

Foreword by **Ken Schwaber**



# Agile Software Engineering with Visual Studio

From Concept to Continuous Feedback

Microsoft  
Visual Studio

Sam Guckenheimer  
Neno Loje

## Praise for *Agile Software Engineering with Visual Studio*

“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 *Agile Software Engineering with Visual Studio*, 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

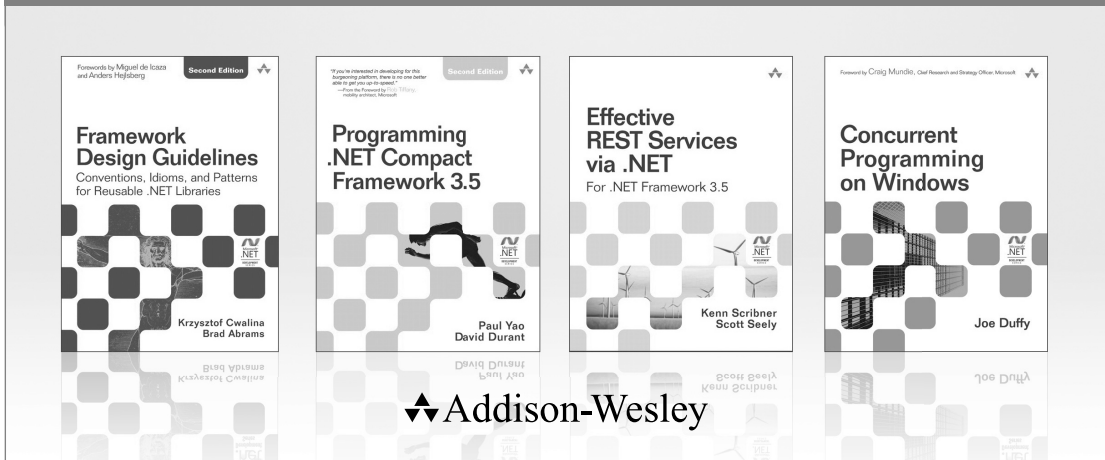
“Agile development practices are a core part of modern software development. Drawing from our own lessons in adopting agile practices at Microsoft, Sam Guckenheimer and Neno Loje not only outline the benefits, but also deliver a hands-on, practical guide to implementing those practices in teams of any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

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

# **Agile Software Engineering with Visual Studio**

## **From Concept to Continuous Feedback**

# Microsoft® .NET Development Series

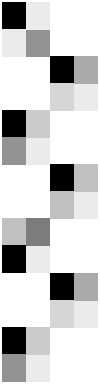


Visit [informit.com/msdotnetseries](http://informit.com/msdotnetseries) for a complete list of available products.

The award-winning **Microsoft .NET Development Series** was established in 2002 to provide professional developers with the most comprehensive, practical coverage of the latest .NET technologies. Authors in this series include Microsoft architects, MVPs, and other experts and leaders in the field of Microsoft development technologies. Each book provides developers with the vital information and critical insight they need to write highly effective applications.



---



# Agile Software Engineering with Visual Studio

---

From Concept 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  
Cape Town • 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 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@pearson.com

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

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

Copyright © 2012 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

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

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: 978-0-321-68585-8

ISBN-10: 0-321-68585-7

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

First printing September 2011

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

—*Sam*









# Contents

---

<i>Foreword</i>	<i>xvii</i>
<i>Preface</i>	<i>xix</i>
<i>Acknowledgements</i>	<i>xxvi</i>
<i>About the Authors</i>	<i>xxvii</i>
<b>1 The Agile Consensus</b>	<b>1</b>
The Origins of Agile	1
Agile Emerged to Handle Complexity	2
Empirical Process Models	4
A New Consensus	4
Scrum	6
<i>Potentially Shippable</i>	7
<i>Increasing the Flow of Value in Software</i>	8
<i>Reducing Waste in Software</i>	9
<i>Transparency</i>	11
<i>Technical Debt</i>	11
An Example	12
<i>Self-Managing Teams</i>	13
<i>Back to Basics</i>	15
Summary	15
End Notes	16

<b>2</b>	<b>Scrum, Agile Practices, and Visual Studio</b>	<b>19</b>
	Visual Studio and Process Enactment	20
	Process Templates	21
	<i>Teams</i>	22
	Process Cycles and TFS	23
	<i>Release</i>	24
	<i>Sprint</i>	26
	<i>Bottom-Up Cycles</i>	30
	<i>Personal Development Preparation</i>	30
	<i>Check-In</i>	30
	<i>Test Cycle</i>	31
	<i>Definition of Done at Every Cycle</i>	35
	Inspect and Adapt	36
	Task Boards	36
	Kanban	38
	Fit the Process to the Project	39
	<i>Geographic Distribution</i>	40
	<i>Tacit Knowledge or Required Documentation</i>	41
	<i>Governance, Risk Management, and Compliance</i>	41
	<i>One Project at a Time Versus Many Projects at Once</i>	41
	Summary	42
	End Notes	43
<b>3</b>	<b>Product Ownership</b>	<b>45</b>
	What Is Product Ownership?	46
	<i>The Business Value Problem: Peanut Butter</i>	47
	<i>The Customer Value Problem: Dead Parrots</i>	47
	<i>The Scope-Creep Problem: Ships That Sink</i>	48
	<i>The Perishable Requirements Problem: Ineffective Armor</i>	49
	Scrum Product Ownership	50
	Release Planning	51
	<i>Business Value</i>	52
	<i>Customer Value</i>	52
	<i>Exciters, Satisfiers, and Dissatisfiers: Kano Analysis</i>	55
	<i>Design Thinking</i>	58
	<i>Customer Validation</i>	62

Qualities of Service	63
<i>Security and Privacy</i>	64
<i>Performance</i>	64
<i>User Experience</i>	65
<i>Manageability</i>	66
How Many Levels of Requirements	67
<i>Work Breakdown</i>	68
Summary	70
End Notes	70
<b>4 Running the Sprint</b>	<b>73</b>
Empirical over Defined Process Control	75
Scrum Mastery	76
<i>Team Size</i>	77
<i>Rapid Estimation (Planning Poker)</i>	78
<i>A Contrasting Analogy</i>	80
Use Descriptive Rather Than Prescriptive Metrics	81
<i>Prevent Distortion</i>	84
<i>Avoid Broken Windows</i>	85
Answering Everyday Questions with Dashboards	86
<i>Burndown</i>	87
<i>Quality</i>	88
<i>Bugs</i>	90
<i>Test</i>	91
<i>Build</i>	93
Choosing and Customizing Dashboards	94
Using Microsoft Outlook to Manage the Sprint	95
Summary	96
End Notes	96
<b>5 Architecture</b>	<b>99</b>
Architecture in the Agile Consensus	100
<i>Inspect and Adapt: Emergent Architecture</i>	100
<i>Architecture and Transparency</i>	101
<i>Design for Maintainability</i>	102

Exploring Existing Architectures	103
<i>Understanding the Code</i>	103
<i>Maintaining Control</i>	109
<i>Understanding the Domain</i>	113
Summary	121
End Notes	123
<b>6 Development</b>	<b>125</b>
Development in the Agile Consensus	126
The Sprint Cycle	127
<i>Smells to Avoid in the Daily Cycle</i>	127
Keeping the Code Base Clean	128
<i>Catching Errors at Check-In</i>	128
<i>Shelving Instead of Checking In</i>	134
Detecting Programming Errors Early	135
<i>Test-Driven Development Provides Clarity</i>	135
<i>Catching Programming Errors with Code Reviews,</i> <i>Automated and Manual</i>	148
Catching Side Effects	152
<i>Isolating Unexpected Behavior</i>	152
<i>Isolating the Root Cause in Production</i>	155
<i>Tuning Performance</i>	156
Preventing Version Skew	160
<i>What to Version</i>	160
<i>Branching</i>	162
<i>Working on Different Versions in Parallel</i>	163
<i>Merging and Tracking Changes Across Branches</i>	165
<i>Working with Eclipse or the Windows Shell Directly</i>	167
Making Work Transparent	168
Summary	169
End Notes	171

<b>7</b>	<b>Build and Lab</b>	<b>173</b>
	Cycle Time	174
	Defining Done	175
	Continuous Integration	177
	Automating the Build	179
	<i>Daily Build</i>	180
	<i>BVTs</i>	181
	<i>Build Report</i>	181
	<i>Maintaining the Build Definitions</i>	183
	<i>Maintaining the Build Agents</i>	183
	Automating Deployment to Test Lab	185
	<i>Setting Up a Test Lab</i>	185
	<i>Does It Work in Production as Well as in the Lab?</i>	187
	<i>Automating Deployment and Test</i>	190
	Elimination of Waste	196
	<i>Get PBIs Done</i>	196
	<i>Integrate As Frequently As Possible</i>	197
	<i>Detecting Inefficiencies Within the Flow</i>	198
	Summary	201
	End Notes	202
<b>8</b>	<b>Test</b>	<b>203</b>
	Testing in the Agile Consensus	204
	<i>Testing and Flow of Value</i>	205
	<i>Inspect and Adapt: Exploratory Testing</i>	206
	<i>Testing and Reduction of Waste</i>	206
	<i>Testing and Transparency</i>	207
	Testing Product Backlog Items	207
	<i>The Most Important Tests First</i>	209
	Actionable Test Results and Bug Reports	212
	<i>No More “No Repro”</i>	214
	<i>Use Exploratory Testing to Avoid False Confidence</i>	216
	Handling Bugs	218
	Which Tests Should Be Automated?	219



