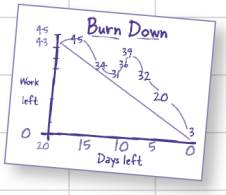
A Brain-Friendly Guide

Head First Software Development

Learn the real user story of how Mary satisfied her customers

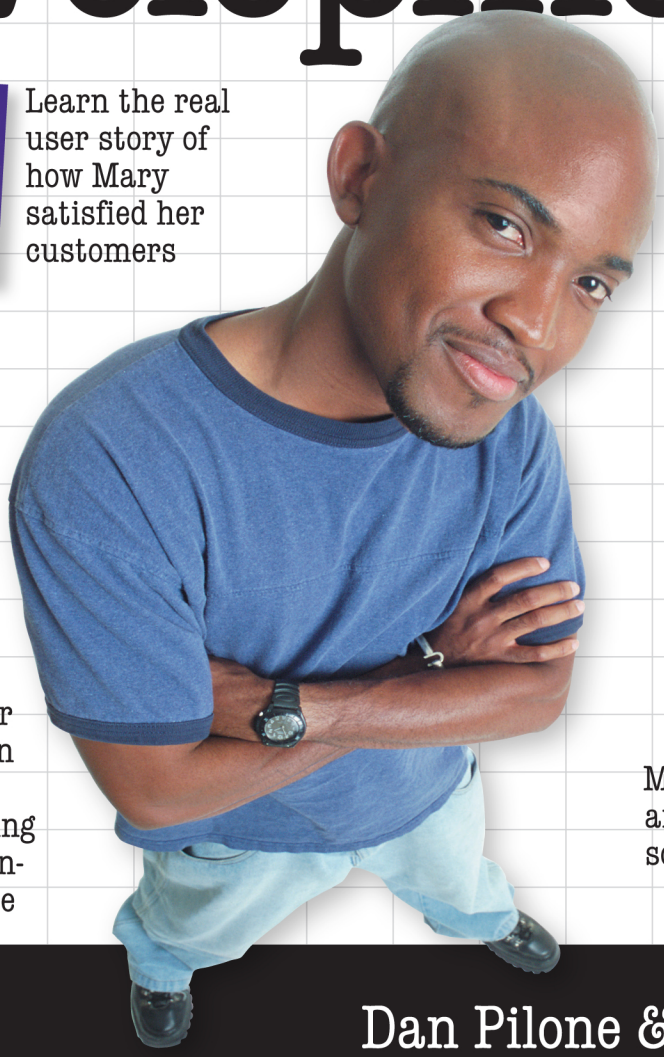
Score big by using velocity to figure out how fast your team can produce

Use test-driven development to avoid unsightly software disasters



Keep your project on schedule by tracking your burn-down rate

Master the techniques and tools of seasoned software developers



O'REILLY®

Dan Pilone & Russ Miles

Head First Software Development

by Dan Pilone and Russ Miles

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 (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators: Kathy Sierra, Bert Bates

Series Editor: Brett D. McLaughlin

Design Editor: Louise Barr

Cover Designers: Louise Barr, Steve Fehler

Production Editor: Sanders Kleinfeld

Indexer: Julie Hawks

Page Viewers: Vinny, Nick, Tracey, and Corinne

Printing History:

December 2007: First Edition.

No sleepovers were conducted in the writing of this book, although one author did purportedly get engaged using his prototype of the iSwoon application. And one pig apparently lost its nose, but we're confident that had nothing to do with the software development techniques espoused by this text.

RepKover™ This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 0-596-52735-7

ISBN-13: 978-0-596-52735-8

[M]

Table of Contents (Summary)

	Intro	xxv
1	great software development: *Pleasing your customer*	1
2	gathering requirements: *Knowing what the customer wants*	29
3	project planning: *Planning for success*	69
4	user stories and tasks: *Getting to the real work*	109
5	good-enough design: *Getting it done with great design*	149
6	version control: *Defensive development*	177
6.5	building your code: *Insert tab a into slot b...*	219
7	testing and continuous integration: *Things fall apart*	235
8	test-driven development: *Holding your code accountable*	275
9	ending an iteration: *It's all coming together...*	317
10	the next iteration: *If it ain't broke... you still better fix it*	349
11	bugs: *Squashing bugs like a pro*	383
12	the real world: *Having a process in life*	417

Table of Contents (the real thing)

Intro

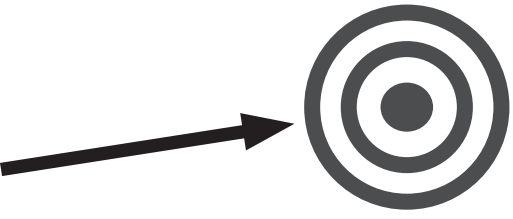
Your brain on Software Development. You're sitting around trying to *learn* something, but your *brain* keeps telling you all that learning *isn't important*. Your brain's saying, "Better leave room for more important things, like which wild animals to avoid and whether naked rock-climbing is a bad idea." So how *do* you trick your brain into thinking that your life really depends on learning how to develop great software?

Who is this book for?	xxvi
We know what you're thinking	xxvii
Metacognition	xxix
Bend your brain into submission	xxxi
Read me	xxxii
The technical review team	xxxiv
Acknowledgments	xxxv

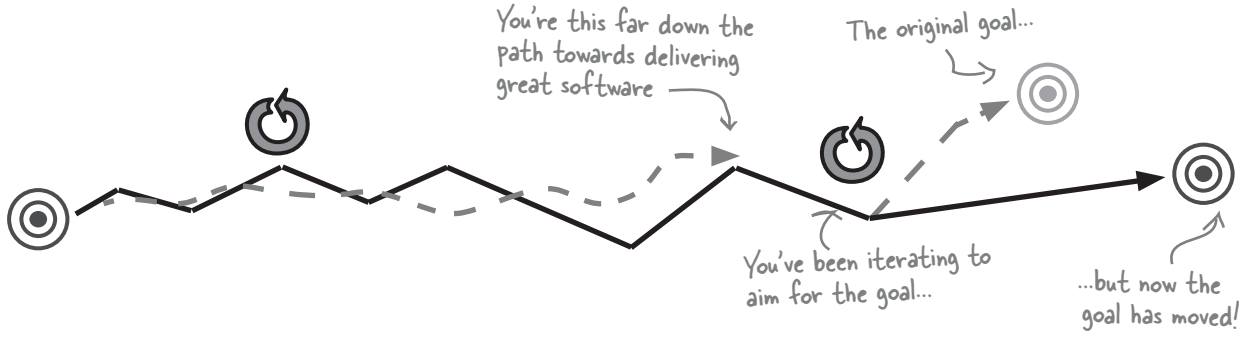
1 great software development
Pleasing your customer

If the customer's unhappy, *everyone's* unhappy!

Every great piece of software starts with a customer's big idea. It's your job as a professional software developer to **bring those ideas to life**. But taking a vague idea and turning it into working code—code that **satisfies your customer**—isn't so easy. In this chapter you'll learn how to avoid being a software development casualty by delivering software that is **needed**, **on-time,** and **on-budget**. Grab your laptop and let's set out on the road to shipping great software.


The Goal

Tom's Trails is going online	2
Most projects have two major concerns	3
The Big Bang approach to development	4
Flash forward: two weeks later	5
Big bang development usually ends up in a big MESS	6
Great software development is...	9
Getting to the goal with ITERATION	10
Each iteration is a mini-project	14
Each iteration is QUALITY software	14
The customer WILL change things up	20
It's up to you to make adjustments	20
But there are some BIG problems...	20
Iteration handles change automatically (well sort of)	22
Your software isn't complete until it's been RELEASED	25
Tools for your Software Development Toolbox	26

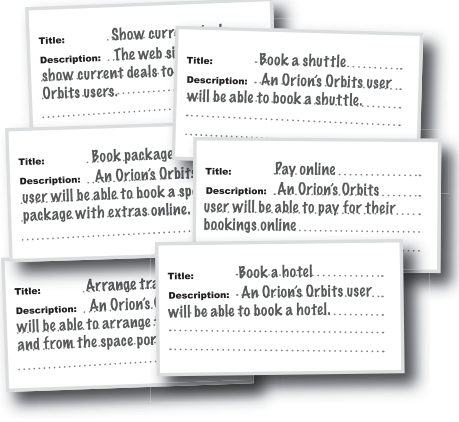


2 gathering requirements

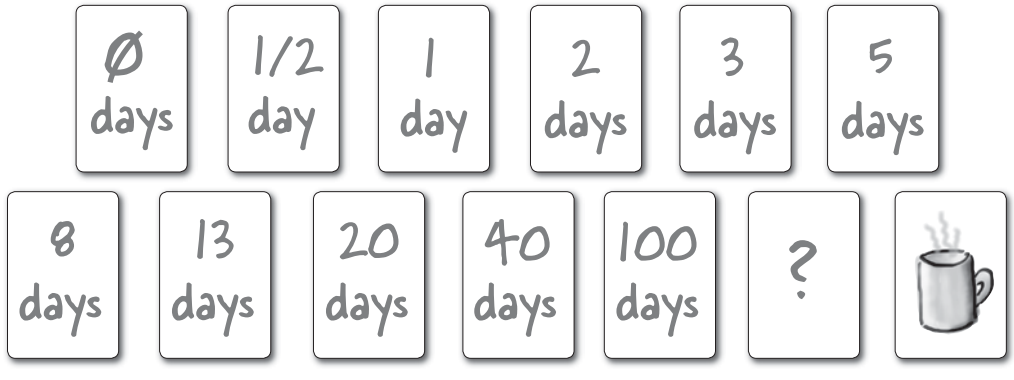
Knowing what the customer wants

You can't always get what you want... but the customer better!

Great software development delivers **what the customer wants**. This chapter is all about **talking to the customer** to figure out what their **requirements** are for your software. You'll learn how **user stories**, **brainstorming**, and the **estimation game** help you get inside your customer's head. That way, by the time you finish your project, you'll be confident you've built what your customer wants... and not just a poor imitation.



Orion's Orbits is modernizing	30
Talk to your customer to get MORE information	33
Bluesky with your customer	34
Sometimes your bluesky session looks like this...	36
Find out what people REALLY do	37
Your requirements must be CUSTOMER-oriented	39
Develop your requirements with customer feedback	41
User stories define the WHAT of your project... estimates define the WHEN	43
Cubicle conversation	47
Playing Planning Poker	48
Put assumptions on trial for their lives	51
A BIG user story estimate is a BAD user story estimate	54
The goal is convergence	57
The requirement to estimate iteration cycle	60
Finally, we're ready to estimate the whole project	



3 project planning

Planning for success

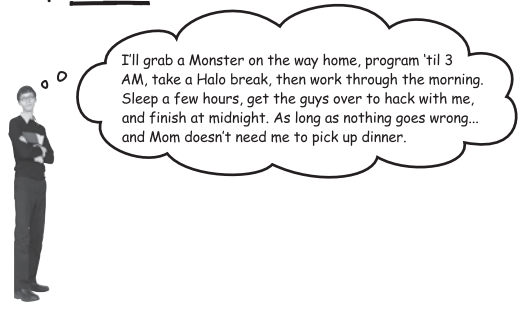
Every great piece of software starts with a great plan.

In this chapter you're going to learn how to create that plan. You're going to learn how to work with the customer to **prioritize their requirements**. You'll **define iterations** that you and your team can then work towards. Finally you'll create an achievable **development plan** that you and your team can confidently **execute** and **monitor.** By the time you're done, you'll know exactly how to get from requirements to milestone 1.0.

Customers want their software NOW!	70
Prioritize with the customer	73
We know what's in Milestone 1.0 (well, maybe)	74
If the features don't fit, re-prioritize	75
More people sometimes means diminishing returns	77
Work your way to a reasonable milestone 1.0	78
Iterations should be short and sweet	85
Comparing your plan to reality	87
Velocity accounts for overhead in your estimates	89
Programmers think in UTOPIAN days...	90
Developers think in REAL-WORLD days...	91
When is your iteration too long?	92
Deal with velocity BEFORE you break into iterations	93
Time to make an evaluation	97
Managing ~~pissed off~~ customers	98
The Big Board on your wall	100
How to ruin your team's lives	103

Here's what a programmer SAYS...

