 89
- designs
 - architecture, 104
 - levels of requirements, 73
 - load tests, 228
 - manageability, 71-72
 - performance, 70
 - products
 - desirability, viability, feasibility, 58-59
 - storyboards, 60-62
 - security/privacy, 70
 - user experience, 70
- desirability, 59
- DevDiv (Microsoft Developer Division), 242
- business goals, 243-244
 - company culture, 244-245
 - debt crisis, 246
 - waste, 245
- improvements after 2005, 247
 - defining done, 249
 - engineering principles, 256
 - flow of value, 256-258
 - iteration backlog, 253-254
 - MQ, 247
 - product planning, 250-252
 - results, 256
- lessons learned
 - broken quality gates, 261
 - product backlog, planning, 259
 - product ownership, 259
 - quality fundamentals, ensuring early, 259
 - social contracts, renewing, 258
 - successes, celebrating, 261
 - teams effects on each other, 260
 - test automation, 260
- scale of work, 242-243
- Visual Studio 2012, 262-263
 - continuous flow, expanding, 262
 - customer feedback, 263
 - geographic coordination, 262
 - including customers in sprint reviews, 263
 - public cloud, 263
- Developer Team (Scrum), 22
- development
 - continuous feedback, 269
 - daily activities. *See* daily cycles
 - potentially shippable increment, 130
- DGML (Directed Graph Markup Language), 124
- diagnostic data adapters (DDAs), 215
- diagrams
 - activity, 118
 - class, 119
 - component, 119
 - extending, 122-124
 - layer
 - code mapping, 114
 - existing codebases, 117
 - extensibility, 117
 - intended dependencies, defining, 114
 - technical debt, reducing, 117
 - validation, 115
- sequence
 - dependencies, 110-113
 - UML, 119
- storing, 118
- UML, 118
 - activity, 118
 - artifacts, sharing, 120
 - component/class/sequence, 119

- extending, 122-124
 - use case, 118
 - work item links, creating, 122
 - use case, 118
 - work item links, creating, 122
- Directed Graph Markup Language (DGML), 124
- dissatisfies, 54-58
- distortion, preventing, 89-90
- documentation
 - architectures, 119
 - fitting processes to projects, 41
 - models, 117
- domains (UML diagrams), 118
 - activity, 118
 - artifacts, sharing, 120
 - component/class/sequence, 119
 - extending, 122-124
 - use case, 118
 - work item links, sharing, 122
- done
 - broken windows theory, 90-91
 - defining, 36, 177-178
 - post-2005 improvements, 248-249
 - validating, 179-185
 - measuring, 32
 - server enforcement, 136-137
- E**
- Eclipse development tools, 169
- edge of chaos, 3-4, 273
- elevator pitch, 53
- eliminating waste. *See* waste, eliminating
- emergent architecture, 105
- empirical process model, 5, 80
- engineering principles, 256
- equivalence classes, 227
- errors
 - catching at check-in, 132-133
 - check-in policies, 135
 - gated check-ins, 136-137
 - operational, isolating, 157
 - performance, diagnosing, 159-162
 - programming, catching, 143
 - automated code analysis, 152
 - code metrics, calculating, 153
 - TDD, 143-145
- estimating PBIs, 82-84
 - benefits, 82
 - crowd wisdom, 83
 - disadvantages, 84
 - inspect and adapt, 83
 - quantity comparisons, 82
 - rapid cognition, 83
 - story point estimates, 82
 - velocity, measuring, 83-84
- exciters, 54-58
- existing architectures
 - code, 107
 - dependency graphs, creating, 107
 - broad coverage, 107
 - code indexing, 108
 - Quick Cluster layout, 109
 - sequence of interactions between, 110-113
 - Solution Explorer, 109
 - top-down, 107
- structures, controlling, 113-117
 - code mapping, 114
 - existing codebases, 117
 - intended dependencies, defining, 114
 - layer validation, 115
- UML diagrams, 118
 - activity, 118
 - artifacts, sharing, 120
 - component/class/sequence, 119
 - extending, 122-124
 - use case, 118
 - work item links, creating, 122
- exploratory testing, 210, 219-221
- extensibility
 - diagrams, 122-124
 - layer diagrams, 117
- extra processing, 10