Automating Scenario Tests	220
<i>Testing “Underneath the Browser” Using HTTP</i>	221
Load Tests, as Part of the Sprint	225
<i>Understanding the Output</i>	228
<i>Diagnosing the Performance Problem</i>	229
Production-Realistic Test Environments	230
Risk-Based Testing	232
<i>Capturing Risks as Work Items</i>	234
<i>Security Testing</i>	235
Summary	235
End Notes	236
<b>9 Lessons Learned at Microsoft Developer Division</b>	<b>239</b>
Scale	240
Business Background	241
<i>Culture</i>	241
<i>Waste</i>	243
<i>Debt Crisis</i>	244
Improvements After 2005	245
<i>Get Clean, Stay Clean</i>	245
<i>Tighter Timeboxes</i>	246
<i>Feature Crews</i>	246
<i>Defining Done</i>	246
<i>Product Backlog</i>	249
<i>Iteration Backlog</i>	251
<i>Engineering Principles</i>	254
Results	254
Law of Unintended Consequences	255
<i>Social Contracts Need Renewal</i>	255
<i>Lessons (Re)Learned</i>	256
<i>Celebrate Successes, but Don’t Declare Victory</i>	258
What’s Next?	259
End Notes	259

<b>10</b>	<b>Continuous Feedback</b>	<b>261</b>
	Agile Consensus in Action	262
	The Next Version	263
	Product Ownership and Stakeholder Engagement	264
	<i>Storyboarding</i>	264
	<i>Getting Feedback on Working Software</i>	265
	<i>Balancing Capacity</i>	267
	<i>Managing Work Visually</i>	268
	Staying in the Groove	270
	<i>Collaborating on Code</i>	272
	<i>Cleaning Up the Campground</i>	273
	Testing to Create Value	275
	TFS in the Cloud	275
	Conclusion	276
	<i>Living on the Edge of Chaos</i>	278
	End Notes	279
	<b>Index</b>	<b>281</b>







## Foreword

---

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, Microsoft's VS 2010. We are both indebted to Aaron Bjork of Microsoft, who developed the Scrum template that instantiates Scrum in VS 2010 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. Since 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 2010 (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 2010 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

---

Five years ago, we extended the world's leading product for individual developers, Microsoft Visual Studio, into Visual Studio Team System, and it quickly became the world's leading product for development teams. This addition of Application Lifecycle Management (ALM) to Visual Studio made life easier and more productive for hundreds of thousands of our users and tens of thousands of our Microsoft colleagues. In 2010, we shipped Visual Studio 2010 Premium, Ultimate, Test Professional, and Team Foundation Server. (We've dropped the Team System name.)

We've learned a lot from our customers in the past five years. Visual Studio 2010 is a huge release that enables a high-performance Agile software team to release higher-quality software more frequently. We set out to enable a broad set of scenarios for our customers. We systematically attacked major root causes of waste in the application lifecycle, 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 and 8, VS 2010 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 2010 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 2010 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, “Test,” 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 2010 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. Chapters 6 and 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 and 4 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 2010 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 Chapters 2 and 4.

**No more late surprises.** Agile teams, working iteratively and incrementally, often use burndown charts to assess their progress. Not only does VS 2010 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, “Running the Sprint,” 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 2010 makes it much easier to work with the existing code by introducing tools for architectural discovery. VS 2010 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 2010 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.

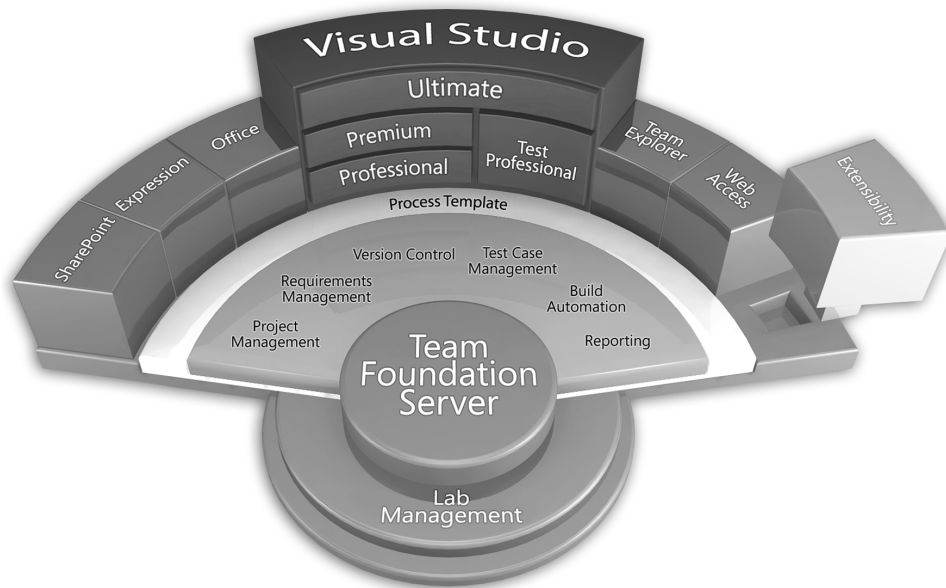
**No more technology silos.** More and more software projects use multiple technologies. In the past, teams often have had to choose different tools based on their runtime targets. As a consequence, .NET and Java teams have not been able to share data across their silos. Visual Studio Team Foundation Server 2010 integrates the two by offering clients in both the Visual Studio and Eclipse integrated development environments (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 2010. 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 2010. 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 2010 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 2010 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, now including Lab Management, forms the server of VS 2010. 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 standalone 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.

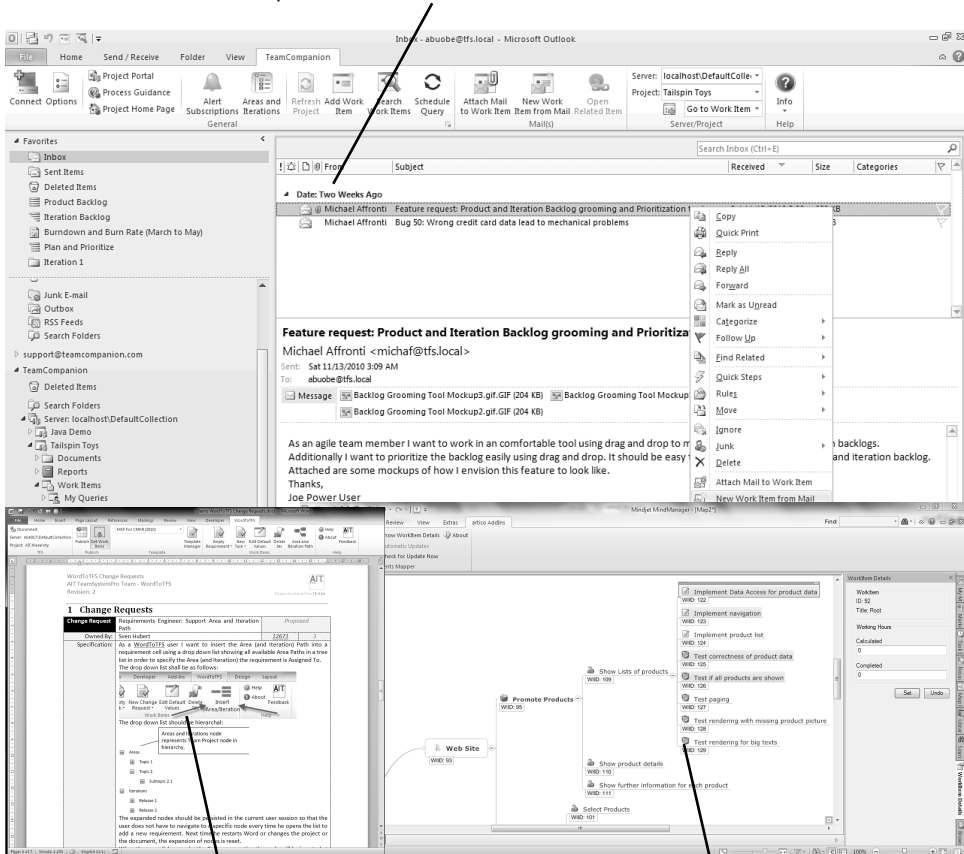
Visual Studio Premium adds the scenarios that are described in Chapter 6, “Development,” around working with the code. Visual Studio Test



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/](http://www.visualstudiowidgets.com/).

Ekobit TeamCompanion uses Microsoft Outlook to connect to TFS.



AIT WordtoTFS makes Microsoft Word a TFS client.

Artiso Requirements Mapper turns Mindjet MindManager into a TFS Client.

**FIGURE P-2:** A broad catalog of partner products extend TFS. Shown here are Artiso Requirements Mapper, Ekobit TeamCompanion, and AIT WordtoTFS.

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.

Unlike earlier versions, VS 2010 does not have role-based editions. This follows our belief in multidisciplinary, self-managing teams. We want to smooth the transitions and focus on the end-to-end flow. Of course, there's plenty more to learn about VS at the Developer Center of <http://msdn.microsoft.com/vstudio/>.