...but here's what he's really THINKING



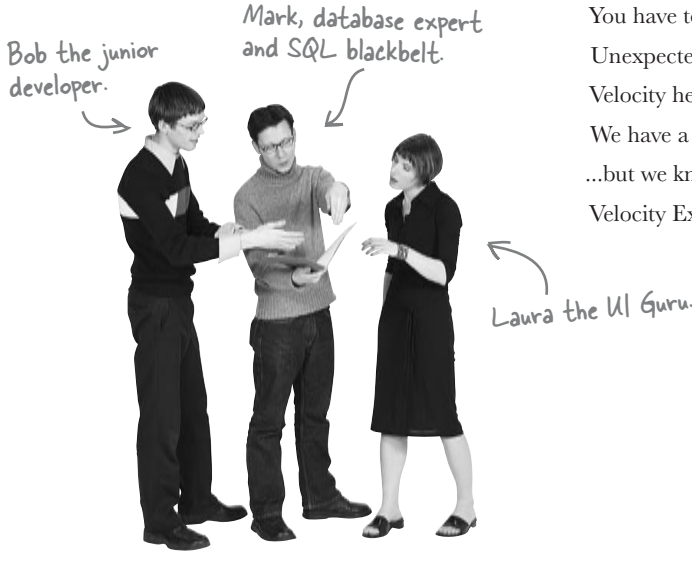
4 user stories and tasks

Getting to the real work

It's time to go to work. User stories captured what you need to develop, but now it's time to knuckle down and **dish out the work that needs to be done** so that you can bring those user stories to life. In this chapter you'll learn how to **break your user stories into tasks,** and how your **task estimates** help you track your project from inception to completion. You'll learn how to update your board, moving tasks from in-progress, to complete, to finally **completing an entire user story**. Along the way, you'll handle and prioritize the inevitable **unexpected work** your customer will add to your plate.

Introducing iSwoon	110
Do your tasks add up?	113
Plot just the work you have left	115
Add your tasks to your board	116
Start working on your tasks	118
A task is only in progress when it's IN PROGRESS	119
What if I'm working on two things at once?	120
Your first standup meeting...	123
Task 1: Create the Date class	124
Standup meeting: Day 5, end of Week 1...	130
Standup meeting: Day 2, Week 2...	136
We interrupt this chapter...	140
You have to track unplanned tasks	141
Unexpected tasks raise your burn-down rate	143
Velocity helps, but...	144
We have a lot to do...	146
...but we know EXACTLY where we stand	147
Velocity Exposed	148

Your first standup meeting...

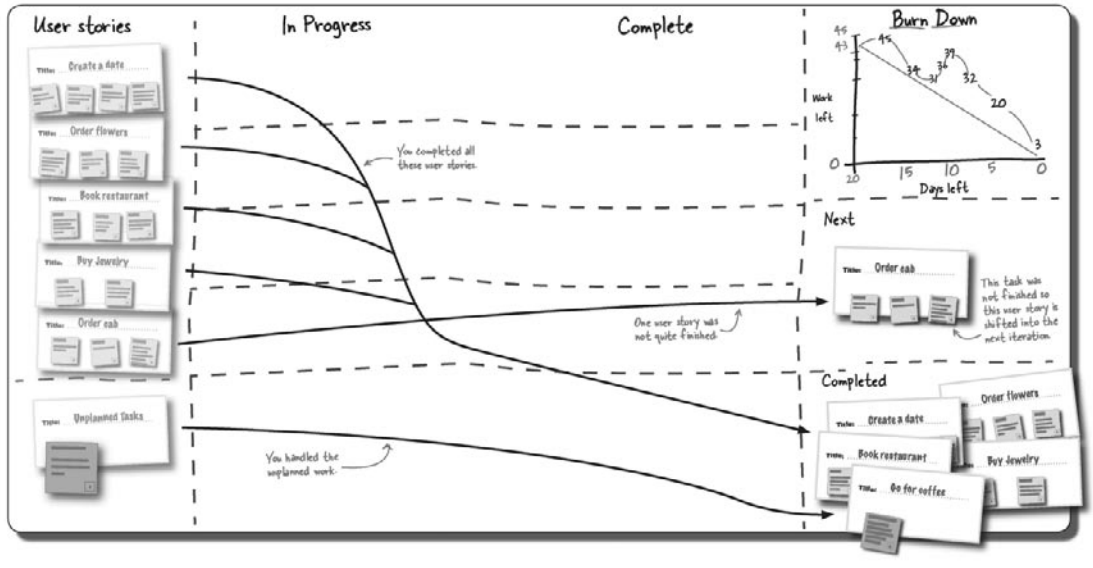


5 good-enough design

Getting it done with great design

Good design helps you deliver. In the last chapter things were looking pretty dire. A **bad design** was making life **hard for everyone** and, to make matters worse, an unplanned task cropped up. In this chapter you'll see how to **refactor** your design so that you and your team can be **more productive**. You'll apply **principles of good design**, while at the same time being wary of striving for the promise of the ***'perfect design'***. Finally you'll **handle unplanned tasks** in exactly the same way you handle all the other work on your project using the big project board on your wall.

iSwoon is in serious trouble...	150
This design breaks the single responsibility principle	153
Spotting multiple responsibilies in your design	156
Going from multiple responsibilies to a single responsibility	159
Your design should obey the SRP, but also be DRY...	160
The post-refactoring standup meeting...	164
Unplanned tasks are still just tasks	166
Part of your task is the demo itself	167
When everything's complete, the iteration's done	170

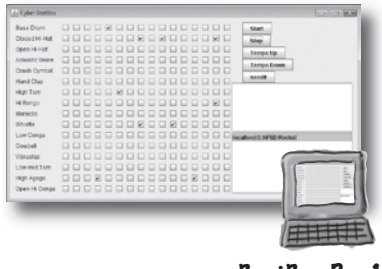


version control

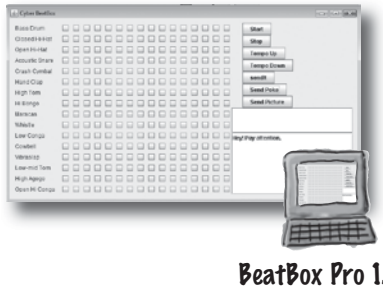
6 Defensive development

When it comes to writing great software, *Safety First!*

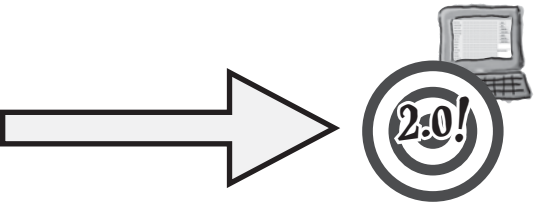
Writing great software isn't easy... especially when you've got to make sure your code works, and **make sure it keeps working**. All it takes is one typo, one bad decision from a co-worker, one crashed hard drive, and suddenly all your work goes down the drain. But with **version control**, you can make sure your **code is always safe** in a code repository, you can **undo mistakes**, and you can make **bug fixes**—to new and old versions of your software.



BeatBox Pro 1.0



BeatBox Pro 1.x



You've got a new contract—BeatBox Pro	178
And now the GUI work...	182
Demo the new BeatBox for the customer	185
Let's start with VERSION CONTROL	188
First set up your project...	190
...then you can check code in and out.	191
Most version control tools will try and solve problems for you	192
The server tries to MERGE your changes	193
If your software can't merge the changes, it issues a conflict	194
More iterations, more stories...	198
We have more than one version of our software...	200
Good commit messages make finding older software easier	202
Now you can check out Version 1.0	203
(Emergency) standup meeting	204
Tag your versions	205
Tags, branches, and trunks, oh my!	207
Fixing Version 1.0...for real this time.	208
We have TWO code bases now	209
When NOT to branch...	212
The Zen of good branching	212
What version control does...	214
Version control can't make sure you code actually works...	215
Tools for your Software Development Toolbox	216

6½

building your code

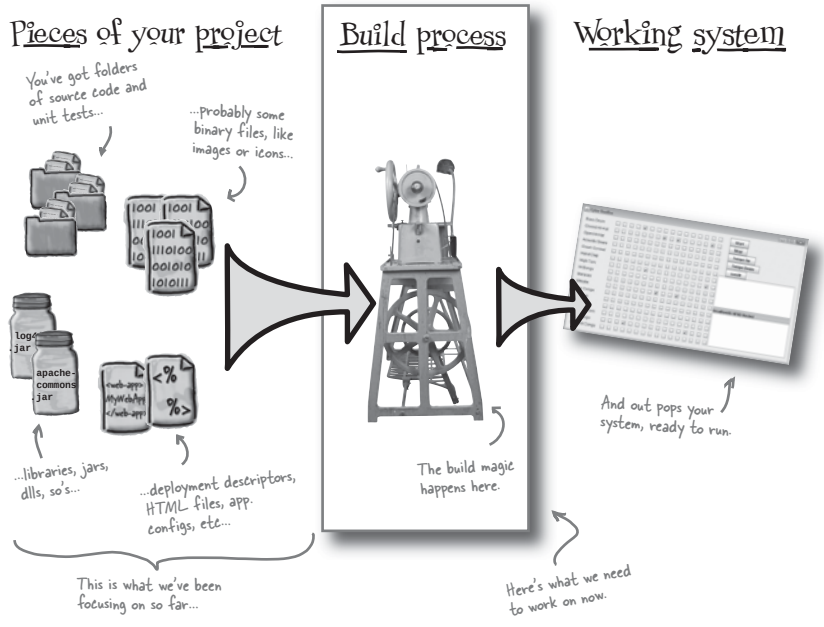
Insert tab a into slot b...

It pays to follow the instructions...

...especially when you write them yourself.

It's not enough to use configuration management to ensure your code stays safe. You've also got to worry about **compiling your code** and packaging it into a deployable unit. On top of all that, which class should be the main class of your application? How should that class be run? In this chapter, you'll learn how a **build tool** allows you to **write your own instructions** for dealing with your source code.

Developers aren't mind readers	220
Building your project in one step	221
Ant: a build tool for Java projects	222
Projects, properties, targets, tasks	223
Good build scripts...	228
Good build scripts go BEYOND the basics	230
Your build script is code, too	232
New developer, take two	233
Tools for your Software Development Toolbox	234



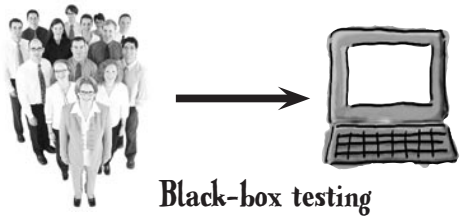
7 testing and continuous integration

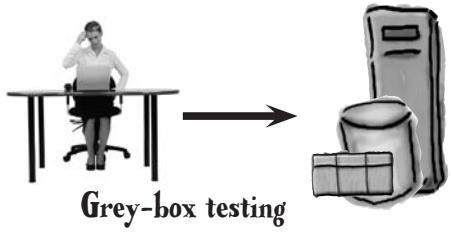
Things fall apart

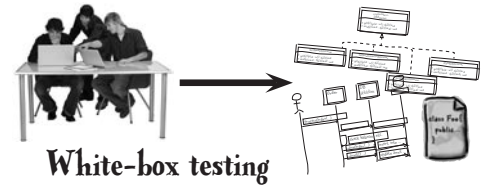
Sometimes even the best developer breaks the build.

Everyone's done it at least once. You're sure **your code compiles**, you've tested it over and over again on your machine and committed it into the repository. But somewhere between your machine and that black box they call a server *someone* must have changed your code. The unlucky soul who does the next checkout is about to have a bad morning sorting out **what used to be working code**. In this chapter we'll talk about how to put together a **safety net** to keep the build in working order and you **productive**.

Things will ALWAYS go wrong...	236
There are three ways to look at your system...	238
Black-box testing focuses on INPUT and OUTPUT	239
Grey-box testing gets you CLOSER to the code	240
White-box testing uses inside knowledge	243
Testing EVERYTHING with one step	248
Automate your tests with a testing framework	250
Use your framework to run your tests	251
At the wheel of CI with CruiseControl	254
Testing guarantees things will work... right?	256
Testing all your code means testing EVERY BRANCH	264
Use a coverage report to see what's covered	265
Getting good coverage isn't always easy...	267
What CM does...	270
Tools for your Software Development Toolbox	274





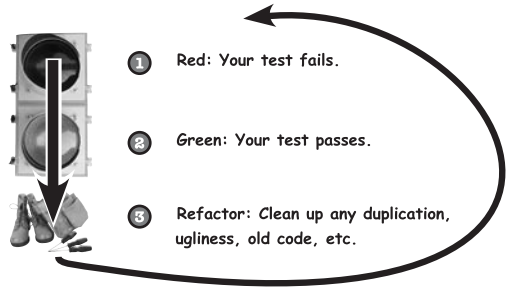


8 test-driven development

Holding your code accountable

Sometimes it's all about setting expectations. Good code needs to work, everyone knows that. But how do **you know your code works**? Even with unit testing, there are still parts of most code that goes untested. But what if testing was a **fundamental part of software development**? What if you did ***everything*** with testing in mind? In this chapter, you'll take what you know about version control, CI, and automated testing and tie it all together into an environment where you can feel **confident** about **fixing bugs**, **refactoring**, and even **reimplementing** parts of your system.