F

failures. *See* bugs
 fault models, 236
 feasibility, 58
 features
 crews, 249
 product planning, 252
 progress, 254
 feedback
 after every sprint cycle, 268
 Build/Measure/Learn, 267
 cloud, 270-271
 continuous
 Build/Measure/Learn, 269
 cycle, 266-267
 cycle activities, 269
 customers, 52
 clear goals, 52
 pain points, 52
 user stories, 53
 validation, 63-69
 vision statements, 52
 cycle activities, 268
 developers, 269
 effective, 64
 Feedback Client for TFS, 65-66
 priorities/execution, 268
 production, 269-271
 querying, 67
 requests, creating, 64-69
 responses, 66
 test labs, 269
 testing from user perspective, 269
 unit tests, 269
 files, versioning, 163
 flow
 continuous, 262
 cumulative flow diagram, 200-201
 inefficiencies, detecting, 200
 measures, 271
 remaining work, tracking, 200-201
 storyboards, 60-62

value

 defined, 5
 post-2005 improvements, 256-258
 product backlog, 8-9
 testing, 209
 transparency/waste reduction,
 reinforcing, 6

G–H

gated check-ins, 136-137, 179-180
 geographic distribution, 40
 goals, customer feedback, 52
 graphs (dependency), creating, 107
 broad coverage, 107
 code indexing, 108
 format, 124
 Quick Cluster layout, 109
 sequence of interactions between,
 110-113
 Solution Explorer, 109
 top-down, 107
 GRC (governance, risk management,
 and compliance), 41
 Great Recession, 273
 greenfield projects, 107
 handling bugs, 29, 223
 HR practices (Microsoft), 244
 Hyper-V Server, 188

I

immersive app testing (Windows 8), 221
 improvements after 2005
 broken quality gates, 261
 defining done, 249
 DevDiv, 247
 engineering principles, 256
 flow of value, 256-258
 iteration backlog, 253-254
 MQ, 247
 product
 backlog, 259
 ownership, 259
 planning, 250-252

- quality fundamentals, ensuring
 - early, 259
 - results, 256
 - social contracts, renewing, 258
 - successes, celebrating, 261
 - teams effects on each other, 260
 - test automation, 260
 - indexing code, 108
 - individual performance, measuring, 86
 - inspect and adapt, 37, 83
 - inspecting working software, 105
 - installing test machines, 189
 - integration
 - continuous (CI), 136-137, 179-180
 - feature crews, 249
 - frequent, 199
 - IntelliTrace, 155
 - interruptions, minimizing, 139
 - checking in work items, 140
 - code reviews, 140-141
 - My Work pane, 139
 - suspending work items, 140
 - iron triangle, 78
 - isolating
 - feature crews, 249
 - operational issues, 157
 - unexpected behaviors, 154-155
 - work, 165
 - iteration backlog improvements, 253-254
- J-K-L**
- Kanban, 38
 - Kaner, Cem, 90
 - Kano analysis, 55-58
 - labs. *See* tests, labs
 - layer diagrams
 - code mapping, 114
 - existing codebases, 117
 - extensibility, 117
 - intended dependences, defining, 114
 - technical debt, reducing, 117
 - validation, 115
 - layering dependencies
 - clean, 113
 - code mapping, 114
 - existing codebases, 117
 - intended dependencies, defining, 114
 - validation, 115
 - Lean origins, 1
 - lessons learned. *See* DevDiv, lessons learned
 - load testing, 228
 - designing, 228
 - output, 232
 - performance problems, diagnosing, 233-234
 - Logan, Dave, 244
- M**
- The Machine That Changed the World* (Womack), 1
 - maintainability, 106-107
 - maintenance
 - builds, 184-185
 - existing code
 - build verification tests, 149
 - data varying, 148
 - redundant code, 151-152
 - unit test gaps, pinpointing, 147-148
 - without tests, 145
 - management
 - designs, 71-72
 - self, 15
 - Scrum, 7
 - Toyota example, 14
 - transparency, 11
 - situations, 3-4
 - sprint, 100
 - manual tests
 - Activity chart, 97
 - playing, 216
 - mastering Scrum, 80-81
 - contrasting techniques, 84
 - estimation (Planning Poker), 82-84
 - team sizes, 81
 - McConnell, Steve, 79