## 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. 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.



## ■ 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.<sup>1</sup>*

—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 “done-ness” 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.

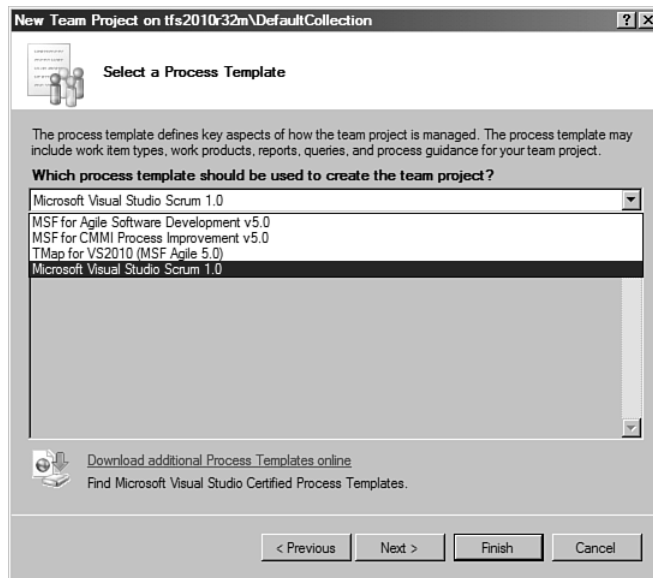


FIGURE 2-2: The Project Creation Wizard lets you create a team project based on any of the currently installed process templates.

Microsoft provides three process templates as standard:

1. **Scrum:** This process template directly supports Scrum, and was developed in collaboration with Ken Schwaber based on the *Scrum Guide*.<sup>2</sup> The Scrum process template defines work item types for Product Backlog Item, Bug, Task, Impediment, Test Case, Shared (Test) Steps, and Sprint. The reports are Release Burndown, Sprint Burndown, and Velocity.

2. **MSF for Agile Software Development:** MSF Agile is also built around a Scrum base but incorporates a broader set of artifacts than the Scrum process template. In MSF Agile, product backlog items are called *user stories* and impediments are called *issues*. The report shown in Figure 1.4 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.<sup>3</sup> 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 <http://msdn.microsoft.com/vstudio/>.

When you create a team project with VS 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  $7 \pm 2$  developers—in other words, 5 to 9 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 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.”<sup>4</sup> 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 Product Backlog Item (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.<sup>5</sup> 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 2-3. Scrum provides some simple rules for managing these.



Figure 2-4. 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.<sup>7</sup>

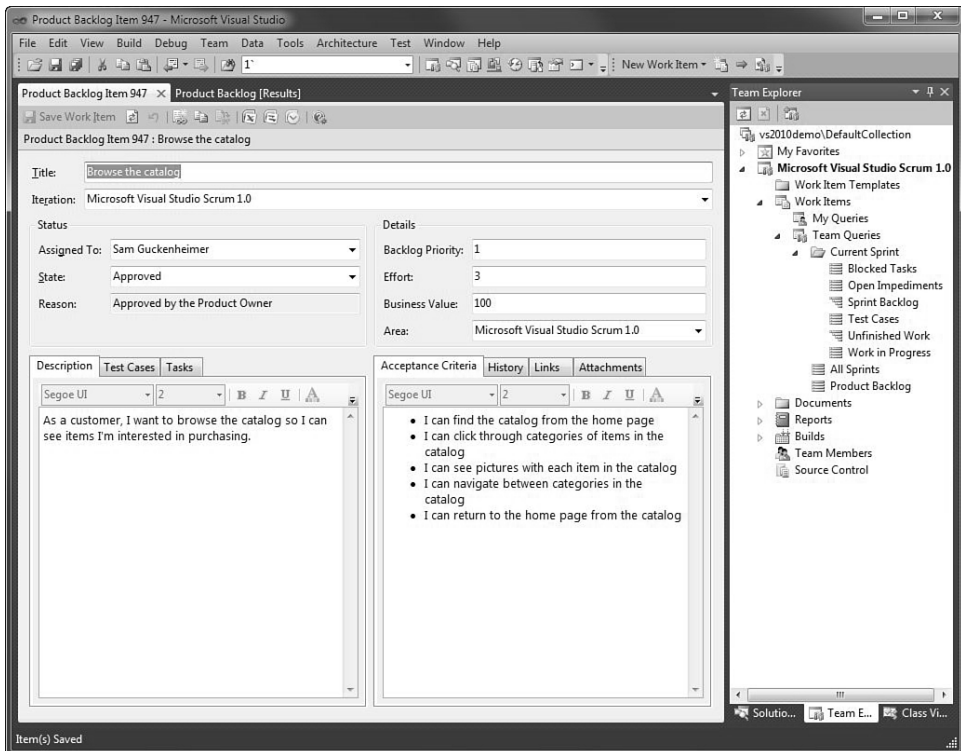


FIGURE 2-4: A product backlog item, shown here as accessed inside the VS IDE, can also be viewed from the Web Portal, Microsoft Excel, Microsoft Project, and many third-party plug-in tools available for TFS.

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 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.<sup>8</sup> 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 2-5).

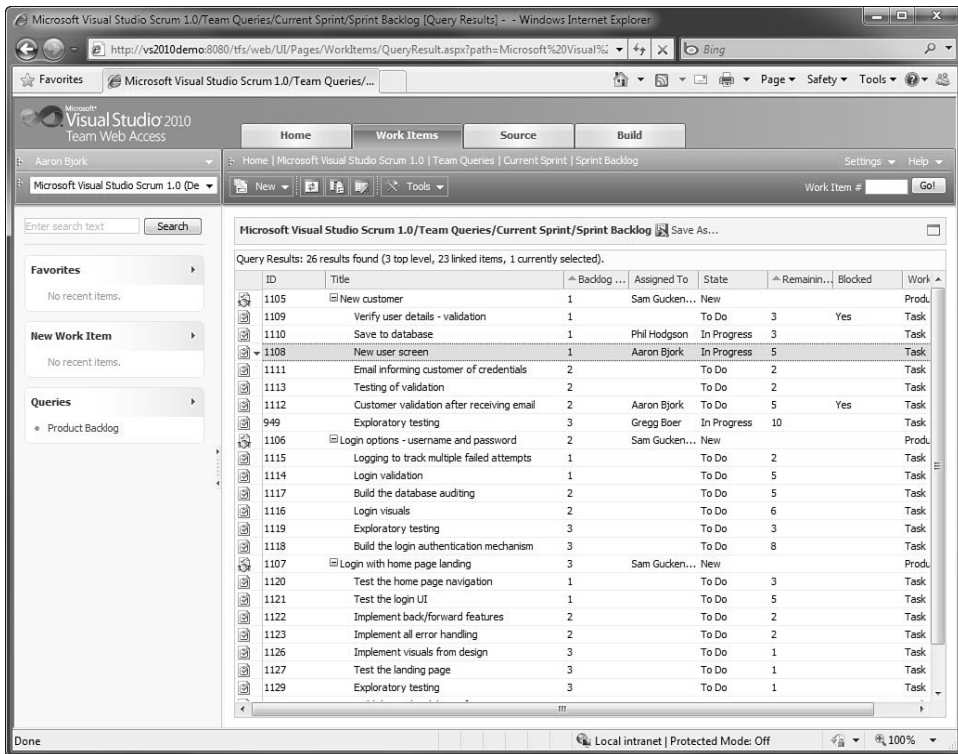


FIGURE 2-5: The sprint backlog, shown here as accessed from the Web Portal, consists of the tasks for the current sprint, derived from the PBIs to which the team has committed.

### ***Don't Confuse Product Backlog and Sprint Backlog***

In my 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.

### ***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 2-6).

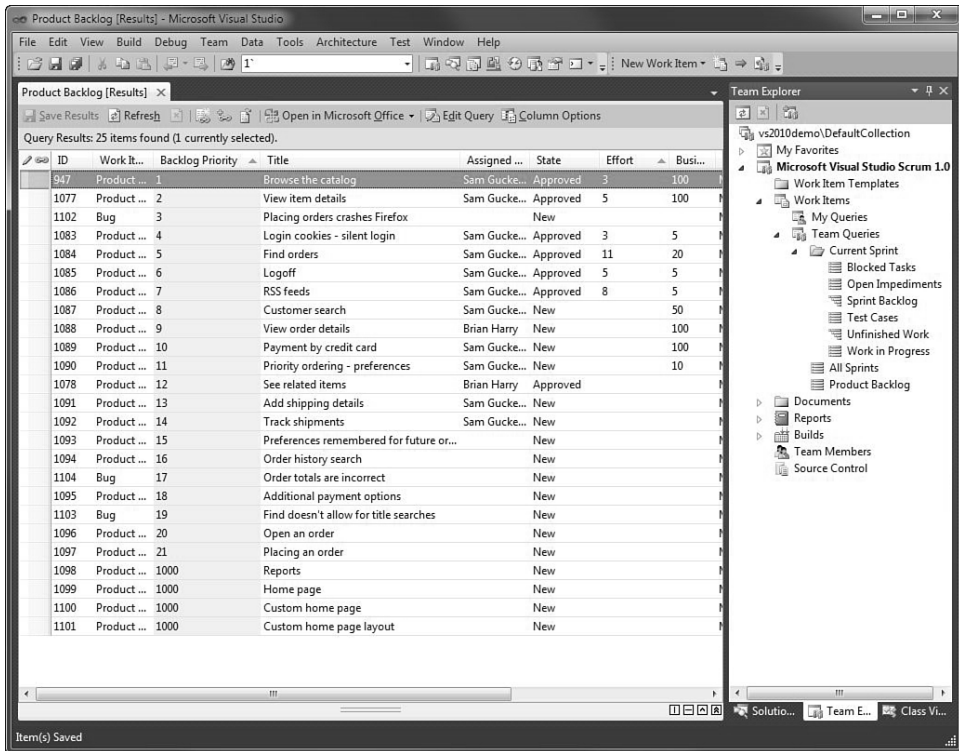


FIGURE 2-6: The product backlog contains the PBIs that express requirements and the bugs that are not handled in the current sprint. This can be accessed from any of the TFS clients; here it is shown in the VS IDE.