Test FIRST, not last 276
So we're going to test FIRST... 277
Welcome to test-driven development 277
Your first test... 278
...fails miserably. 279
Get your tests to GREEN 280
Red, green, refactor... 281
In TDD, tests DRIVE your implementation 286
Completing a task means you've got all the tests you need, and they all pass 288
When your tests pass, move on! 289
Simplicity means avoiding dependencies 293
Always write testable code 294
When things get hard to test, examine your design 295
The strategy pattern provides formultiple implementations of a single interface 296
Keep your test code with your tests 299
Testing produces better code 300
More tests always means lots more code 302
Strategy patterns, loose couplings, object stand ins... 303
We need lots of different, but similar, objects 304
What if we generated objects? 304
A mock object stands in for real objects 305
Mock objects are working object stand-ins 306
Good software is testable... 309
It's not easy bein' green... 310
A day in the life of a test-driven developer... 312
Tools for your Software Development Toolbox 314

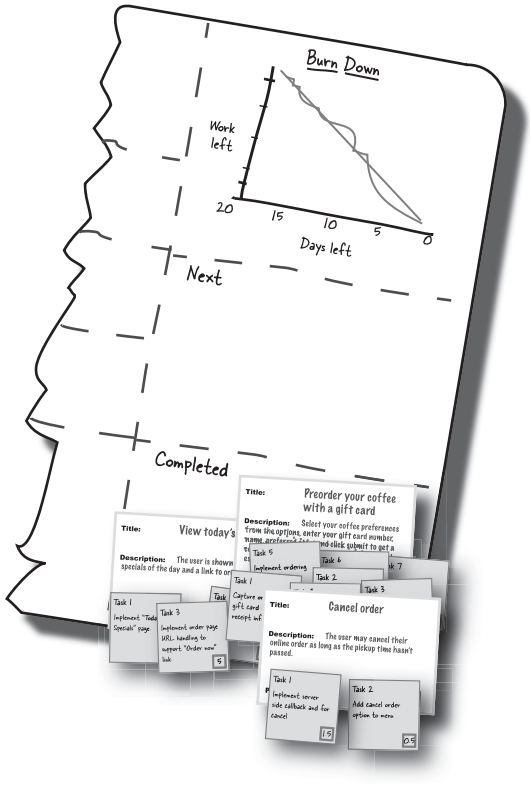


9 ending an iteration

It's all coming together...

You're almost finished! The team's been working hard and things are wrapping up. Your tasks and user stories are **complete**, but what's the best way to spend that extra day you ended up with? Where does **user testing** fit in? Can you squeeze in one more round of **refactoring** and **redesign**? And there sure are a lot of lingering **bugs**... when do those get fixed? It's all part of **the end of an iteration**... so let's get started on getting finished.

Your iteration is just about complete... 318
...but there's lots left you could do 319
System testing MUST be done... 324
...but WHO does system testing? 325
System testing depends on a complete system to test 326
Good system testing requires TWO iteration cycles 327
More iterations means more problems 328
Top 10 Traits of Effective System Testing 333
The life (and death) of a bug 334
So you found a bug.... 336
Anatomy of a bug report 337
But there's still plenty left you COULD do... 338
Time for the iteration review 342
Some iteration review questions 343
A GENERAL priority list for getting EXTRA things done... 344
Tools for your Software Development Toolbox 346



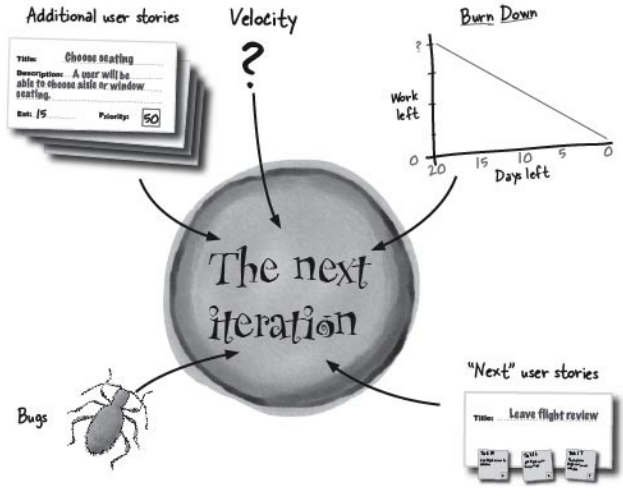
10 the next iteration

If it ain't broke...you still better fix it

Think things are going well?
Hold on, that just might change...

Your iteration went great, and you're delivering working software on-time. Time for the next iteration? No problem, right? Unfortunately, not right at all. Software development is all about **change,** and **moving to your next iteration** is no exception. In this chapter you'll learn how to prepare for the ***next*** iteration. You've got to **rebuild your board** and **adjust your stories** and expecations based on what the customer wants ***NOW***, not a month ago.

What is working software? 350
You need to plan for the next iteration 352
Velocity accounts for... the REAL WORLD 359
And it's STILL about the customer 360
Someone else's software is STILL just software 362
Customer approval? Check! 365
Testing your code 370
Houston, we really do have a problem... 371
Trust NO ONE 373
It doesn't matter who wrote the code.
If it's in YOUR software, it's YOUR responsibility. 373
You without your process 378
You with your process 379



11 bugs

Squashing bugs like a pro

Your code, your responsibility…your bug, your reputation!

When things get tough, it's **up to you** to bring them back from the brink. **Bugs**, whether they're in your code or just in code that your software uses, are a **fact of life** in software development. And, like everything else, the way you handle bugs should fit into the rest of your process. You'll need to **prepare your board**, **keep your customer in the loop**, **confidently estimate** the work it will take to fix your bugs, and apply **refactoring** and **prefactoring** to fix and avoid bugs in the future.

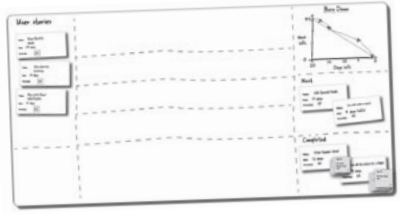
Previously on Iteration 2	386
First, you've got to talk to the customer	386
Priority one: get things buildable	392
We could fix code...	394
...but we need to fix functionality	395
Figure out what functionality works	396
NOW you know what's not working	399
What would you do?	399
Spike test to estimate	400
What do the spike test results tell you?	402
Your team's gut feel matters	404
Give your customer the bug fix estimate	406
Things are looking good...	410
...and you finish the iteration successfully!	411
AND the customer is happy	412
Tools for your Software Development Toolbox	414

12 the real world

Having a process in life

You've learned a lot about Software Development. But before you go pinning burn down graphs in everyone's office, there's just a little more you need to know about dealing with each project... on its own terms. There are a lot of **similarities** and **best practices** you should carry from project to project, but there are **unique** things everywhere you go, and you need to be ready for them. It's time to look at how to apply what you've learned to **your particular project**, and where to go next for **more learning**.

Pinning down a software development process	418
A good process delivers good software	419
Formal attire required...	424
Some additional resources...	426
More knowledge == better process	427
Tools for your Software Development Toolbox	428

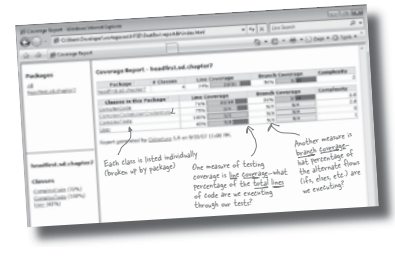


Story and Burn Down board

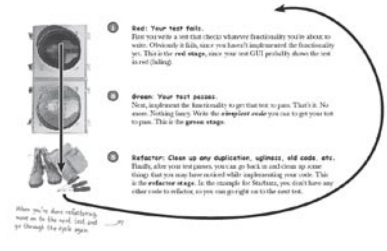
User Stories

Configuration Management (CM)

Continuous Integration (CI)



Test Coverage



Test Driven Development (TDD)

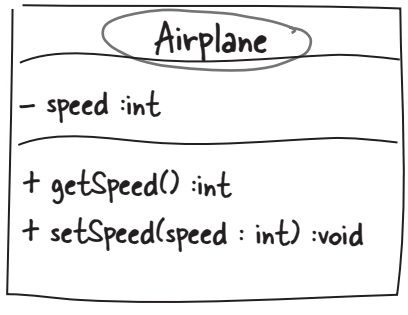
appendix 1: leftovers

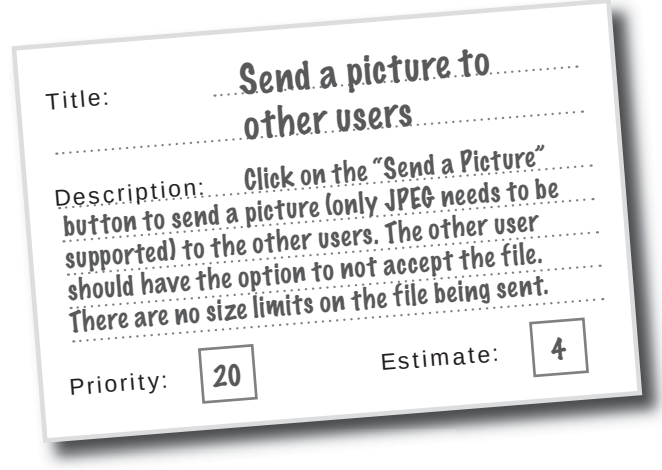
The top 5 things (we didn't cover)

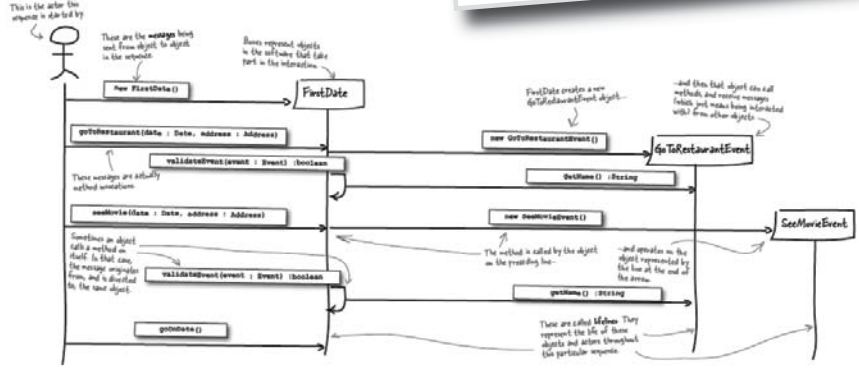
Ever feel like something's missing? We know what you mean…

Just when you thought you were done... there's more. We couldn't leave you without a few extra things, things we just couldn't fit into the rest of the book. At least, not if you want to be able to carry this book around without a metallic case and castor wheels on the bottom. So take a peek and see what you (still) might be missing out on.

#1. UML class Diagrams 434
#2. Sequence diagrams 436
#3. User stories and use cases 438
#4. System tests vs. unit tests 440
#5. Refactoring 441





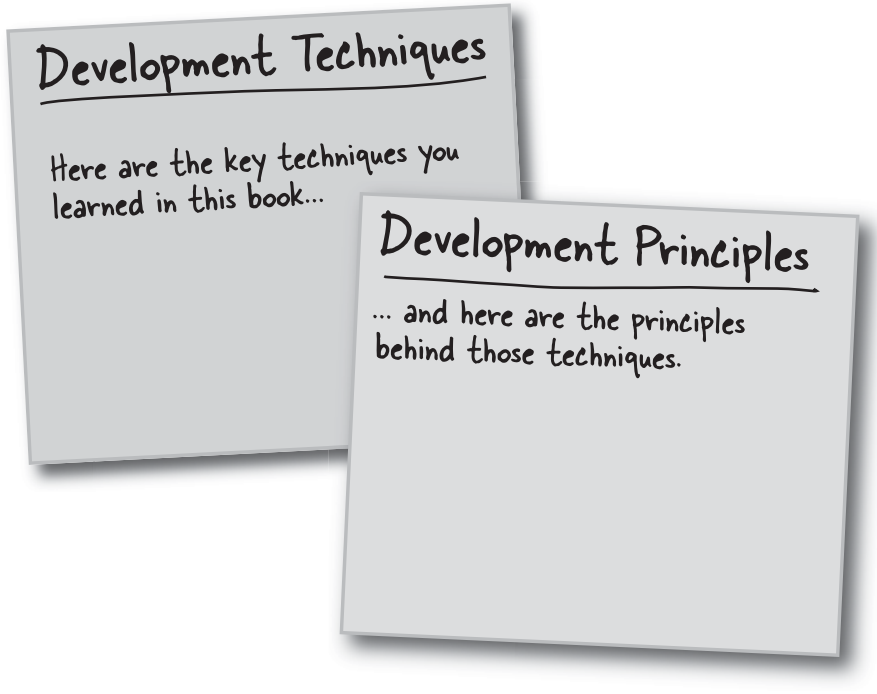


appendix 2: techniques and principles

Tools for the experienced software developer

Ever wished all those great tools and techniques were in one place? This is a roundup of all the software development **techniques** and **principles** we've covered. Take a look over them all, and see if you can **remember what each one means**. You might even want to **cut these pages out** and tape them to the bottom of your **big board**, for everyone to see in your daily standup meetings.