- measuring
 - done, 32
 - individual performance, 86
 - success, 104
 - velocity, 83-84
 - merging branches, 167
 - methodologies (Scrum)
 - cycles, 7
 - potentially shippable increments, 8
 - product backlog, 8-9
 - self-managing teams, 7
 - metrics
 - broken windows theory, 90-91
 - descriptive, 89
 - distortion, preventing, 89-90
 - prescriptive, 87-88
 - programming errors, catching, 153
 - Microsoft
 - Developer Division. *See* DevDiv
 - HR practices, 244
 - Outlook, 100
 - public cloud, 263
 - Test Manager. *See* MTM
 - milestone quality (MQ), 247
 - minimizing
 - interruptions, 139
 - checking in work items, 140
 - code reviews, 140-141
 - My Work pane, 139
 - suspending work items, 140
 - overhead, 39
 - models
 - documenting, 117
 - process
 - defined, 3, 78-80
 - empirical, 5, 80
 - projects
 - artifacts, sharing, 120
 - UML diagrams, 118-119, 122-124
 - work item links, creating, 122
 - monitoring, 203
 - motion, 10
 - MQ (milestone quality), 247
 - MSF for Agile Software Development
 - process template, 22
 - MSF for CMMI Process Improvement
 - process template, 22
 - MTM (Microsoft Test Manager), 33, 211
 - bugs, reproducing, 218
 - DDAs (diagnostic data adapters), 215
 - exploratory testing, 219-221
 - immersive app testing on
 - Windows 8, 221
 - solving, 218
 - build comparisons, 33
 - manual tests, playing, 216
 - Recommended Tests list, 213-214
 - test cases
 - inferring, 212
 - organizing/tracking, 213
 - plans, 213
 - running, 216
 - shared steps, 214
 - test steps, 214
 - muda, 9-10
 - Multi-Tier Analysis, 159-160
 - mura, 9-10
 - muri, 9-10
 - My Work pane
 - checking in work items, 140
 - code reviews, 140-141
 - personal task backlog, organizing, 139
 - suspending work items, 140
- ## N–O
- negative tests, 210
 - OData (Open Data Protocol), 100
 - operational issues, isolating, 157
 - organizing test cases, 213
 - overburden, 10
 - overhead, minimizing, 39
 - overproduction, 10
 - ownership. *See* product ownership

P

- pain points, 52
- PBIs. *See* product backlog, items
- peanut buttering, 47, 250
- performance
 - application problems, diagnosing, 233-234
 - backlog, ensuring, 259
 - individuals, measuring, 86
 - QoS, 70
 - tuning, 159-162
 - Wizard, 159-160
- perishable requirements, 48-50
- permissions, 24
- personal task backlog, organizing, 139
- pesticide paradox, 219
- The Pet Shoppe* (Python sketch), 47
- planning
 - products, 250
 - backlog, 259
 - experiences, 252
 - features, 252
 - scenarios, 251-252
 - taxonomy, 250
 - value propositions, 251
 - release, 51
 - business value, 51
 - customer value, 52-53
 - scale, 54
 - tests, 213
- Planning Poker, 27, 82-84
 - benefits, 82
 - crowd wisdom, 83
 - disadvantages, 84
 - inspect and adapt, 83
 - quantity comparisons, 82
 - rapid cognition, 83
 - story point estimates, 82
 - velocity, measuring, 83-84
- plug-ins, 143
- policies
 - build check-in, 137
 - check-in, 32, 135
- Poppendieck, Mary and Tom, 10
- post-2005 improvements
 - DevDiv, 247
 - done, defining, 248-249
 - engineering principles, 256
 - flow of value, 256-258
 - iteration backlog, 253-254
 - lessons learned
 - broken quality gates, 261
 - product backlog, planning, 259
 - product ownership, 259
 - quality fundamentals, ensuring early, 259
 - social contracts, renewing, 258
 - successes, celebrating, 261
 - teams effects on each other, 260
 - test automation, 260
 - MQ, 247
 - product planning, 250-252
 - results, 256
- potentially shippable increments, 8, 130
- prescriptive metrics, 87-88
- privacy (QoS), 70
- Process Template Editor, 39
- processes
 - Agile, 2
 - customizing to projects, 39
 - documentation, 41
 - geographic distribution, 40
 - GRC, 41
 - Process Template Editor, 39
 - project switching, 41
- cycles, 24
 - bottom-up, 31
 - check-in, 31-32
 - done, defining, 36
 - personal development, 31
 - release, 24, 27
 - sprint, 27
 - test, 32-36
- enactment, 20
- models
 - defined, 3, 78-80
 - empirical, 5, 80