### ***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**

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.

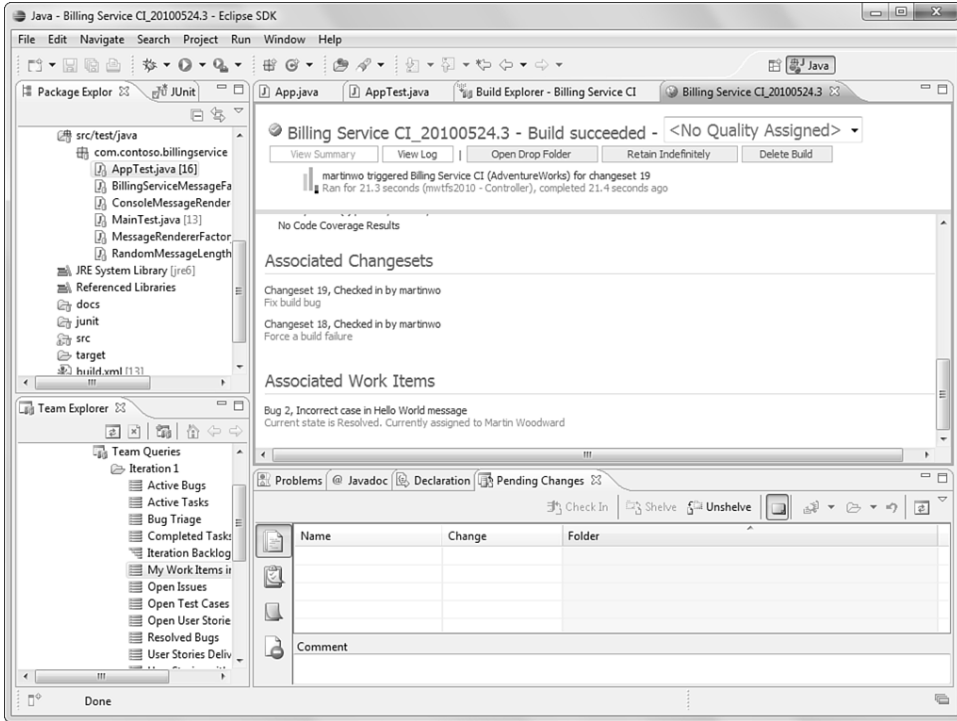
To avoid the frequent broken builds, TFS offers a form of continuous integration called *gated check-in*. Gated check-in extends the continuous integration workflow, in that it provisions a server and runs the team build *before* check-in. Only if the full build passes, *then* the server accepts the 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*. 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

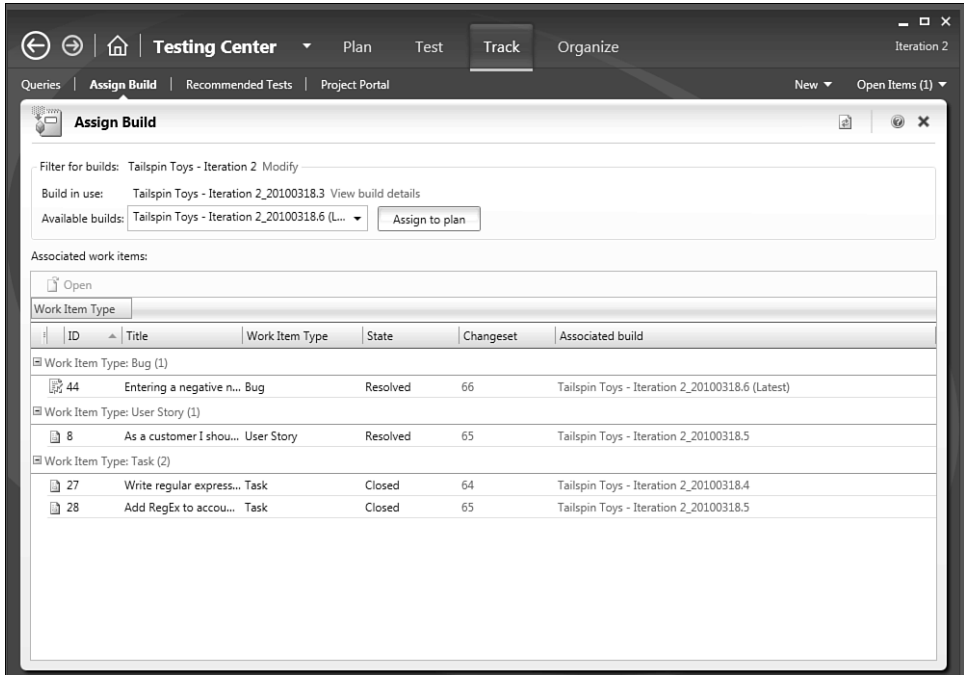
Completed PBIs need to be tested, as do bug fixes. Typically, team members check in code in small increments many times before completing a PBI. However, when a PBI is completed, a test cycle may start. In addition, many PBIs and bug fixes are often completed in rapid succession, and these can be combined 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 2-7.



**FIGURE 2-7:** 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 2-8. (See Chapter 8, “Test,” for more information.)



**FIGURE 2-8:** 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:

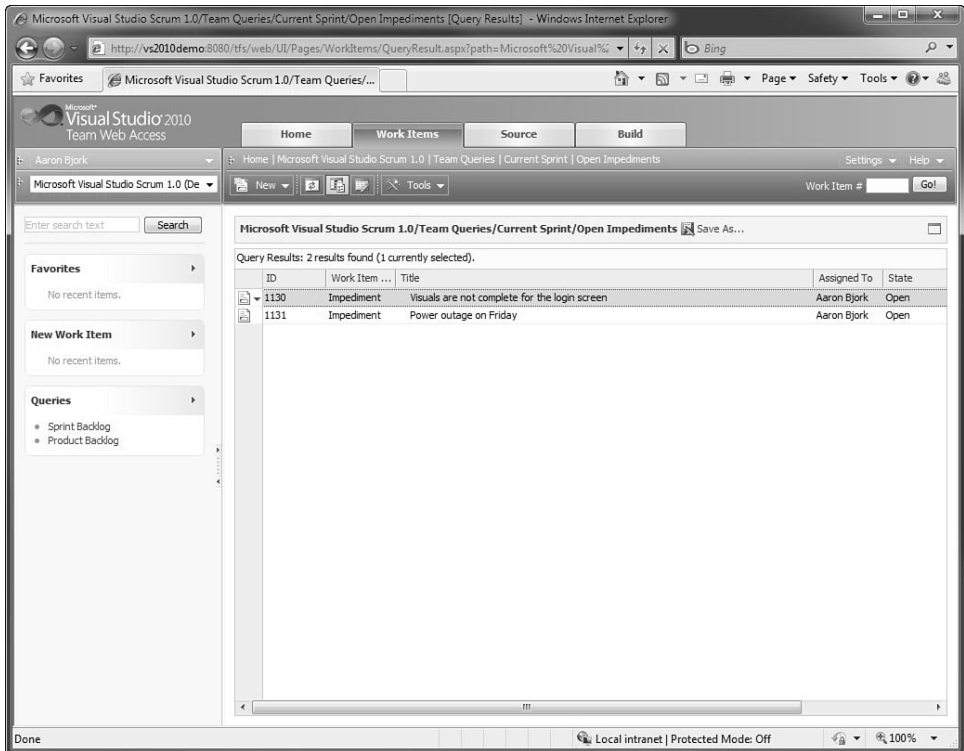
1. What has the team member accomplished since the last meeting?
2. What will the team member accomplish before the next meeting?
3. What obstacles currently impede the team member?

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 2-7 and 2-8 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 My Active Items query should align with question 2.
- The Open Impediments or Open Issues query, shown in Figure 2-9, should match question 3.

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 questions about whose data to trust.



**FIGURE 2-9:** The Open Impediments query shows the current state of blocking issues as of the daily scrum.

### 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?”<sup>9</sup> 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.