Development Techniques 444
Development Principles 446



Advance Praise for *Head First Software Development*

"*Head First Software Development* is a whimsical but very thoughtfully designed series of information diagrams and clever illustrations meant to accurately and clearly convey information directly into YOUR brain. It's a whole new kind of book."

— Scott Hanselman
Software Developer, Speaker, Author
Scott Hanselman's Computer Zen

"This is one of those books experienced developers wish they'd had back when they got started. I know, I'm one of them."

— Burk Hufnagel, Senior Software Architect

"I could have avoided a whole world of pain if I had read this book before my last project!"

— This developer asked to remain anonymous, so her last project's manager wouldn't be upset!

"*Head First Software Development* teaches many valuable lessons that will help anyone deliver quality software on time and on budget. Following the core principles taught in this book will help keep your project on track from start to finish. No matter how long you've been developing software, *Head First Software Development* will give you essential tools for developing successful projects from start to finish."

— Adam Z. Szymanski, Software Project Manager, Naval Research Laboratory

"The ideas in this book can be used by new and experienced managers to immediately improve their overall software development process."

— Dan Francis, Software Engineering Manager, Fortune 50 company

"A fresh new perspective on the software development process. A great introduction to managing a development team from requirements through delivery."

— McClellan Francis, Software Engineer

Praise for *Head First Object-Oriented Analysis and Design*

"*Head First Object-Oriented Analysis and Design* is a refreshing look at the subject of OOA&D. What sets this book apart is its focus on learning. There are too many books on the market that spend a lot of time telling you why, but do not actually enable the practitioner to start work on a project. Those books are very interesting, but not very practical. I strongly believe that the future of software development practice will focus on the practitioner. The authors have made the content of OOA&D accessible and usable for the practitioner."

— Ivar Jacobson, Ivar Jacobson Consulting

"I just finished reading *HF OOA&D*, and I loved it! The book manages to get across the essentials of object-oriented analysis and design with UML and use cases, and even several lectures on good software design, all in a fast-paced, easy to understand way. The thing I liked most about this book was its focus on why we do OOA&D—to write great software! By defining what great software is and showing how each step in the OOA&D process leads you towards that goal, it can teach even the most jaded Java programmer why OOA&D matters. This is a great 'first book' on design for anyone who is new to Java, or even for those who have been Java programmers for a while but have been scared off by the massive tomes on OO Analysis and Design."

— Kyle Brown, Distinguished Engineer, IBM

"Finally a book on OOA&D that recognizes that the UML is just a notation and that what matters when developing software is taking the time to think the issues through."

— Pete McBreen, Author, *Software Craftsmanship*

"The book does a good job of capturing that entertaining, visually oriented, 'Head First' writing style. But hidden behind the funny pictures and crazy fonts is a serious, intelligent, extremely well-crafted presentation of OO Analysis and Design. This book has a strong opinion of how to design programs, and communicates it effectively. I love the way it uses running examples to lead the reader through the various stages of the design process. As I read the book, I felt like I was looking over the shoulder of an expert designer who was explaining to me what issues were important at each step, and why."

— Edward Sciore, Associate Professor, Computer Science Department
Boston College

"This is a well-designed book that delivers what it promises to its readers: how to analyze, design, and write serious object-oriented software. Its contents flow effortlessly from using use cases for capturing requirements to analysis, design, implementation, testing, and iteration. Every step in the development of object-oriented software is presented in light of sound software engineering principles. The examples are clear and illustrative. This is a solid and refreshing book on object-oriented software development."

— Dung Zung Nguyen, Lecturer
Rice University

Praise for *Head First Design Patterns*

"I received the book yesterday and started to read it on the way home... and I couldn't stop. I took it to the gym and I expect people saw me smiling a lot while I was exercising and reading. This is tres 'cool'. It is fun but they cover a lot of ground and they are right to the point. I'm really impressed."

—Erich Gamma, IBM Distinguished Engineer, and co-author of Design Patterns

"'Head First Design Patterns' manages to mix fun, belly-laughs, insight, technical depth and great practical advice in one entertaining and thought provoking read. Whether you are new to design patterns, or have been using them for years, you are sure to get something from visiting Objectville."

—Richard Helm, coauthor of "Design Patterns" with rest of the Gang of Four—Erich Gamma, Ralph Johnson, and John Vlissides

"I feel like a thousand pounds of books have just been lifted off of my head."

—Ward Cunningham, inventor of the Wiki and founder of the Hillside Group

"This book is close to perfect, because of the way it combines expertise and readability. It speaks with authority and it reads beautifully. It's one of the very few software books I've ever read that strikes me as indispensable. (I'd put maybe 10 books in this category, at the outside.)"

—David Gelernter, Professor of Computer Science, Yale University and author of "Mirror Worlds" and "Machine Beauty"

"A Nose Dive into the realm of patterns, a land where complex things become simple, but where simple things can also become complex. I can think of no better tour guides than the Freemans."

—Miko Matsumura, Industry Analyst, The Middleware Company Former Chief Java Evangelist, Sun Microsystems

"I laughed, I cried, it moved me."

—Daniel Steinberg, Editor-in-Chief, java.net

"My first reaction was to roll on the floor laughing. After I picked myself up, I realized that not only is the book technically accurate, it is the easiest to understand introduction to design patterns that I have seen."

—Dr. Timothy A. Budd, Associate Professor of Computer Science at Oregon State University and author of more than a dozen books, including *C++ for Java Programmers*

"Jerry Rice runs patterns better than any receiver in the NFL, but the Freemans have out run him. Seriously...this is one of the funniest and smartest books on software design I've ever read."

—Aaron LaBerge, VP Technology, ESPN.com

Head First Software Development

by Dan Pilone and Russ Miles

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 (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators: Kathy Sierra, Bert Bates

Series Editor: Brett D. McLaughlin

Design Editor: Louise Barr

Cover Designers: Louise Barr, Steve Fehler

Production Editor: Sanders Kleinfeld

Indexer: Julie Hawks

Page Viewers: Vinny, Nick, Tracey, and Corinne

Printing History:

December 2007: First Edition.

Vinny, Tracey, Nick and Dan

Russ and Corinne

No sleepovers were conducted in the writing of this book, although one author did purportedly get engaged using his prototype of the iSwoon application. And one pig apparently lost its nose, but we're confident that had nothing to do with the software development techniques espoused by this text.

ISBN-10: 0-596-52735-7

ISBN-13: 978-0-596-52735-8

[M]

To everyone who's worked on a project with us and told us where we've gone wrong, where we've gone right, and what books to read…here's our contribution back.

Author(s) of Head First Software Development

Russ Miles

Dan Pilone

Dan is eternally grateful to his wife Tracey for letting him finish this book. Dan is a software architect for Vangent, Inc., and has led teams for the Naval Research Laboratory and NASA, building enterprise software. He's taught graduate and undergraduate Software Engineering at Catholic University in Washington, D.C. Some of his classes were interesting.

Dan started writing for O'Reilly by submitting a proposal for this book a little over five years ago. Three UML books, some quality time in Boulder, Colorado, with the O'Reilly Head First team, and a co-author later, he finally got a chance to put this book together.

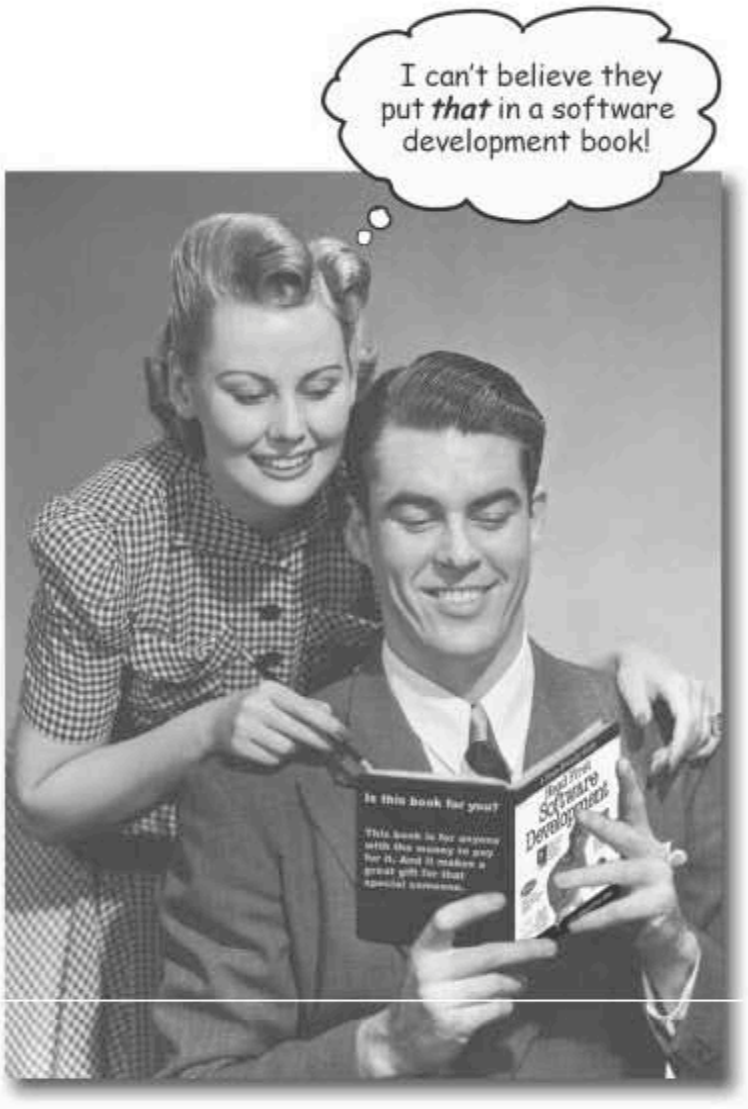
While leading a team of software developers can be challenging, Dan is waiting patiently for someone to write **Head First Parenting** to help sort out seriously complex management problems.

Russ is totally indebted to his fiancée, Corinne, for her complete love and support while writing this book. Oh, and he still can't believe she said yes to getting married next year, but I guess some guys have all the luck!

Russ has been writing for a long time and gets a huge kick out of demystifying technologies, tools, and techniques that shouldn't have been so mystified in the first place. After being a developer at various ranks for many years, Russ now keeps his days (and sometimes nights) busy by heading up a team of software developers working on super secret services for the music industry. He's also just finished up his Oxford Masters degree that only took him five years. He's looking forward to a bit of rest...but not for too long.

Russ is an avid guitar player and is relishing the spare time to get back to his guitars. The only thing he's missing is **Head First Guitar**...c'mon Brett, you know you want that one!

Intro



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

Who is this book for?

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

1. Do you have access to a computer and **some background in programming**?
2. Do you want to **learn techniques for building and delivering great software?** Do you want to **understand** the principles behind iterations and test-driven development?
3. Do you prefer **stimulating dinner party conversation** to **dry**, **dull**, **academic lectures**?

We use Java in the book, but you can squint and pretend it's C#. No amount of squinting will make you think it's Perl, though.

this book is for you.

Who should probably back away from this book?

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

1. **Are you completely new to Java?**

   (You don't need to be advanced, and if you know C++ or C# you'll understand the code examples just fine.)

2. Are you a kick-butt development manager looking for a ***reference* book**?

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 iterations are anthropomorphized?

this book is not for you.

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

We know what you're thinking

"How can *this* be a serious software development book?"

"What's with all the graphics?"

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

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.*

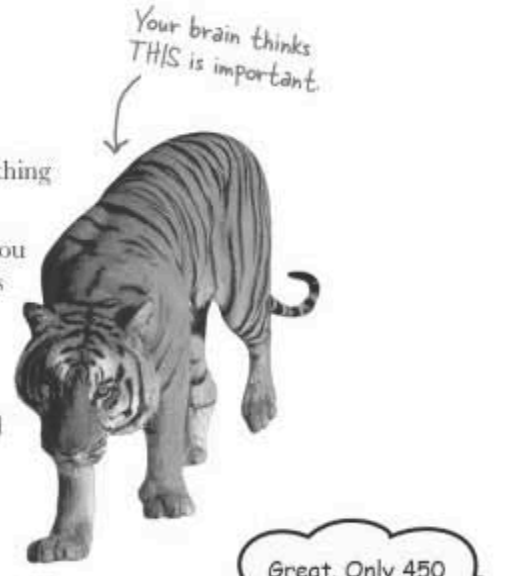
And that's how your brain knows...

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, ten 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* non-important 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 the guy with the handle "BigDaddy" on MySpace probably isn't someone to meet with after 6 PM.

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."