- team structures
 - permissions, 24
 - Scrum, 22
 - TFS, 23
 - templates, 21-22
 - product backlog, 8-9
 - business value, 51
 - creating, 51
 - customer feedback, 52
 - exciters, satisfiers, dissatisfiers, 54-58
 - items
 - acceptance tests, 27
 - bugs, handling, 29
 - completing, 199
 - cycle time, 176-177
 - diagram links, 122
 - estimating. *See* estimating PBIs, 82-84
 - minimum quality, 178
 - progress chart, 93
 - release cycle, 25
 - testing, 213-214
 - planning, 259
 - QoS, 69
 - ensuring, 259
 - manageability, 71-72
 - performance, 70
 - security and privacy, 70
 - user experience, 70
 - release cycle definition, 25
 - requirements, 73
 - scale, 54
 - sprint backlog, compared, 28
 - user stories, 53
 - velocity, 84
 - work breakdown, 73
- Product Owner (Scrum), 22
- product ownership
 - defined, 46
 - explicit agreement, 259
 - problems, 47-48
 - QoS, 69-72
 - release planning, 51
 - business value, 51
 - customer validation, 63-69
 - customer value, 52-53
 - design, 58-59
 - exciters, satisfiers, dissatisfiers, 54-58
 - scale, 54
 - storyboards, 60-62
 - requirements
 - levels, 73
 - perishable, 48-50
 - Scrum, 50
 - work breakdown, 73
 - production
 - feedback, 269-271
 - realistic test environments, 234-235
 - products
 - backlog. *See* product backlog
 - design
 - desirability, viability, feasibility, 58-59
 - storyboards, 60-62
 - desirability, 59
 - feasibility, 58
 - ownership. *See* product ownership
 - performance/reliability, 259
 - planning, 250-252
 - viability, 58
 - programming errors, catching, 143
 - automated code analysis, 152
 - code metrics, calculating, 153
 - TDD, 143-145
 - programs, support, 163
 - progress, viewing, 254
 - projects
 - brownfield. *See* brownfield projects
 - Creation Wizard, 21
 - greenfield, 107
 - switching, 41
 - public cloud (Microsoft), 263
 - Python, Monty, 47

Q

QoS

- backlog
 - ensuring, 259
 - minimum, 178
- done, defining, 177-178
- potentially shippable increment, 130
- requirements, 69
 - manageability, 71-72
 - performance, 70
 - security and privacy, 70
 - user experience, 70
- understanding during sprints, 106
- Quality dashboard, 93-95
- quality gates, 248, 261
- quantity comparisons, 82
- query-based suites, 213
- querying data, 100
- Quick Cluster layout (dependency graphs), 109