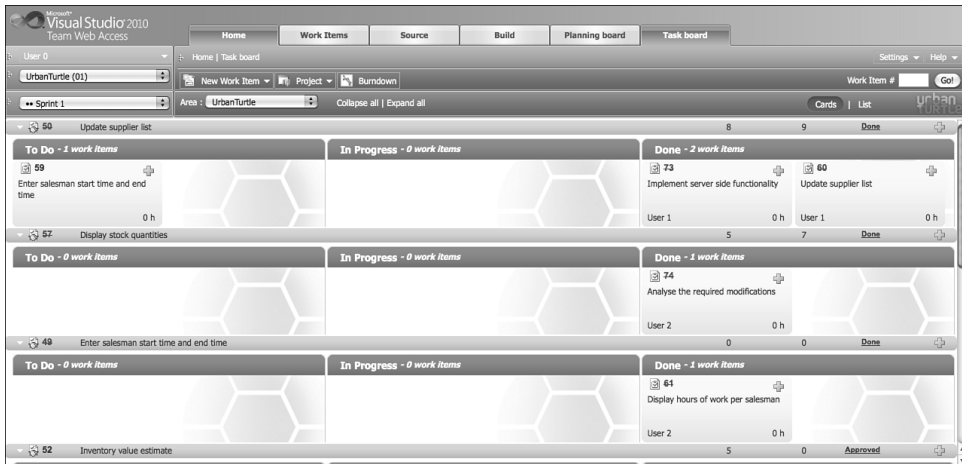
## Task Boards

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.

Scrum teams often visualize the tasks of the sprint backlog on the wall with a *task board*. Manual task boards use sticky notes, where rows group the tasks related to a particular PBI and columns show the progress of tasks from planned to in progress to done. As a task progresses, the task owner moves it along the board.

Several automated task boards currently visualize the sprint backlog of TFS, as shown in Figure 2-10. They provide 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.



**FIGURE 2-10:** Many TFS add-ins display the product and sprint backlogs as a task board. This add-in is called Urban Turtle and is available from <http://urbanturtle.com>

At Microsoft, we use these to coordinate Scrum teams across Redmond, Raleigh, Hyderabad, Shanghai, and many smaller sites. In Chapter 10, “Continuous Feedback,” you can see how we have productized our internal taskboards in the next version of TFS.

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 “signboards”) 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.<sup>10</sup> 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.<sup>11</sup> 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.

## Kanban

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.<sup>12</sup> 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.

Recent conferences have featured many experience reports comparing Scrum and Kanban. Kanban clearly works well where the team's workflow is relatively stable, the PBIs are fairly consistent, and the release vision well understood. For example, sustaining engineering projects often have a backlog of maintenance requests that are similarly sized and lend themselves well to Kanban.

In other words, in the Stacey terminology of Figure 1-1, Kanban works well for *complicated* or *simple* projects. The jury is out with regard to *complex* projects. Where higher degrees of uncertainty exist in the process, the explicit sprint cadence of Scrum can prove invaluable. In my experience, the concept of team commitment and the sprint rhythm are empowering to teams working in uncharted territory, the common ground of software development.

## 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.<sup>13</sup>

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 2-11).

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; tacit knowledge or required documentation; and governance, risk management, and compliance.

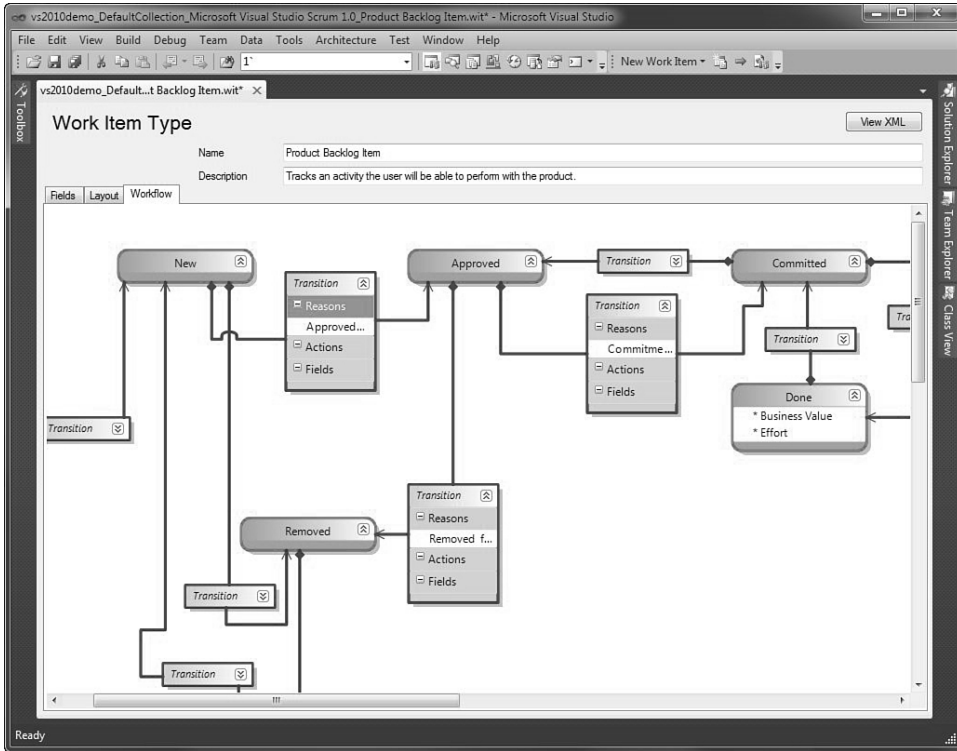


FIGURE 2-11: The Process Template Editor (in the TFS Power Tools on the VS Gallery) enables you to customize work item types, form design, and workflows.

## 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.<sup>14</sup>

**Table 2-1: Waste Caused by Project Switching**

<b>Number of Simultaneous Projects</b>	<b>Percent of Working Time Available per Project</b>	<b>Loss to Context Switching</b>
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 10, 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 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 lifespan. 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.

## End Notes

- <sup>1</sup> 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>.
- <sup>2</sup> Ken Schwaber and Jeff Sutherland, *Scrum Guide*, February 2010, available at [www.scrum.org/scrumguides/](http://www.scrum.org/scrumguides/).
- <sup>3</sup> [www.sei.cmu.edu](http://www.sei.cmu.edu)
- <sup>4</sup> Kent Beck with Cynthia Andres, *Extreme Programming Explained: Embrace Change, Second Edition* (Boston: Addison-Wesley, 2005), 34.
- <sup>5</sup> Mentioned in the *Scrum Guide*, and discussed in somewhat greater length in Ken Schwaber and Mike Beedle, *Agile Software Development with Scrum* (Prentice Hall, 2001), 25.



- <sup>6</sup> *Scrum Guide*, 9.
- <sup>7</sup> Tom DeMarco and Timothy Lister, *Waltzing with Bears: Managing Risk on Software Projects* (New York: Dorset House, 2003), 130.
- <sup>8</sup> Mike Cohn, *Agile Estimating and Planning* (Prentice Hall, 2005).
- <sup>9</sup> Cockburn, *op. cit.*
- <sup>10</sup> Ohno, *op. cit.*, 26.
- <sup>11</sup> 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>.
- <sup>12</sup> 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*.
- <sup>13</sup> Barry Boehm and Richard Turner, *Balancing Agility with Discipline: A Guide for the Perplexed* (Boston: Addison-Wesley, 2004), 152.
- <sup>14</sup> Gerald M. Weinberg, *Quality Software Management: Systems Thinking* (New York: Dorset House, 1992), 284.



# Index

---

## A

- acceptance testing. *See* testing
- accessibility, 65
- action logs, 13
- actionable test results, 212-213
- activity diagrams, 114
- advantages of Visual Studio 2010, xix-xxii
- Agile Alliance, 2
- Agile Consensus
  - advantages of, 2-3, 15
  - architecture, 100
    - dependency graphs, 103-106
    - diagram extensibility, 119-121
    - emergent architecture, 100-101
    - layer diagrams, 109-112
    - maintainability, 102-103
    - modeling projects, 113-119
    - sequence diagrams, 106-108
    - transparency, 101-102
  - builds
    - automated builds, 179-180
    - build agents, 183-185
    - build definitions, maintaining, 183
    - Build Quality Indicators report, 168-169
    - Build reports, 181-182
    - BVTs (build verification tests), 146-147, 181, 246
    - CI (continuous integration), 177-179
    - cycle time, 174-175
    - daily builds, 180
    - done*, definition of, 35, 80, 175-177
    - elimination of waste, 196-200
    - failures, 199-200
  - continuous feedback cycle
    - advantages of, 263
    - illustration, 262
  - descriptive metrics, 81-86
  - development. *See* development
  - empirical process control, 75-76
  - empirical process models, 4
  - flow, 8
  - multiple dimensions of project health, 86
  - origins of, 1-2
  - principles of, 4-6
  - product ownership. *See* product ownership
  - rapid estimation, 78-81
  - Scrum
    - explained, 6
    - potentially shippable increments, 7
    - product backlog, 8-9
    - reduction of waste, 9-13
    - technical debt, 11-12
    - transparency, 11
    - user stories, 8
  - Scrum mastery, 76-77
  - self-managing teams, 13-14
  - team size, 77
  - testing in, 204
    - exploratory testing, 206
    - flow of value, 205
    - reduction of waste, 206-207
    - transparency, 207
  - transparency, 5
- Agile Management for Software Engineering* (Anderson), 8
- analysis paralysis, 29-30
- Anderson, David J., 8, 38
- architecture, 100
  - dependency graphs, 103-106
  - diagram extensibility, 119-121
  - emergent architecture, 100-101
  - layer diagrams, 109-112
  - maintainability, 102-103
  - modeling projects, 113
    - activity diagrams, 114
    - class diagrams, 115-116
    - component diagrams, 115
    - Model Links, 117-119
    - sequence diagrams, 115