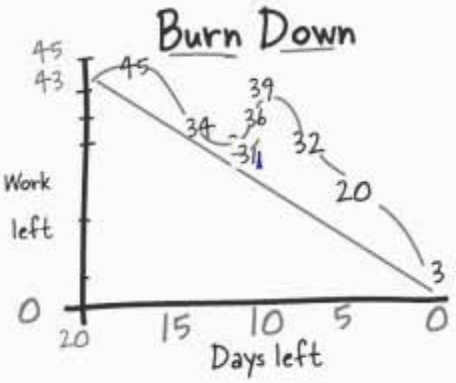
We think of a "Head First" reader as a learner.

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:

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 how to really develop great software. And you probably don't want to spend a lot of time. If you want to use what you read in this book, 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 treat software development like it was a hungry tiger?

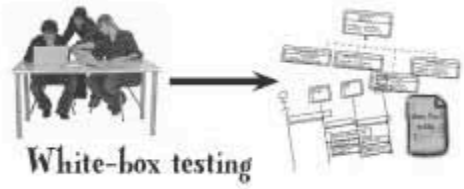
There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *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-do-able, 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, etc., 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. **Write a lot of software!**

   There's only one way to learn to develop software: you have to **actually develop software**. And that's what you're going to do throughout this book. We're going to give you lots of requirements to capture, techniques to evaluate, and code to test and improve: every chapter has exercises that pose a problem for you to solve. Don't just skip over them—a lot of the learning happens when you solve the exercises. We included a solution to each exercise—don't be afraid to **peek at the solution** if you get stuck! (It's easy to get snagged on something small.) But try to solve the problem before you look at the solution.

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. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

We assume you are familiar with object-oriented programming.

It would take an entire book to teach you object-oriented programming (like, say, **Head First OOA&D**). We chose to focus this book on software development principles rather than design or language basics. We picked Java for our examples because it's fairly common, and pretty self-documenting; but everything we talk about should apply whether you're using Java, C#, C++, or Visual Basic (or Ruby, or...) However, if you've never programmed using an object-oriented language, you may have some trouble following some of the code. In that case we'd strongly recommend you get familiar with one of those languages before attacking some of the later chapters in the book.

We don't cover every software development process out there.

There are tomes of information about different ways to write software. We don't try to cover every possible approach to developing code. Instead, we focus on techniques that we know work and fit well together to produce great software. Chapter 12 specifically talks about ways to tweak your process to account for unique things on your project.

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. Some exercises are there just to make you think about **how** you would solve the problem. ***Don't skip the exercises.*** The crossword puzzles are the only thing you don't *have* to do, but they're good for giving your brain a chance to think about the words and terms you've been learning in a different context.

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 examples are as lean as possible.

Our readers tell us that it's frustrating to wade through 200 lines of an example looking for the two lines they need to understand. Most examples in this book are shown within the smallest possible context, so that the part you're trying to learn is clear and simple. Don't expect all of the examples to be robust, or even complete—they are written specifically for learning, and aren't always fully functional.

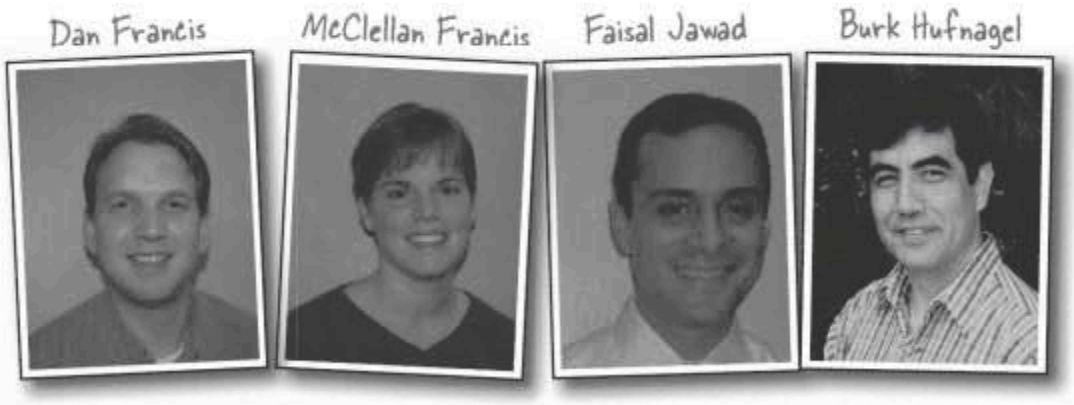
We've placed the full code for the projects on the Web so you can copy and paste them into your text editor. You'll find them at:

http://www.headfirstlabs.com/books/hfsd/

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 technical review team



Technical Reviewers:

This book wouldn't be anything like it is without our technical reviewers. They called us out when they disagreed with something, gave us "hurrah"s when something went right, and sent back lots of great commentary on things that worked and didn't work for them in the real world. Each of them brought a different perspective to the book, and we really appreciate that. For instance, **Dan Francis** and **McClellan Francis** made sure this book didn't turn into *Software Development for Java*.

We'd particularly like to call out **Faisal Jawad** for his thorough and supportive feedback (he started the "hurrahs"). **Burk Hufnagel** provided great suggestions on other approaches he'd used on projects and made one author's late night of updates a lot more fun with his suggestion to include, "Bad dev team. No biscuit."

Finally, we'd like to thank **Lisa Kellner** and **Kristin Stromberg** for their great work on readability and pacing. This book wouldn't be what it is without all of your input.

Acknowledgments

Our editor:

Don't let the picture fool you. **Brett** is one of the sharpest and most professional people we've ever worked with, and his contributions are on every page. Brett supported this book through every positive and negative review and spent quality time with us in Washington, D.C. to make this a success. More than once, he baited us into a good argument and then took notes while we went at it. This book is a result of his hard work and support and we really appreciate it.

The O'Reilly team:

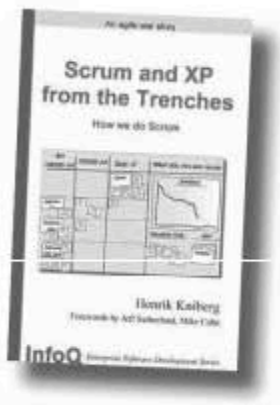
Lou Barr is the reason these pages look so "awesome." She's responsible for turning our vaguely worded "something that conveys this idea and looks cool" comments into pages that teach the material like no other book around.

Lou's favorite American-ism...

We'd also like to thank **Laurie Petrycki** for giving us the opportunity and making the tough decisions to get this book to where it is. We'd also like to thank **Catherine Nolan** and **Mary Treseler** for getting this book kicked off. Finally we'd like to thank **Caitrin McCullough**, **Sanders Kleinfeld**, **Keith McNamara**, and the rest of the O'Reilly production team for taking this book from the rough pages we sent in to a printed, high-class book with flawless grammar.

Scrum and XP from the Trenches:

We want to extend a special thanks to Henrik Kniberg for his book **Scrum and XP from the Trenches**. This book had significant influence on how we develop software and is the basis for some of the techniques we describe in this book. We're very grateful for the excellent work he's done.



Good book... highly recommended!

Our families:

No acknowledgments page would be complete without recognizing the contributions and sacrifices our families made for this book. Vinny, Nick, and Tracey have picked up the slack in the Pilone house for almost two years while this book came together. I can't convey how much I appreciate that and how your support and encouragement while "Daddy worked on his book" made this possible. Thank you.

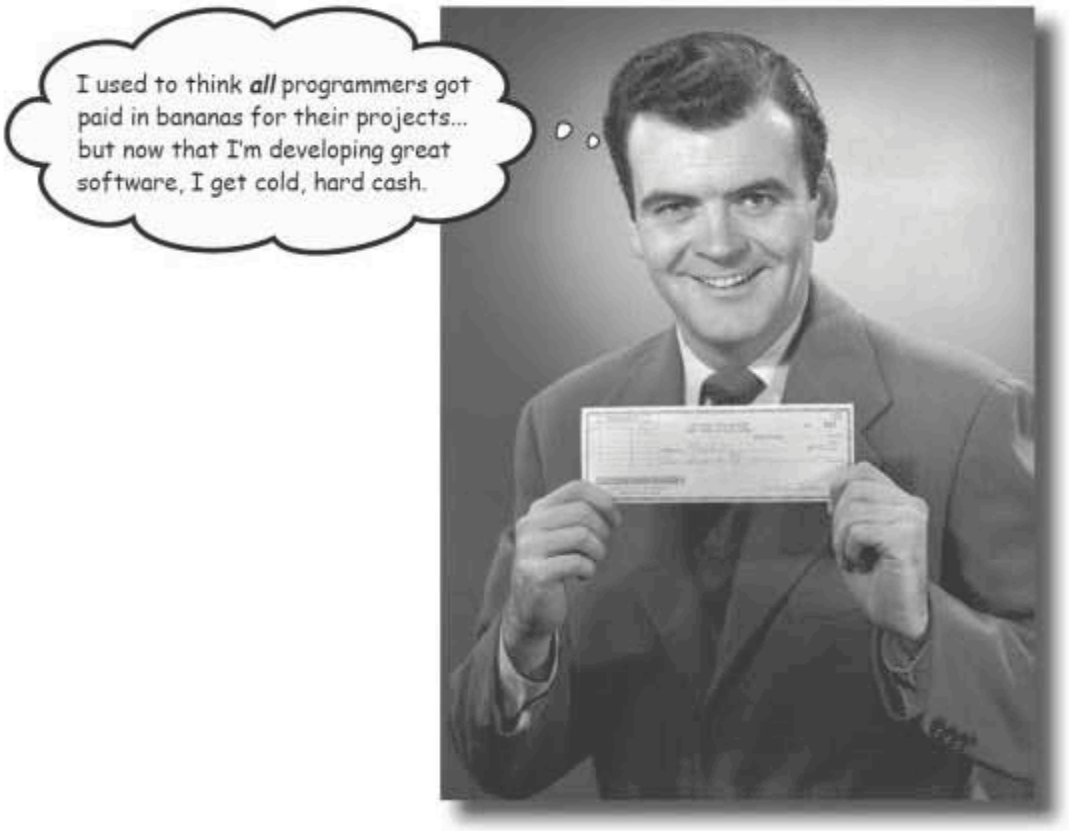
A massive thank you also goes out to the Miles household, that's Corinne (the boss) and Frizbee, Fudge, Snuff, Scrappy, and Stripe (those are the pigs). At every step you guys have kept me going and I really can't tell you how much that means to me. Thanks!

Safari® Books Online

When you see a Safari® icon on the cover of your favorite technology book that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at `http://safari.oreilly.com`.

Pleasing your customer



If your customer's unhappy, *everyone's* unhappy!

Every great piece of software starts with a customer's big idea. It's your job as a professional software developer to **bring that idea to life**. But taking a vague idea and turning it into working code—code that **satisfies your customer**—isn't so easy. In this chapter you'll learn how to avoid being a software development casualty by delivering software that is **needed**, **on time,** and **on budget**. Grab your laptop, and let's set out on the road to shipping great software.

Tom's Trails is going online

Trekkin' Tom has been providing world-famous trail guides and equipment from his shack in the Rockies for years. Now, he wants to ramp up his sales using a bit of the latest technology.

Most projects have two major concerns

Talk to most customers and, besides their big idea, they've probably got two basic concerns:

How much will it cost?

No surprise here. Most customers want to figure out how much cash they'll have to spend. In this case, though, Tom has a pile of money, so that's not much of a concern for him.

Usually, cash is a limitation. In this case, Tom's got money to spare, and figures what he spends will turn into an even bigger pile.