R

- rapid cognition, 83
- Rapid Development* (McConnell), 79
- reactivations (bugs), 95
- readiness, 96
- Recent Builds chart, 98
- Recommended Tests list (MTM), 213-214
- Red-Green-Refactor, 143
- reducing
 - cycle time
 - build failures, 202
 - code coverage/tests, monitoring, 203
 - flow inefficiencies, detecting, 200
 - integrating frequently, 199
 - PBIs, completing, 199
 - remaining work, tracking, 200-201
 - waste
 - Bug Ping-Pong, 12-13
 - build automations, 12
 - defined, 5
 - flow of value/transparency, reinforcing, 6
 - Taiichi Ohno's Taxonomy of Waste, 9-11
 - testing, 210
- redundant code, 151-152
- release planning, 24, 27, 51
 - business value, 51
 - customer validation, 63-69
 - customer value, 52-53
 - design, 58-59
 - exciters, satisfiers, dissatisfiers, 54-58
 - scale, 54
 - storyboards, 60-62
 - user stories, 53
- released versions (solutions), tracking, 165
- reliability, 259
- remaining work, tracking, 200-201
- Remove Customer Dissatisfiers, 252
- renewing social contracts, 258
- reports
 - Build, 183-184
 - Build Quality Indicators, 171, 203
 - Build Success Over Time, 202
 - production realistic test environments, 234
- reproducing bugs, 218
 - DDAs, 215
 - evolving tests, 219-221
 - immersive app testing on Windows 8, 221
 - solving, 218
- requirements
 - perishable, 48-50
 - product backlog, 73
 - QoS, 69-72
- results, 256
- reviews
 - code, 140-141
 - sprints, 263
- Ries, Eric, 267

risks

- testing, 236-238
 - capturing risks as work items, 237
 - fault models, 236
 - security testing, 238
- work items, 237

S

satisfiers, 54-58

scale

- user stories, 54
- work, 242-243

scenarios

- product planning, 251-252
- tests, automating, 224-227

Schmea Compare, 163

Schwaber, Ken, 3

scope creep problems, 48

Scrum

- cycles, 7, 24-27
 - daily, 34-36
 - inspect and adapt, 37
 - mastery, 22, 80-81
 - contrasting techniques, 84
 - estimation (Planning Poker), 82-84
 - team sizes, 81
 - overview, 80
 - potentially shippable increments, 8, 130
 - process template, 21
 - product backlog, 8-9
 - product ownership, 50
 - self-managing teams, 7, 11
 - task boards, 37-38
 - taxonomy of waste, 9-11
 - team structures, 22
- SCVMM test environments
(System Center Virtual Machine
Manager), 188
- security
- QoS, 70
 - testing, 238

self-managing teams, 15

- Scrum, 7
- Toyota example, 14
- transparency, 11

sequence diagrams

- dependencies, 110-113
- UML, 119

servers, done enforcement, 136-137

sharing

- artifacts, 120
 - work item steps, 214
- shelving code, 138-139

side effects, 154

- operational issues, isolating, 157
- performance, tuning, 159-162
- unexpected behaviors, isolating, 154-155

simple management situations, 3

social contracts, renewing, 258

software, working, 105

Sogeti Test Management Approach
(TMap) process template, 22

Solution Explorer, 109

Source Code Explorer, 167

Source Control Explorer, 163

sprint, 27

backlog, 27-28

bugs

- charts, 94
- handling, 29
- trends, 95

burndown charts, 93

code coverage/churn, 95

crowd wisdom, 83

daily builds chart, 94

dashboards

- Bugs, 95
- Build, 98
- Burndown, 92-93
- overview, 91
- Quality, 93-95
- Test, 96-98