- UML Model Explorer, 116-117
    - use case diagrams, 114
  - sequence diagrams, 106-108
  - transparency, 101-102
  - attractiveness, 65
  - Austin, Robert, 81
  - automated builds, 179-180
  - automated deployment, 190-196
  - automated testing, 219-220
    - coded UI tests, 220-221
    - equivalence classes, 223-224
    - Web performance tests, 221-223
  - automatic code analysis, 148-149
  - availability, 66
- B**
- backlogs
    - iteration backlog, 251-253
    - product backlog, 8-9, 24
      - Microsoft Developer Division case study, 249-251
      - PBIs (product backlog items), 174-175, 196-197
      - problems solved by, 47-50
      - testing product backlog items, 207-211
    - sprint backlog, 27-28
  - balancing capacity, 267
  - baseless merges, 165-166
  - Beck, Kent, 10, 135, 203
  - Beizer, Boris, 216
  - Boehm, Barry, 26, 39
  - bottom-up cycles, 30
  - branching, 162-166
    - branch visualization, 248
    - branching by release, 163
  - Brandeis, Louis, 11
  - broken windows' effect, 85-86
  - Brooks, Frederick, 45-46
  - Brown, Tim, 58
  - Bug Ping-Pong, 12-13
  - bugs
    - debugging
      - Multi-Tier Analysis, 156
      - operational issues, 155-156
      - performance errors, 156-160
      - profiling, 156-160
      - with IntelliTrace, 152-154
    - handling, 28-29, 218-219
  - Bugs dashboard, 90-91
  - build check-in policy, 133-134
  - build process templates, 183
  - Build Quality Indicators report, 168-169
  - Build reports, 181-182
  - Build Success Over Time report, 200
  - build verification tests (BVTs), 146-147, 181, 246
  - builds. *See also* deployment
    - automated builds, 179-180
    - build agents, 183-185
    - build definitions, maintaining, 183
    - Build Quality Indicators report, 168-169
    - Build reports, 181-182
    - BVTs (build verification tests), 146-147, 181, 246
    - Builds dashboard, 93-94
    - Burndown dashboard, 87-88
    - business background (Microsoft Developer Division case study)
      - culture, 241-243
      - debt crisis, 244-245
      - waste, 243
    - business value problem, 47
    - butterfly effect, 279
    - BVTs (build verification tests), 146-147, 181, 246
- C**
- Capability Maturity Model Integration (CMMI), 22
  - capacity, balancing, 267
  - catching errors at check-in, 128-130
    - build check-in policy, 133-134
    - changesets, 129
    - check-in policies, 131-132
    - gated check-in, 132-134
    - shelving, 134-135
  - Change by Design* (Brown), 58
  - changesets, 129
  - chaos theory, 279
  - check-in policies, 30-31, 131-132
  - check-in, catching errors at, 128-130
    - build check-in policy, 133-134
    - changesets, 129
    - check-in policies, 30-31, 131-132
    - gated check-in, 132-134
    - shelving, 134-135
  - choosing dashboards, 94-95
  - CI (continuous integration), 177-179
  - class diagrams, 115-116
  - classic continuous integration (CI), 177
  - clones, finding, 273-274
  - cloud, TFS (Team Foundation Server) on, 275-276
  - CMMI (Capability Maturity Model Integration), 22
  - Cockburn, Alistair, 19
  - code coverage, 141-142
  - Code Review Requests, 272
  - code reviews
    - automatic code analysis, 148-149
    - manual code reviews, 151
  - code, collaborating on, 272

- coded UI tests, 220-221
- Cohn, Mike, 26, 54, 97
- collaborating on code, 272
- compatibility, 65
- component diagrams, 115
- concurrency, 65
- configuration testing, 187-190
- conformance to standards, 67
- continuous feedback cycle
  - advantages of, 263
  - illustration, 262
- continuous integration (CI), 177-179
- correction, 10
- Create Bug feature, 214-215
- crowds, wisdom of, 80
- CTPs (customer technical previews), 246
- culture (Microsoft Developer Division case study), 241-243
- cumulative flow diagrams, 198-199
- customer technical previews (CTPs), 246
- customer validation, 62-63
- customer value problem, 47-48
- customizing dashboards, 94-95
- cycle time, 174-175

**D**

- daily builds, 180
- daily cycle, 33-35
- daily stand-up meeting, 33, 77
- dashboards
  - Bugs, 90-91
  - Builds, 93-94
  - Burndown, 87-88
  - choosing, 94-95
  - customizing, 94-95
  - importance of, 86
  - Quality, 88, 90
  - Test, 91-93
- DDAs (diagnostic data adapters), 212
- debugging
  - with IntelliTrace, 152-154
  - Multi-Tier Analysis, 156
  - operational issues, 155-156
  - performance errors, 156-160
  - profiling, 156-160
- defined process control, 75
- defined process model, 2
- dependency graphs, 103-106
- deployment test labs
  - automating deployment and test, 190-196
  - configuration testing, 187-190
  - setting up, 185-186
- descriptive metrics, 81-86
- design thinking, 58-60
- desirability, 59
- detecting inefficiencies, 198-200
- DevDiv. *See* Microsoft Developer Division case study
- Developer Division. *See* Microsoft Developer Division case study

- development, 126
  - branching, 162-166
  - catching errors at check-in, 128-130
    - build check-in policy, 133-134
    - changesets, 129
    - check-in policies, 131-132
    - gated check-in, 132-134
    - shelving, 134-135
  - common problems, 127-128
  - debugging
    - IntelliTrace, 152-154
    - Multi-Tier Analysis, 156
    - operational issues, 155-156
    - performance errors, 156-160
    - profiling, 156-160
  - Eclipse Team Explorer Everywhere (TEE)
    - plug-in, 167
  - merging, 165
  - sprint cycle, 127
  - TDD (test-driven development)
    - advantages of, 136-138
    - BVTs (build verification tests), 146-147
    - code coverage, 141-142
    - code reviews, 148-151
    - explained, 135-136
    - generating tests for existing code, 138-140
    - Red-Green-Refactor, 136
    - test impact analysis, 143
    - variable data, 144-145
  - TFS Power Tools, 167-168
  - transparency, 168-169
  - versioning, 160-161
- DGML (Directed Graph Markup Language), 121
- diagnostic data adapters (DDAs), 212
- diagrams
  - activity diagrams, 114
  - class diagrams, 115-116
  - component diagrams, 115
  - cumulative flow diagrams, 198-199
  - extensibility, 119-121
  - layer diagrams, 109-112
  - Model Links, 117-119
  - sequence diagrams, 106-108, 115
  - use case diagrams, 114
- dimensions of project health, 86.
  - See also* dashboards
- Directed Graph Markup Language (DGML), 121
- discoverability, 65
- dissatisfiers, 55
- distortion, preventing, 84
- documentation, 41
- done*
  - definition of, 35, 80, 175-177
  - Microsoft Developer Division case study, 246-248

**E**

ease of use, 65  
 Eclipse Team Explorer Everywhere (TEE)  
   plug-in, 167  
 efficiency, 65  
 Ekobit TeamCompanion, 35  
 eliminating waste  
   detecting inefficiencies, 198-200  
   integrating code and tests, 197-198  
   PBIs (product backlog items), 196-197  
 emergent architecture, 100-101  
 empirical process control, 75-76  
 empirical process models, 4  
 enforcing permissions, 23  
 engineering principles (Microsoft Developer  
   Division case study), 254  
 epics, 54  
 equivalence classes, 223-224  
 errors, catching at check-in, 128-130  
   build check-in policy, 133-134  
   changesets, 129  
   check-in policies, 131-132  
   gated check-ins, 132-134  
   shelving, 134-135  
 excitors, 55  
 experiences (Microsoft Developer Division case  
   study), 250  
 exploratory testing, 206, 216-218, 275  
 extensibility (diagrams), 119-121  
 extra processing, 10

**F**

failures (build), 199-200  
 fault model, 233  
 fault tolerance, 65  
 feasibility, 59  
 feature crews (Microsoft Developer Division  
   case study), 246  
 features (Microsoft Developer Division case  
   study), 250  
 feedback  
   continuous feedback cycle  
     advantages of, 263  
     illustration, 262  
   in next version of VS product line, 265-267  
 feedback assistant, 265-267  
 Fibonacci sequence, 78, 261  
 15-minute daily scrum, 33, 77  
 finding clones, 273-274  
 fitting processes to projects, 39  
   documentation, 41  
   geographic distribution, 40  
   governance, risk management, and  
     compliance (GRC), 41  
   project switching, 41-42  
 flow, 8  
   cumulative flow diagrams, 198-199  
   flow of value, testing and, 205  
 forming-storming-norming-performing, 51  
 Franklin, Benjamin, 128, 239

full-motion video, 13  
 functionality, 75  
 FXCop, 148

**G**

Garlinghouse, Brad, 47  
 gated check-in (GC), 31, 132-134, 177-178, 248  
 General Motors (GM), 15  
 generating tests for existing code, 138-140  
 geographic distribution, 40  
 GM (General Motors), 15  
 granularity of requirements, 67-68  
 graphs, dependency, 103-106  
 GRC (governance, risk management, and  
   compliance), 41  
 Great Recession, impact on software  
   practices, 278