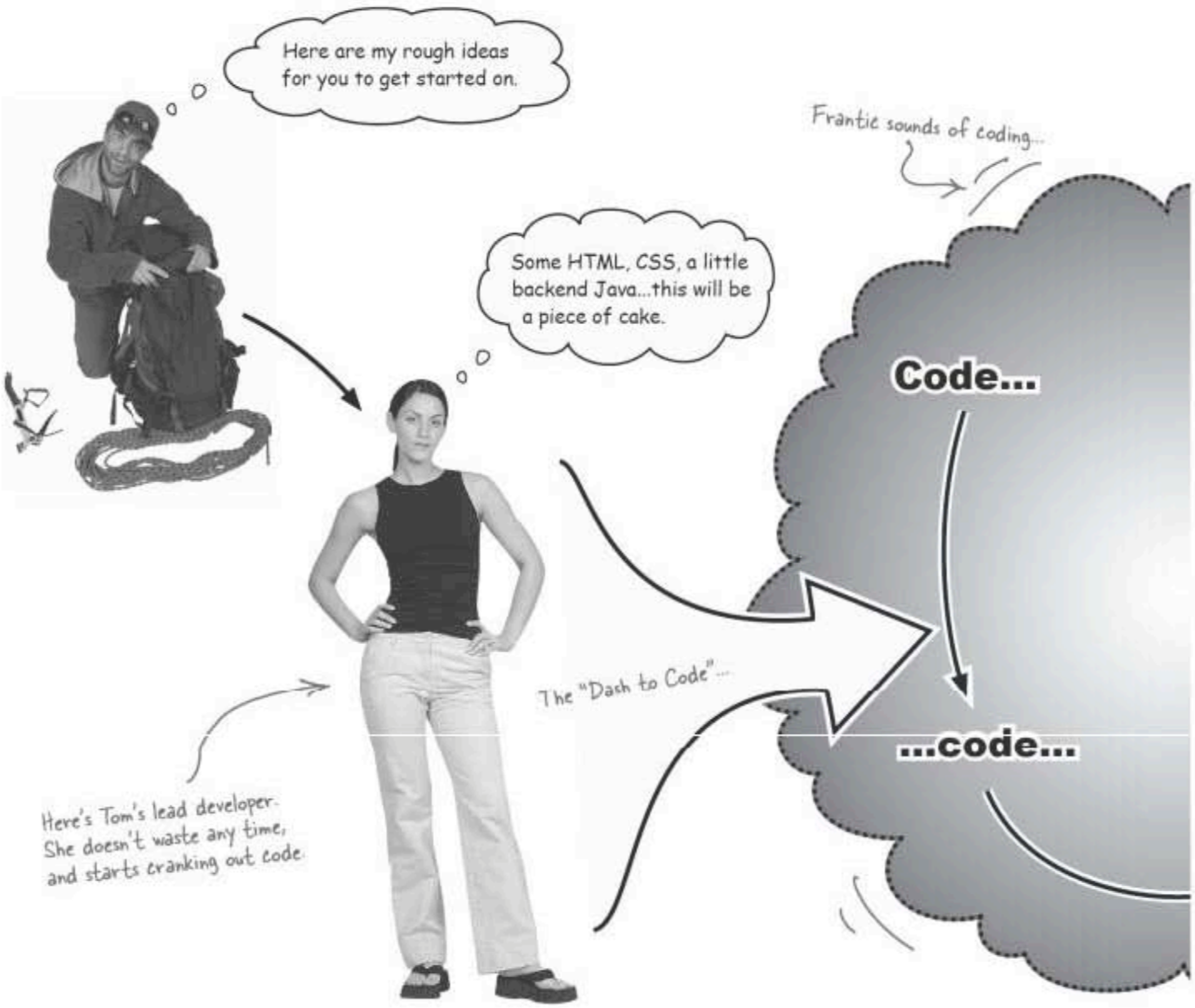
How long will it take?

The other big constraint is time. Almost no customer ever says, "Take as long as you want!" And lots of the time, you have a specific event or date the customer wants their software ready for.

In Tom's case, he wants his website available in three months' time, ready for the big TrailMix Conference coming up.

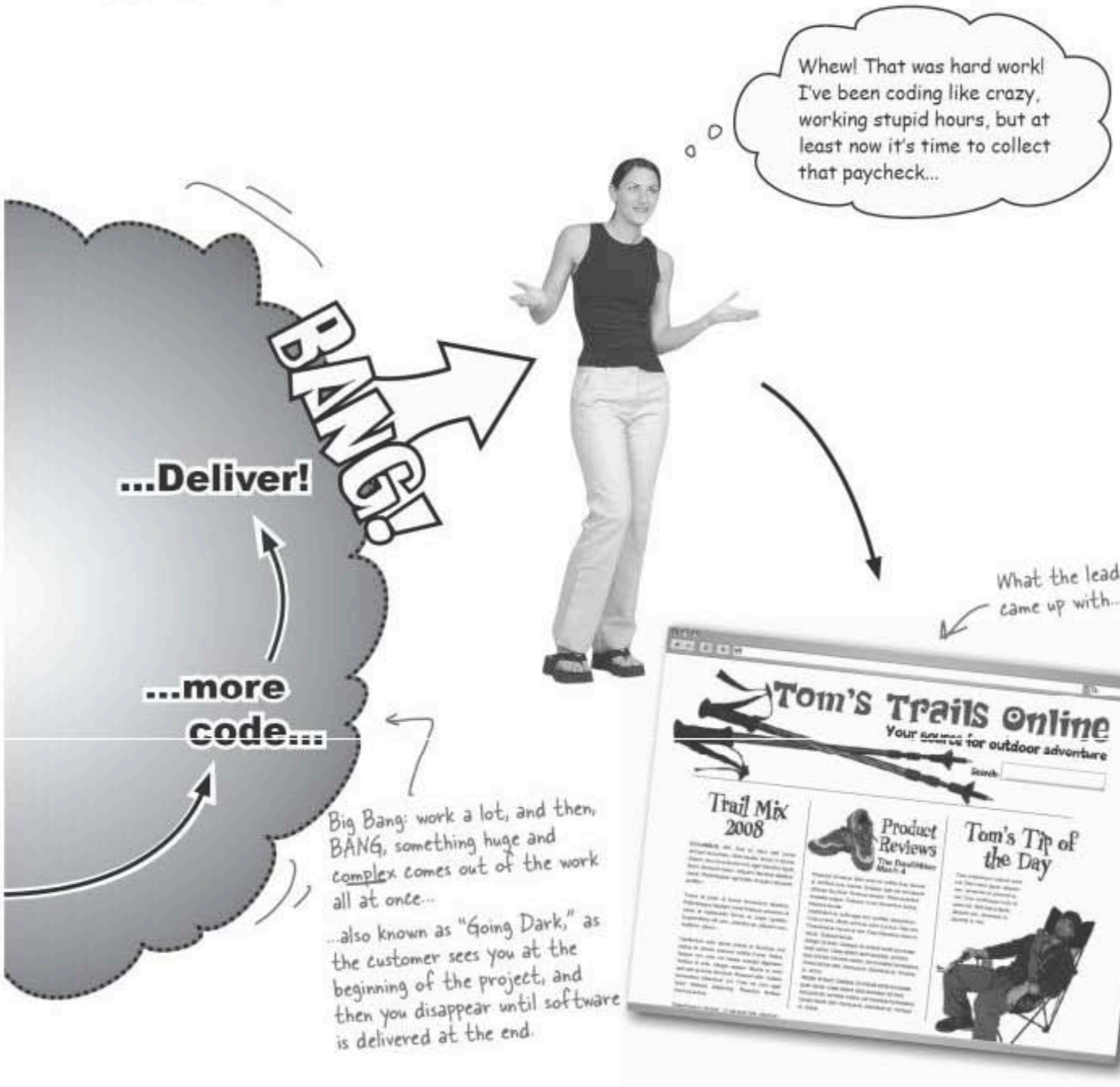
The Big Bang approach to development

With only a month to get finished, there's no time to waste.
The lead developer Tom hired gets right to work.



Flash forward: two weeks later

Tom's lead developer has pulled out all the stops to build Tom's Trails Online, putting all her finest coding skills into play and producing what she thinks Tom wants her to build.



Big bang development usually ends up in a BIG MESS

Even though a lot of work went into the project, Tom hasn't seen any of the (hopefully) completed work yet. Let's see what he thinks about the completed website.

If your customer isn't happy, you built the wrong software.

Big bang software usually means working a whole lot, but it also means not showing the customer much until your work is done. The risk with that approach is you **think** you're building what the customer wants with no real feedback until you **think** you're finished.

The emphasis here is that you think you're finished... but you may not be.

And, no matter how great YOU think your software is, it's the customer you have to make happy. So if the customer doesn't like what you've built, don't waste time trying to tell them they're wrong. Just get ready to do some rework.

But how do you figure out what the customer really wants? It's not always easy...

Can you figure out where things went wrong? Below are three things Tom said he wanted his site to allow for. Your job is to choose the option underneath each item that most closely matches what Tom means. And for the third one, you've got to figure out what he means on your own. Good luck!

1 **Tom says, "The customer should be able to search for trails."**

- ☐ The customer should see a map of the world and then enter an address to search for trails near a particular location.
- ☐ The customer should be able to scroll through a list of tourist spots and find trails that lead to and from those spots.
- ☐ The customer should be able to enter a zip code and a level of difficulty and find all trails that match that difficulty and are near that zip code.

2 **Tom says, "The customer should be able to order equipment."**

- ☐ The customer should be able to view what equipmentTom has and then create an order for items that are in stock.
- ☐ The customer should be able to order any equipment they need, but depending on stock levels the order may take longer if some of the items are back-ordered.

3 **Tom says, "The customer should be able to book a trip."**

Write what YOU think the software should do here

.....

.....

.....

Confused about what Tom really meant? It's okay... just do your best.

If you're having a hard time figuring out which option to choose, that's perfectly normal. Do your best, and we'll spend a lot more time in this chapter talking about how to figure out what the customer means.

Sharpen your pencil Solution

Can you figure out where things went wrong? Your job was to choose the option underneath each item that most closely matches what Tom means. And for the third one, you had to figure out what he means on your own.

?

A big question mark? That's your answer? How am I supposed to develop great software when I don't even know for sure what the customer wants?

If you're not sure what the customer wants, or even if you *think* you're sure, always go back and ask

When it comes to what is needed, the customer is king. But it's really rare for the customer to know ***exactly*** what he wants at the beginning of the project.

When you're trying to understand what your customer wants, sometimes there's not even a right answer in the customer's head, let alone in yours! If you disappear in a hurry and start coding, you may only have half the story... or even less.

But software shouldn't be guesswork. You need to ensure that you develop great software even when what's needed is not clear up front. So go ***ask*** the customer what they mean. ***Ask*** them for more detail. ***Ask*** them for options about how you might implement about their big ideas.

Software development is NOT guesswork. You need to keep the customer in the loop to make sure you're on the right path.

Great software development is...

We've talked about several things that you'll need for successful software. You've got the customer's big ideas to deal with, their money you're spending, and a schedule you've got to worry about. You've got to get all of those things right if you're going to build consistently great software.

Great software development delivers...

What the customer needs, otherwise called the software requirements. We'll talk more about requirements in the next chapter...

What is needed,

When we agreed with the customer that the software would be finished.

On Time,

and

On Budget

Not billing the customer for more money than was agreed upon.

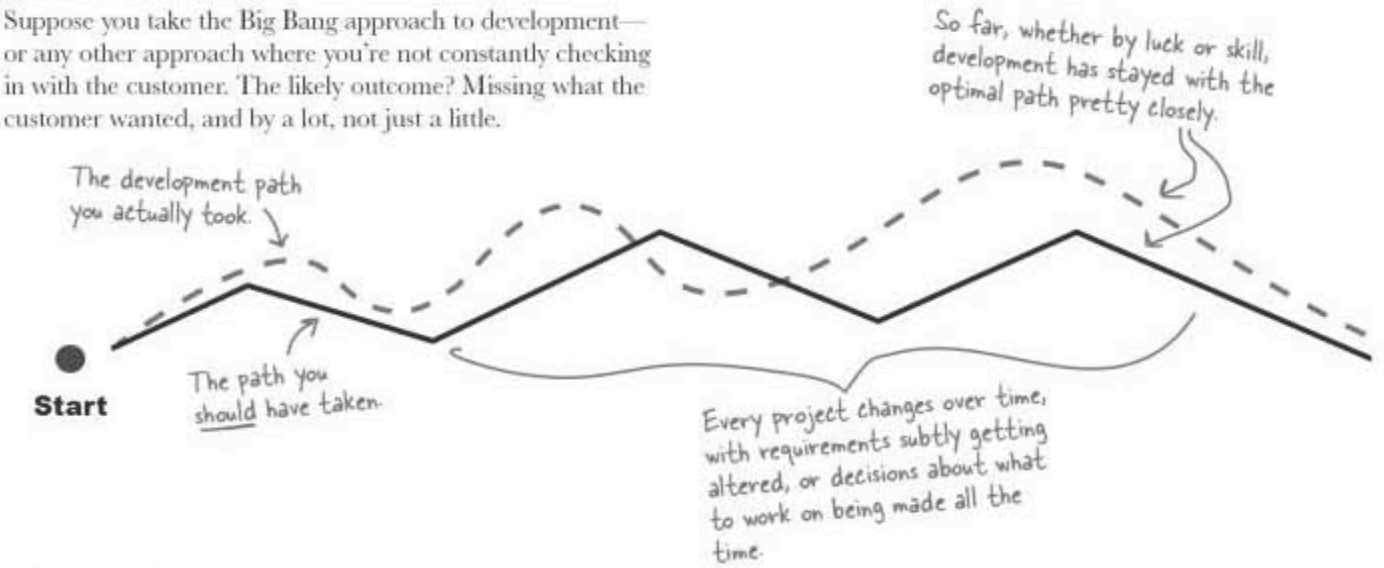
Can you think of three examples of software you've been involved with where at least one of these rules was broken?