- impediments, 93
 - inspect and adapt, 37, 83
 - managing with Microsoft Outlook, 100
 - metrics
 - broken windows theory, 90-91
 - descriptive, 89
 - distortion, preventing, 89-90
 - prescriptive, 87-88
 - overview, 80
 - Planning Poker, 27, 82-84
 - QoS, understanding, 106
 - quantity comparison, 82
 - rapid cognition, 83
 - reviews, 263
 - task boards, 37-38
 - team sizes, 81
 - test cases
 - progress, tracking, 93
 - readiness, 96
 - timeboxing, 31
 - velocity, 83-84
 - Stacey Matrix, 3-4
 - Stacey, Ralph D., 3
 - standard process templates, 21
 - standard test environments, 188
 - staying in the groove, 139
 - checking in work items, 140
 - code reviews, 140-141
 - My Work pane, 139
 - suspending work items, 140
 - storing diagrams, 118
 - story points
 - estimating, 82
 - measuring, 83-84
 - story units, 82
 - storyboards, 60-62
 - Strategic Management and Organisational Dynamics* (Stacey), 3
 - structures, control, 113-117
 - code mapping, 114
 - existing codebases, 117
 - intended dependencies, defining, 114
 - layer validation, 115
 - successes
 - celebrating, 261
 - measuring, 104
 - support programs, versioning, 163
 - suspending work items, 140
 - Swedish Vasa*, 48
 - System Center Operations
 - Manager, 157
 - System Center Virtual Machine
 - Manager (SCVMM), 188
- T**
- tacit knowledge, 41
 - Taiichi Ohno's Taxonomy of Waste, 9-11
 - task boards, 37-38
 - Task Burndown chart, 93
 - Task Progress chart, 93
 - taxonomy of waste (Taiichi Ohno), 9-11
 - TDD (test-driven development), 143-145
 - Team Companion, 100
 - teams
 - behaviors, distorting, 88
 - distortion, preventing, 89-90
 - effects on each other, 260
 - geographic distribution, 40
 - meetings, 34-37
 - permissions, 24
 - project switching, 41
 - self-managing, 15
 - Scrum, 7
 - Toyota, 14
 - transparency, 11
 - sizes, 81
 - structures, 22-23
 - task boards, 37-38
 - velocity, measuring, 83-84
 - technical debt, 11-12
 - defined, 11
 - layer diagrams, 117

- templates
 - build process, 184
 - process, 21
 - custom, 22
 - customizing, 39-41
 - MSF for Agile software Development, 22
 - MSF for CMMI Process Improvement, 22
 - Project Creation Wizard, 21
 - Scrum, 21
 - Sogeti Test Management Approach (TMap) process template, 22
- tests
 - acceptance, 27
 - automating, 223-224
 - automation, 260
 - bugs, reproducing, 218
 - DDAs, 215
 - evolving tests, 219-221
 - immersive app testing on Windows 8, 221
 - solving, 218
 - cases
 - inferring, 212
 - organizing/tracking, 213
 - readiness, 96
 - shared steps, 214
 - sprint, 93
 - steps, 214
 - choosing, 213-214
 - configuration, 189, 192
 - critical cases, 189
 - deploying test machines, 192
 - installing test machines, 189
 - cycles, 32-36
 - daily builds, 196
 - dashboard, 96-98
 - driven development (TDD), 143-145
 - exploratory, 210, 219-221
 - Failure Analysis chart, 98
 - flow of value, 209
 - handling bugs, 223
 - integrating frequently, 199
- labs
 - build deployment, automating, 192, 196
 - cloud, 198
 - continuous feedback, 269
 - Management feature, 188
 - multiple, 196
 - physical and virtual machines, 189, 192
 - production-realistic, 234-235
 - SCVMM, 188
 - setting up, 187-189
 - standard, 188
 - support, 186
 - tests, running, 196
- load, 228
 - designing, 228
 - output, 232
 - performance problems, diagnosing, 233-234
- Management Approach (TMap) process template, 22
- manual
 - Activity chart, 97
 - playing, 216
- monitoring, 203
- MTM. *See* MTM
- negative, 210
- Plan Progress chart, 93, 96
- plans, 213
- production-realistic environments, 234-235
- risks, 236-238
- running, 216
- scenario, automating, 224-227
- security, 238
- settings, 216
- test-run failures chart, 98
- transparency, 211
- unit. *See* unit testing
- waste reduction, 210
- Web performance, 225-227

timeboxing sprint planning, 31
 TMap (Test Management Approach)
 process template, 22
 top-down dependency graphs, 107
 tours, 210
 Toyota self-management example, 14
 tracking
 branch changes, 167
 test cases, 213
 transparency
 architecture, 105-106
 defined, 5
 development activities, 170-171
 flow of value/waste reduction,
 reinforcing, 6
 Scrum, 8
 self-managing teams, 11
 testing, 211
 transportation, 10
Tribal Leadership (Logan et al.), 244
 tuning performance, 159-162