**H-I**

Haig, Al, 255  
 Howell, G., 73  
 inefficiencies, detecting, 198-200  
 installability, 66  
 integration  
   integrating code and tests, 197-198  
   Microsoft Developer Division case study,  
     247-248  
 IntelliTrace, 13, 152-154  
 interoperability, 67  
 interruptions, handling, 270-272  
 iron triangle, 75  
 isolation (Microsoft Developer Division case  
   study), 247-248  
 iteration backlog (Microsoft Developer Division  
   case study), 251-253

**K-L**

Kanban, 38  
 Kano analysis, 55-58  
 Kay, Alan C., 99, 261  
 Koskela, L., 73  
 Lab Management, xxiii  
 layer diagrams, 109-112  
 Lean, 1  
*Liber Abaci* (Fibonacci), 261  
 lightweight methods, 2  
 links, Model Links, 117-119  
 load modeling, 226  
 load testing  
   diagnosing performance problems with,  
     229-230  
   example, 226-228  
   explained, 225  
   load modeling, 226  
   output, 228-229  
 Logan, Dave, 242, 258  
 logs, 13

**M**

*The Machine That Changed the World*  
(Womack), 1

maintainability, 66  
     build agents, 183-185  
     build definitions, 183  
     designing for, 102-103  
 manageability, 66-67  
 managing work visually, 268-270  
 manual code reviews, 151  
 Martin, Bob, 273  
 McConnell, Steve, 75  
 Mean Time to Recover (MTTR), 66  
 mean time to repair (MTTR), 277  
 merging, 165  
 metrics, descriptive versus prescriptive, 81-86  
 Microsoft Developer Division case study, 240  
     culture, 241-243  
     debt crisis, 244-245  
     *done*, definition of, 246-248  
     engineering principles, 254  
     feature crews, 246  
     future plans, 259  
     integration and isolation, 247-248  
     iteration backlog, 251-253  
     management lesson, 258  
     MQ (milestone for quality), 245-246  
     product backlog, 249-251  
     results, 254-255  
     scale, 240  
     timeboxes, 246  
     unintended consequences, 255-258  
     waste, 243  
 Microsoft Outlook, managing sprints from, 95  
 Microsoft Test Manager. *See* MTM (Microsoft Test Manager)  
 milestone for quality (MQ), 245-246  
 Model Explorer, 116-117  
 Model Links, 117-119  
 modeling projects, 113  
     activity diagrams, 114  
     class diagrams, 115-116  
     component diagrams, 115  
     Model Links, 117-119  
     sequence diagrams, 115  
     UML Model Explorer, 116-117  
     use case diagrams, 114  
 Moles, 139  
 monitorability, 66  
 Moore, Geoffrey, 52  
 motion, 10  
 MQ (milestone for quality), 245-246  
 MSF Agile process template, 22  
 MSF for CMMI Process Improvement process template, 22  
 MTM (Microsoft Test Manager), 32, 207-211  
     actionable test results, 212-213  
     Create Bug feature, 214-215  
     DDAs (diagnostic data adapters), 212  
     exploratory testing, 216-218

query-based suites, 209  
 Recommended Tests, 209-210  
 Shared Steps, 211  
 test data, 211  
 test plans, 209  
 test settings, 213  
 test steps, 211  
 test suites, 209

MTTR (Mean Time to Recover), 66

MTTR (mean time to repair), 277

*muda*, 9-10

Multi-Tier Analysis, 156

multiple dimensions of project health, 86.

*See also* dashboards

*mura*, 9-10

*muri*, 9-10

must-haves, 55

**N-O**

negative testing, 206

Newton's Cradle, 125

"No Repro" results, eliminating, 214-215

Ohno, Taiichi, 10, 37

operability, 66

operational issues, 155-156

Outlook, managing sprints from, 95

overburden, 10

overproduction, 10

**P**

pain points, 53

paper prototyping, 59

PBIs (product backlog items), 24, 174-175,  
196-197

Peanut Butter Manifesto, 47

peanut buttering, 249

performance, 64-65

    performance problems, diagnosing,  
    229-230

    tuning, 156-160

    Web performance tests, 221-222

perishable requirements problem, 49-50

permissions, enforcing, 23

personal development preparation, 30

personas, 52

pesticide paradox, 216

*The Pet Shoppe*, 47

Pex, 139

planning

    Planning Poker, 78-80

    releases, 51-54

        business value, 52

        customer value, 52-53

        pain points, 53

        scale, 54

        user story form, 54

        vision statements, 53

    sprints, 77

Planning Poker, 78-81

- PMBOK (Project Management Body of Knowledge), 3, 69
  - policies
    - build check-in policy, 133-134
    - check-in policies, 31, 131-132
  - Poppendieck, Tom, 9
  - portability, 67
  - potentially shippable increments, 7, 26, 126
  - PowerPoint, 62
  - PreFAST, 148
  - prescriptive metrics, 81-86
  - preventing distortion, 84
  - privacy, 64
  - process cycles, 23-24
    - bottom-up cycles, 30
    - check-in, 30-31
    - daily cycle, 33-35
    - definition of *done* at every cycle, 35
    - personal development preparation, 30
    - releases, 24-26
    - sprints
      - avoiding analysis paralysis, 29-30
      - explained, 26-27
      - handling bugs, 28-29
      - retrospectives, 36
      - reviews, 36
      - sprint backlogs, 27-28
    - test cycle, 31-32
  - process enactment, 20
  - process models
    - defined process model, 2
    - empirical process models, 4
  - process templates, 21-22
  - processes, fitting to projects, 39
    - documentation, 41
    - geographic distribution, 40
    - governance, risk management, and compliance (GRC), 41
    - project switching, 41-42
  - product backlog, 8-9
    - Microsoft Developer Division case study, 249-251
      - experiences, 250
      - features, 250
      - scenarios, 250
    - PBIs (product backlog items), 24, 174-175, 196-197
    - problems solved by
      - business value problem, 47
      - customer value problem, 47-48
      - perishable requirements problem, 49-50
      - scope creep problem, 48
      - testing product backlog items, 207-211
  - Product Owners, 22. *See also* product ownership
  - product ownership, 256
    - customer validation, 62-63
    - design thinking, 58-60
    - explained, 46-47, 50
    - granularity of requirements, 67-68
    - in next version of VS product line
      - balancing capacity, 267
      - feedback assistant, 265-267
      - storyboarding, 264
      - taskboard visualization, 268-270
  - Kano analysis, 55-58
  - qualities of service (QoS), 63-64
    - manageability, 66-67
    - performance, 64-65
    - security and privacy, 64
    - user experience, 65
  - release planning, 51-54
    - business value, 52
    - customer value, 52-53
    - pain points, 53
    - scale, 54
    - user story form, 54
    - vision statements, 53
  - storyboarding, 60-62
    - work breakdown, 68-70
  - production-realistic test environments, 230-231
  - profiling, 156-160
  - Project Creation Wizard, 21
  - Project Management Body of Knowledge (PMBOK), 3, 69
  - project switching, 41-42
  - projects, fitting processes to, 39
    - documentation, 41
    - geographic distribution, 40
    - governance, risk management, and compliance (GRC), 41
    - project switching, 41-42
- ## Q-R
- QoS (qualities of service), 63-64
    - manageability, 66-67
    - performance, 64-65
    - security and privacy, 64
    - user experience, 65
  - quality, 75
  - Quality dashboard, 88-90
  - quality gates, 247
  - quantities, comparison of, 79
  - query-based suites, 209
  - Quick Cluster, 106
  - rapid cognition, 79
  - Rapid Development* (McConnell), 75
  - rapid estimation, 78-81
  - Reagan, Ronald, 255
  - Recommended Tests, 209-210
  - recoverability, 66
  - Red-Green-Refactor, 136
  - reduction of waste, 9-10
    - Bug Ping-Pong, 12-13
    - Taiichi Ohno's taxonomy of waste, 10
    - testing and, 206-207