Getting to the goal with ITERATION

The secret to great software development is **iteration**. You've already seen that you can't simply ignore the customer during development. But iteration provides you a way to actually ask the question, at each step of development, "How am I doing?" Here are two projects: one without iteration, and one with.

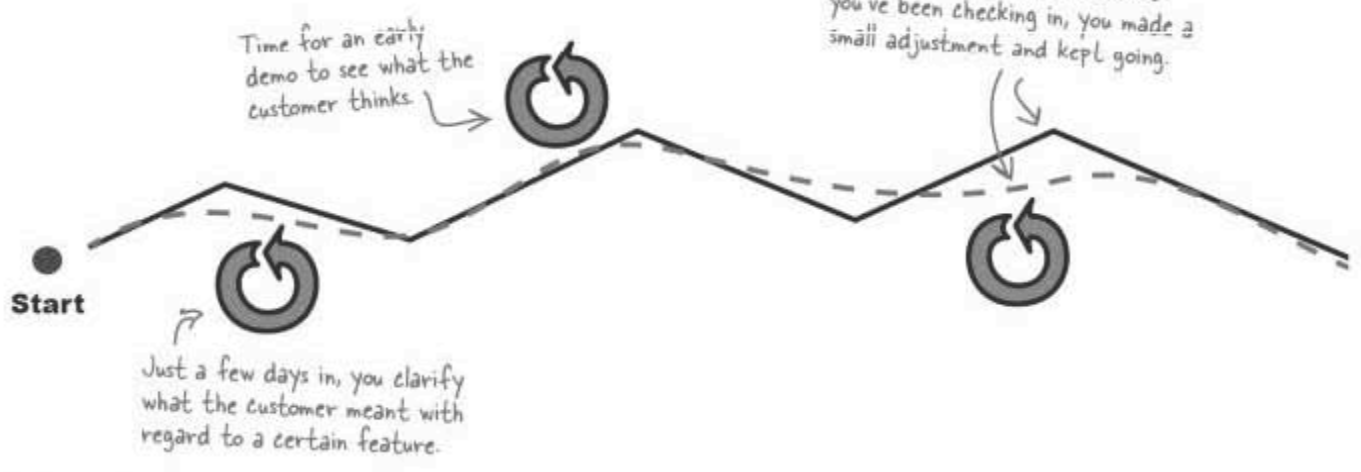
Without iteration...

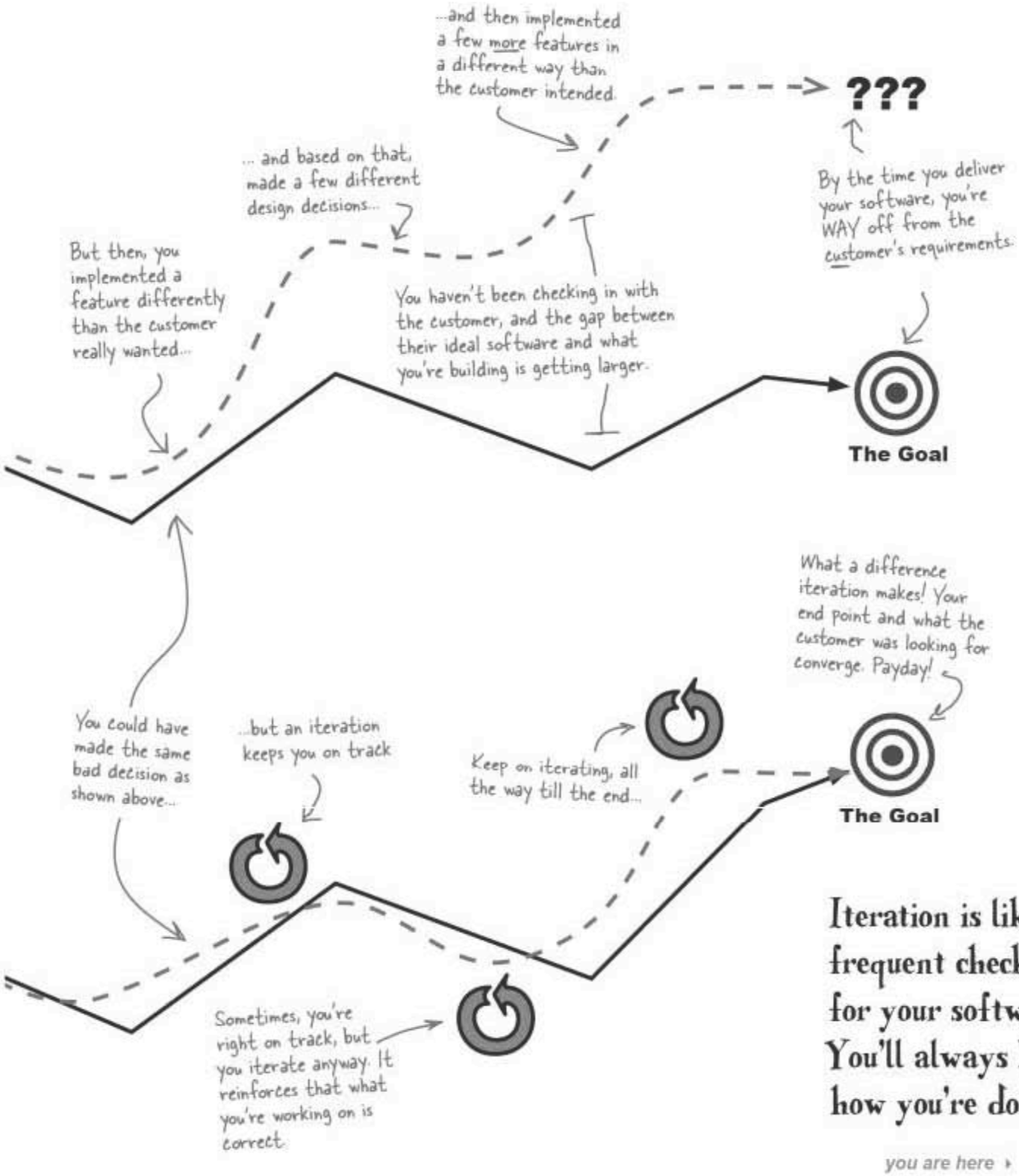
Suppose you take the Big Bang approach to development—or any other approach where you're not constantly checking in with the customer. The likely outcome? Missing what the customer wanted, and by a lot, not just a little.



With iteration...

This time, you decide that every time you make significant progress you'll check with the customer and refine what you're working on. You also don't make any major decisions without incorporating customer feedback.





Iteration is like a frequent checkup for your software. You'll always know how you're doing.

there are no Dumb Questions

Q: What if I'm sure that I know what the customer wants at the beginning of a project? Do I still need to iterate?

A: Absolutely. Iteration and getting feedback from your customer is important *especially* when you think you know it all up front. Sometimes it can seem like a complete no-brainer on a simple piece of software, but checking back with the customer is ALWAYS worth it. Even if the customer just tells you you're doing great, and even if you actually do start out with all the right requirements, iteration is still a way to make sure you're on the right track. And, don't forget, the customer can always change their mind.

Q: My entire project is only two months long. Is it worth iterating for such a short project?

A: Yep, iteration is still really useful even on a very short project. Two months is a whopping 60 days of chances to deviate from the customer's ideal software, or misunderstand a customer's requirement. Iteration lets you catch any potential problems like this before they creep into your project. And, better yet, before you look foolish in front of your customer.

No matter how big the team, or how long the project, iteration is ALWAYS one of the keys to building great software.

Q: Wouldn't it just be better to spend more time getting to know what the customer really wants, really getting the requirements down tight, than always letting the customer change their mind midstream?

A: You'd think so, but actually this is a recipe for disaster. In the bad old days, developers used to spend ages at the beginning of a project trying to make sure they got all the customer's requirements down completely before a single line of code or design decision was made.

Unfortunately, this approach still failed. Even if you think that you completely understand what the customer needs at the beginning, the ***customer*** often doesn't understand. So they're figuring out what they want as much as you are.

You need a way of helping your team and your customer grow their understanding of their software as you build it, and you can't do that with a Big Bang, up-front requirements approach that expects everything to be cast in stone from day one.

Q: Who should be involved in an iteration?

A: Everyone who has a say in whether your software meets its requirements, and everyone who is involved in meeting those requirements. At a minimum, that's usually your customer, you, and any other developers working on the project.

Q: But I'm only a team of one, do I still need to iterate?

A: Good question, and the answer is yes (starting to detect a theme here?). You might only be a development team of one, but when it comes to your project there are always, at a minimum, two people who have a stake in your software being a success: your customer and you. You still have two perspectives to take into account when making sure your software is on the right path, so iteration is still really helpful even in the smallest of teams.

Q: How early in a project should I start iterating?

A: As early as you have a piece of software running that you can discuss with your customer. We normally recommend around 20 work days—1 calendar month, per iteration as a rule of thumb—but you could certainly iterate earlier. One- or two-week iterations are not unheard of. If you aren't sure about what a customer means on Day 2, call them. No sense waiting around, guessing about what you should be doing, right?

Q: What happens when my customer comes back with bad news, saying I'm way off on what I'm building. What do I do then?

A: Great question! When the worst happens and you find that you've deviated badly during an iteration, then you need to bring things back into line over the course of the next couple of iterations of development. How to do this is covered later on, but if you want to take a peek now, fast-forward to Chapter 4.

OK, I get it, iteration is important. But you said I should iterate every time I have working software, around every 30 calendar days, or 20 work days. What if I don't have anything that can run after a month? What can I show the customer?

20 working days is only a guideline. You might choose to have longer or shorter iterations for your project.

An iteration produces working software

With the old Big Bang approach to developing software, you probably wouldn't have any software ready until the end of the project, which is the worst time to realize that you've gone wrong!

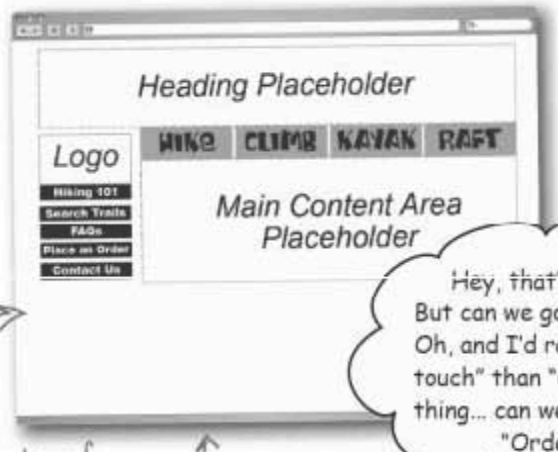
Continuous building and testing is covered in Chapters 6 and 7.

With iteration, you check every step of the way that you're going in the right direction. That means making sure your software builds from almost day one (and more like hour one if you can manage it). You shouldn't have long periods where code doesn't work or compile, even if it's just small bits of functionality.

A working build also makes a big difference to your team's productivity because you don't have to spend time fixing someone else's code before you can get on with your own tasks

Then you show your customer those little pieces of functionality. It's not much, sometimes, but you can still get an OK from the customer.

Tom got to see working software, and made some important comments you could address right away.



Here's a very simple portion of the Tom's Trails website. It only has the navigation, but it's still worth seeing what Tom thinks.

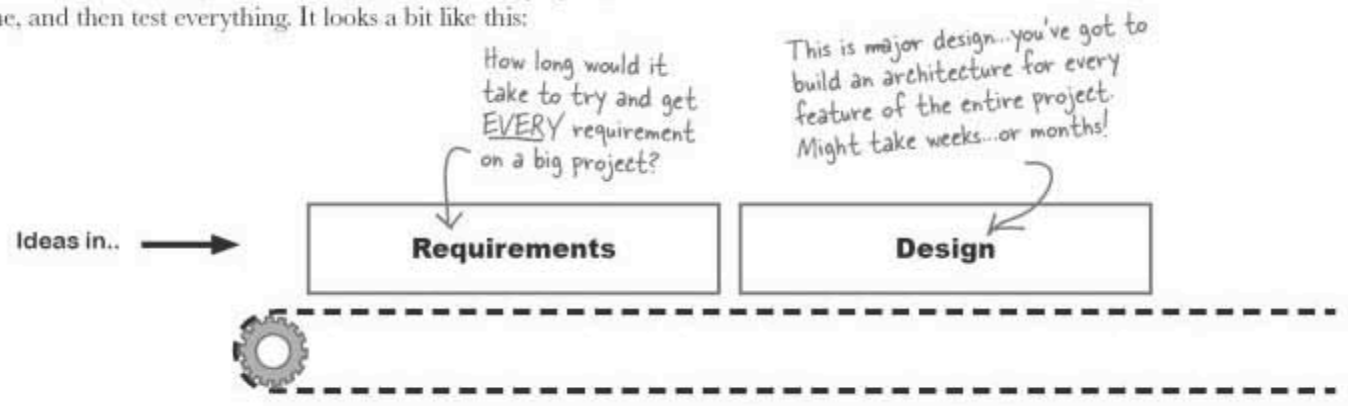
Instead of building the entire site at once, we broke the problem up into smaller chunks of functionality. Each chunk can then be demonstrated to the customer separately.