U

UML, 117-118
 activity, 118
 artifacts, sharing, 120
 component/class sequence, 119
 extending, 122-124
 Model Explorer, 120
 use case, 118
 work item links, creating, 122
 unexpected behaviors, isolating, 154-155
 unit testing, 143
 automated code analysis, 152
 build verification tests, 149
 code metrics, calculating, 153
 continuous feedback, 269
 data, varying, 148
 existing code without tests, 145
 gaps, pinpointing, 147-148
 redundant code, 151-152
 TDD, 143-145
 unreasonable, 10

unwanted dependencies, 115
 use case diagrams, 118
 user experience (designs), 70
 user stories, 53-54
 Progress chart, 93
 Test Status chart, 97
*User Stories: For Agile Software
 Development*, 53

V

validating
 customers, 63-69
 definition of done
 build, automating, 181-185
 continuous integration (CI), 179-180
 dependency layers, 115
 value
 business
 problems, 47
 release planning, 51
 customer, 52-53
 clear goals, 52
 pain points, 52
 problems, 47
 release planning, 53
 user stories, 53
 vision statements, 52
 defining, 15
 flow
 defined, 5
 measures, 271
 post-2005 improvements, 256-258
 product backlog, 8-9
 testing, 209
 transparency/waste reduction,
 reinforcing, 6
 propositions, 251
Vasa, 48
 velocity, 83-84
 versioning, 162
 branching, 164
 benefits, 165
 by release, 165

- merging changes, 167
 - tracking changes, 167
 - viewing, 167
 - work isolation, 165
 - compilers, 163
 - database schema, 163
 - files, 163
 - support programs, 163
 - viability, 58
 - viewing
 - branches, 167
 - features progress, 254
 - unwanted dependencies, 115
 - virtual machines. *See* VMs
 - vision statements, 52
 - Visual Studio 2012, 262-263
 - Visualization and Modeling SDK, 122-124
 - VMs (virtual machines), test labs, 186
 - build deployment, automating, 192, 196
 - cloud, 198
 - multiple, 196
 - physical and virtual machines, 189, 192
 - SCVMM, 188
 - setting up, 187-189
 - standard, 188
 - support, 186
 - tests, running, 196
 - VS Developer Center website, 22
- W-Z**
- waiting, 10
 - waste
 - DevDiv, 245
 - eliminating
 - build failures, 202
 - code coverage/tests, monitoring, 203
 - done PBIs, 199
 - flow inefficiencies, detecting, 200
 - integrating frequently, 199
 - remaining work, tracking, 200-201
 - no repro (bugs), 218
 - DDAs, 215
 - evolving tests, 219-221
 - immersive app testing on
 - Windows 8, 221
 - solving, 218
 - project switching, 41
 - reduction
 - Bug Ping-Pong, 12-13
 - build automations, 12
 - defined, 5
 - flow of value/transparency, reinforcing, 6
 - Taiichi Ohno's Taxonomy of Waste, 9-11
 - testing, 210
 - Web performance tests, 227
 - creating, 225
 - running, 226
 - test data, varying, 227
 - websites
 - CloudShare, 198
 - unit testing plug-ins, 143
 - VS Developer Center, 22
 - Windows
 - immersive app testing, 221
 - shell, 169
 - wizards
 - Performance, 159-160
 - Project Creation, 21
 - Womack, Jim, 1
 - work
 - breakdown, 73
 - isolation, 165
 - items
 - checking in, 140
 - links, creating, 122
 - risks, 237
 - shared steps, 214
 - suspending, 140
 - working software, inspecting, 105