- releases
    - explained, 23-26
    - planning, 51-54
      - business value, 52
      - customer value, 52-53
      - pain points, 53
      - scale, 54
      - user story form, 54
      - vision statements, 53
  - reliability, 66
  - reporting, 231-232
    - Build Quality Indicators report, 168-169
    - Build reports, 181-182
    - Build Success Over Time report, 200
  - resources, 75
  - responsiveness, 65
  - results (Microsoft Developer Division case study), 254-255
  - retrospectives (sprint), 36, 77
  - reviews (sprint), 36, 77
  - Ries, Eric, 173
  - risk-based testing, 232-235
  - roles
    - Product Owner, 22
      - customer validation, 62-63
      - design thinking, 58-60
      - explained, 46-50
      - granularity of requirements, 67-68
      - Kano analysis, 55-58
      - qualities of service (QoS), 63-67
      - release planning, 51-54
      - storyboarding, 60-62
      - work breakdown, 68-70
    - Scrum Master, 22
    - Team of Developers, 22
  - Romer, Paul, 1
- ## S
- SaaS (software as a service), 275
  - satisfiers, 55
  - scalability, 65
  - scale, 54, 240
  - scenarios, 250
  - Schema Compare, 161
  - Schwaber, Ken, 3, 15, 24, 97
  - scope creep, 48
  - screenshots, 13
  - Scrum
    - daily cycle, 33-35
    - explained, 6
    - Planning Poker, 78-80
    - potentially shippable increments, 7, 126
    - product backlog, 8-9
    - product ownership
      - customer validation, 62-63
      - design thinking, 58-60
      - explained, 46-47, 50
      - granularity of requirements, 67-68
      - Kano analysis, 55-58
      - qualities of service (QoS), 63-67
      - release planning, 51-54
      - storyboarding, 60-62
      - work breakdown, 68-70
    - reduction of waste, 9-10
      - Bug Ping-Pong, 12-13
      - Taiichi Ohno's taxonomy of waste, 10
      - testing and, 206-207
    - releases, 23-26, 51-54
    - Scrum Guide*, 24, 77
    - Scrum mastery, 76-77
    - sprints, 23, 30
      - avoiding analysis paralysis, 29-30
      - definition of, 77
      - explained, 26-27
      - handling bugs, 28-29
      - planning, 77
      - retrospectives, 36, 77
      - reviews, 36, 77
      - sprint backlogs, 27-28
    - task boards, 36-38
    - team size, 77
    - teams, 22-23
    - technical debt, 11-12
    - transparency, 11
      - user stories, 8
    - Scrum Guide*, 24, 77
    - Scrum Master, 22
    - Scrum process template, 21
    - security, 64
    - security testing, 235
    - self-managing teams, 13-14
    - sequence diagrams, 106-108, 115
    - serviceability, 67
    - Shared Steps, 211, 220
    - shelving, 134-135
    - size of teams, 77
    - Sketchflow, 62
    - software as a service (SaaS), 275
    - software under test (SUT), 219
    - sprint backlogs, 27-28
    - sprints, 23, 30
      - avoiding analysis paralysis, 29-30
      - definition of, 77
      - done*, 80
      - explained, 26-27
      - handling bugs, 28-29
      - managing
        - with dashboards. *See* dashboards
        - with Microsoft Outlook, 95
      - planning, 77
      - Planning Poker, 78-80
      - retrospectives, 36, 77
      - reviews, 36, 77
      - sprint backlogs, 27-28
      - sprint cycle, 127
    - Stacey Matrix, 3
    - Stacey, Ralph D., 3
    - standards, conformance to, 67
    - static code analysis, 148-149
    - story points, 78



- story-point estimation, 78-80
  - storyboarding, 60-62, 264
  - Strategic Management and Organisational Dynamics* (Stacey), 3
  - stubs, 139
  - SUT (software under test), 219
  - Sutherland, Jeff, 24
  - system configurations, 13
- T**
- task boards, 36-38
  - taskboard visualization, 268-270
  - TDD (test-driven development)
    - advantages of, 136-138
    - BVTs (build verification tests), 146-147
    - code coverage, 141-142
    - code reviews
      - automatic code analysis, 148-149
      - manual code reviews, 151
    - explained, 135-136
    - generating tests for existing code, 138-140
    - Red-Green-Refactor, 136
    - test impact analysis, 143
    - variable data, 144-145
  - Team Explorer, xxiii
  - Team Explorer Everywhere (TEE), xxiii, 167
  - Team Foundation Server. *See* TFS
  - Team of Developers, 22
  - team projects, 20
  - TeamCompanion, 95
  - teams, 22-23
    - self-managing teams, 13-14
    - size of, 77
  - technical debt, 11-12, 244-245
  - TEE (Team Explorer Everywhere), xxiii, 167
  - templates, process templates, 21-22
  - Test dashboard, 91-93
  - test-driven development. *See* TDD
  - test impact analysis, 143
  - Test Management Approach (TMap), 22
  - testability, 67
  - testing. *See also* debugging
    - in Agile Consensus, 204
      - exploratory testing, 206
      - flow of value, 205
      - reduction of waste, 206-207
      - transparency, 207
    - automated testing, 219-220
      - coded UI tests, 220-221
      - equivalence classes, 223-224
      - Web performance tests, 221-223
    - exploratory testing, 275
    - fault model, 233
    - integrating code and tests, 197-198
    - load testing
      - diagnosing performance problems
        - with, 229-230
      - example, 226-228
      - explained, 225
      - load modeling, 226
      - output, 228-229
- MTM (Microsoft Test Manager), 207-211
  - actionable test results, 212-213
  - Create Bug feature, 214-215
  - DDAs (diagnostic data adapters), 212
  - exploratory testing, 216-218
  - query-based suites, 209
  - Recommended Tests, 209-210
  - Shared Steps, 211
  - test data, 211
  - test plans, 209
  - test settings, 213
  - test steps, 211
  - test suites, 209
- negative testing, 206
- production-realistic test environments, 230-231
- reporting, 231-232
- risk-based testing, 232-235
- security testing, 235
- SUT (software under test), 219
- TDD (test-driven development)
  - advantages of, 136-138
  - BVTs (build verification tests), 146-147
  - code coverage, 141-142
  - code reviews, 148-151
  - explained, 135-136
  - generating tests for existing code, 138-140
  - Red-Green-Refactor, 136
  - test impact analysis, 143
  - variable data, 144-145
- test automation, 257
- test category, 146
- test configurations, 188
- test cycle, 31-32
- test data, 211
- test labs
  - automating deployment and test, 190-196
  - configuration testing, 187-190
  - setting up, 185-186
- test plans, 209
- test settings, 213
- test steps, 211
- test suites, 209
- TFS (Team Foundation Server), 20
  - explained, xxiii-xxv
  - fitting processes to projects, 39
    - documentation, 41
    - geographic distribution, 40
    - governance, risk management, and compliance (GRC), 41
    - project switching, 41-42
  - Power Tools, 167-168
  - process cycles, 23
    - bottom-up cycles, 30
    - check-in, 30-31

- daily cycle, 33-35
- definition of *done* at every cycle, 35
- personal development
  - preparation, 30
  - releases, 24-26
  - sprints, 26-30
  - test cycle, 31-32
- on Windows Azure, 275-276
- themes, 54
- time, 75
- timeboxes (Microsoft Developer Division case study), 246
- TMap (Test Management Approach), 22
- tours, 206
- Toyota, 1, 14, 37
- transparency, 5, 11
  - architecture, 101-102
  - development, 168-169
  - testing and, 207
- Tribal Leadership* (Logan et al), 242
- tuning performance, 156-160
- Turner, Richard, 39

## U

- UML
  - activity diagrams, 114
  - class diagrams, 115-116
  - component diagrams, 115
  - Model Explorer, 116-117
  - sequence diagrams, 115
  - use case diagrams, 114
- “The Underlying Theory of Project Management Is Obsolete” (Koskela and Howell), 73
- uninstallability, 66
- unintended consequences (Microsoft Developer Division case study), 255-258
- unreasonableness, 10
- use case diagrams, 114
- user experience, 65
- user stories, 8
- User Stories Applied: For Agile Software Development* (Cohn), 54
- user story form, 54

## V

- validation, customer, 62-63
- variable data, 144-145
- Vasa*, 48
- velocity, 80
- version skew, preventing
  - branching, 162-166
  - merging, 165
  - versioning, 160-161
- versioning, 160-161
- viability, 59
- virtual machine snapshots, 13
- vision statements, 53

- Visual Studio Premium, xxiii
- Visual Studio Test Professional, xxiii
- Visual Studio. *See* VS
- visualization, taskboard, 268-270
- vNext (next version of VS product line), 263
  - balancing capacity, 267
  - clones, finding, 273-274
  - Code Review Requests, 272
  - exploratory testing, 275
  - feedback assistant, 265-267
  - impact on flow of value, 276-278
  - interruptions, handling, 270-272
  - storyboarding, 264
  - taskboard visualization, 268-270
  - Team Foundation Server on Windows Azure, 275-276
- VS (Visual Studio)
  - process enactment, 20
  - process templates, 21-22
- vNext (next version of VS product line), 263
  - balancing capacity, 267
  - clones, finding, 273-274
  - Code Review Requests, 272
  - exploratory testing, 275
  - feedback assistant, 265-267
  - impact on flow of value, 276-278
  - interruptions, handling, 270, 272
  - storyboarding, 264
  - taskboard visualization, 268-270
  - Team Foundation Server on Windows Azure, 275-276

## W-X-Y-Z

- waiting, 10
- waste
  - eliminating
    - detecting inefficiencies, 198-200
    - integrating code and tests, 197-198
    - PBIs (product backlog items), 196-197
  - Microsoft Developer Division case study, 243
  - reducing, 9-10
    - Bug Ping-Pong, 12-13
    - Taiichi Ohno’s taxonomy of waste, 10
    - testing and, 206-207
- Web performance tests, 221-223
- Weinberg, Gerald, 41
- Wideband Delphi Method, 26
- Windows Azure, TFS (Team Foundation Server) on, 275-276
- WIP (work-in-progress) limits, 38
- wizards, Project Creation Wizard, 21
- Womack, Jim, 1, 15
- work breakdown, 68-70
- work item types, 21
- work-in-progress (WIP) limits, 38
- world readiness, 65