Hey, that's looking good. But can we go with rounded tabs? Oh, and I'd rather call it "Get in touch" than "Contact Us." Last thing... can we add an option for "Order Status?"

Each iteration is a mini-project

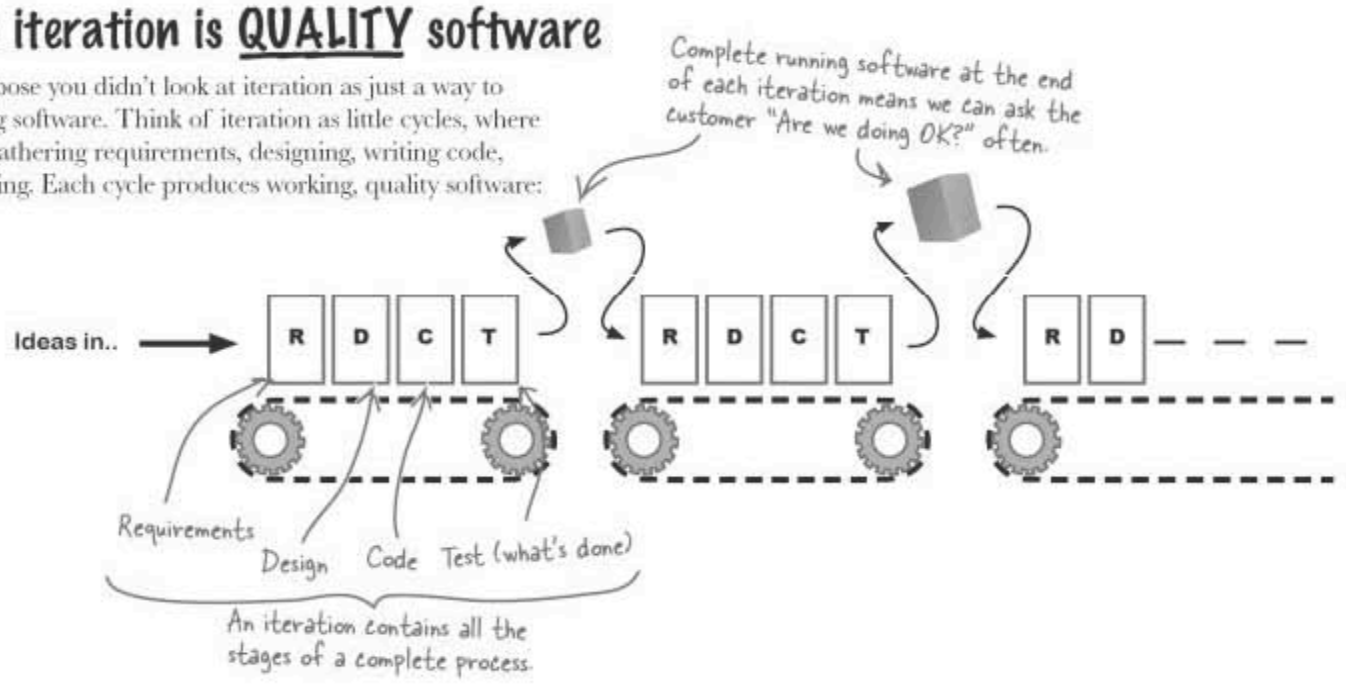
With iteration, you take the steps you'd follow to build the entire project, and put those steps into **each iteration**. In fact, each iteration is a mini-project, with its own requirements, design, coding, testing, etc., built right in. So you're not showing your customer junk... you're showing them well-developed bits of the final software.

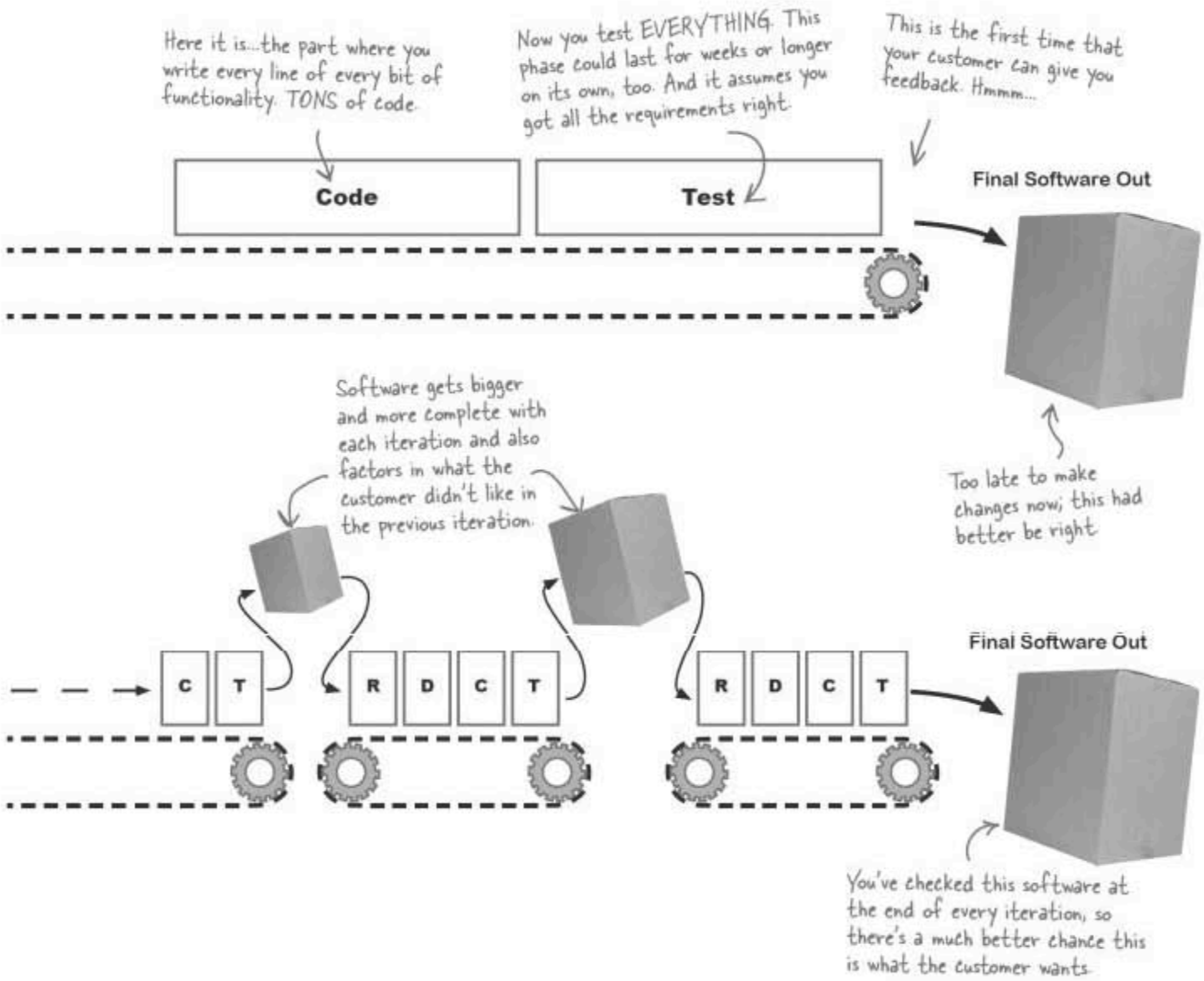
Think about how most software is developed: You gather requirements (what your customer wants), build a design for the entire project, code for a long time, and then test everything. It looks a bit like this:



Each iteration is QUALITY software

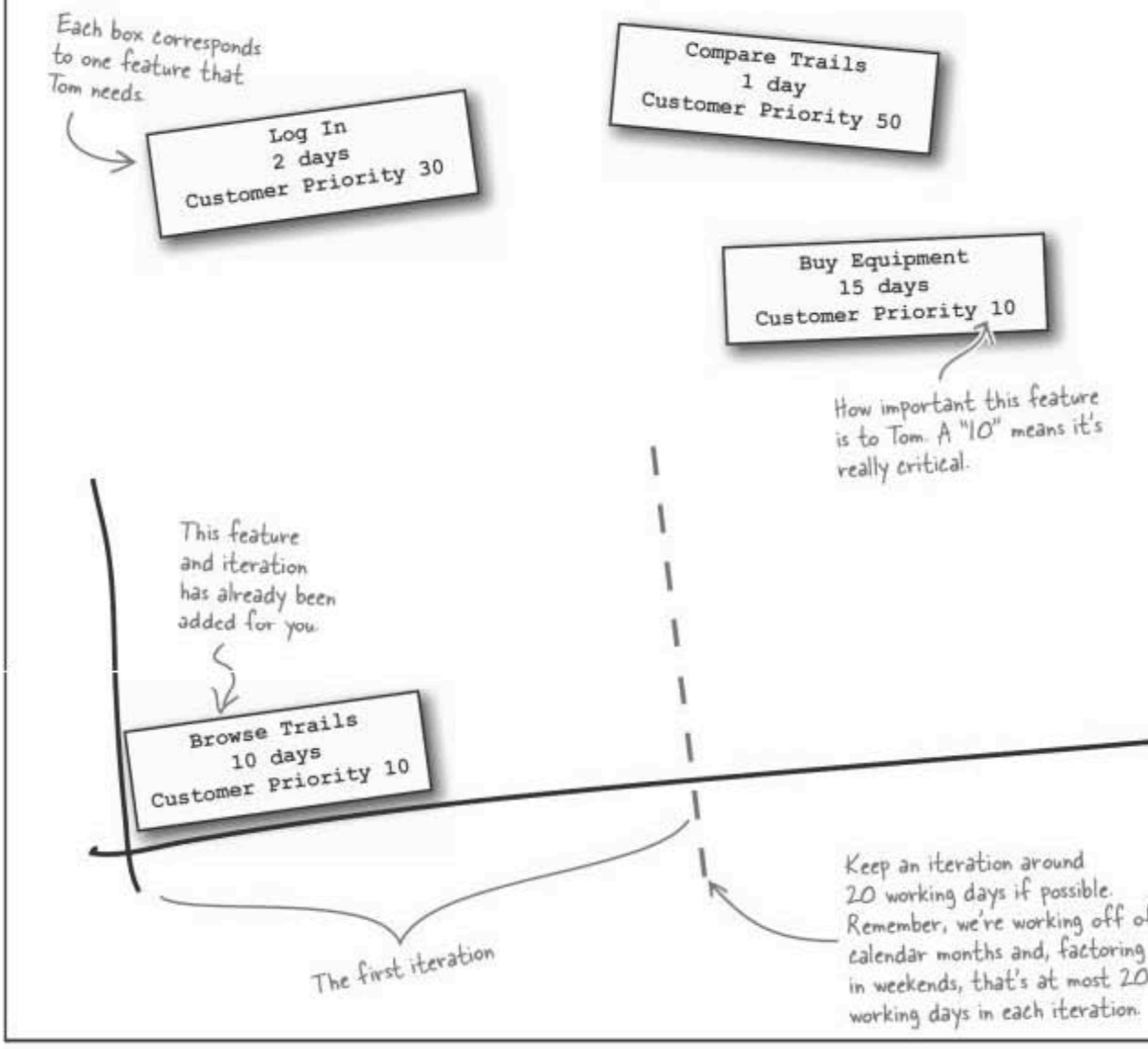
But suppose you didn't look at iteration as just a way to write big software. Think of iteration as little cycles, where you're gathering requirements, designing, writing code, and testing. Each cycle produces working, quality software:




Here it is...the part where you write every line of every bit of functionality. TONS of code.
Now you test EVERYTHING. This phase could last for weeks or longer on its own, too. And it assumes you got all the requirements right.
This is the first time that your customer can give you feedback. Hmmm...
Code
Test
Final Software Out
Too late to make changes now; this had better be right
Software gets bigger and more complete with each iteration and also factors in what the customer didn't like in the previous iteration.
C
T
R
D
C
T
R
D
C
T
Final Software Out
You've checked this software at the end of every iteration, so there's a much better chance this is what the customer wants

Exercise

It's time to bring iterations into play on Tom's Trails. Each of the features that Tom wants for Trails Online has had estimates added to specify how long it will take to actually develop. Then we figured out how important each is to Tom and then assigned a priority to each of them (10 being the highest priority, 50 being the lowest). Take each feature and position them along the project's timeline, adding an iteration when you think it might be useful.



Oh, one other thing. Tom doesn't want customers to be able to buy equipment unless they've logged in to the web site. Be sure and take that into account in your plan.

Each feature has an estimate to show how long it should take to develop that feature (in actual working days).

List Equipment
7 days
Customer Priority 10

Add a Review
2 days
Customer Priority 20

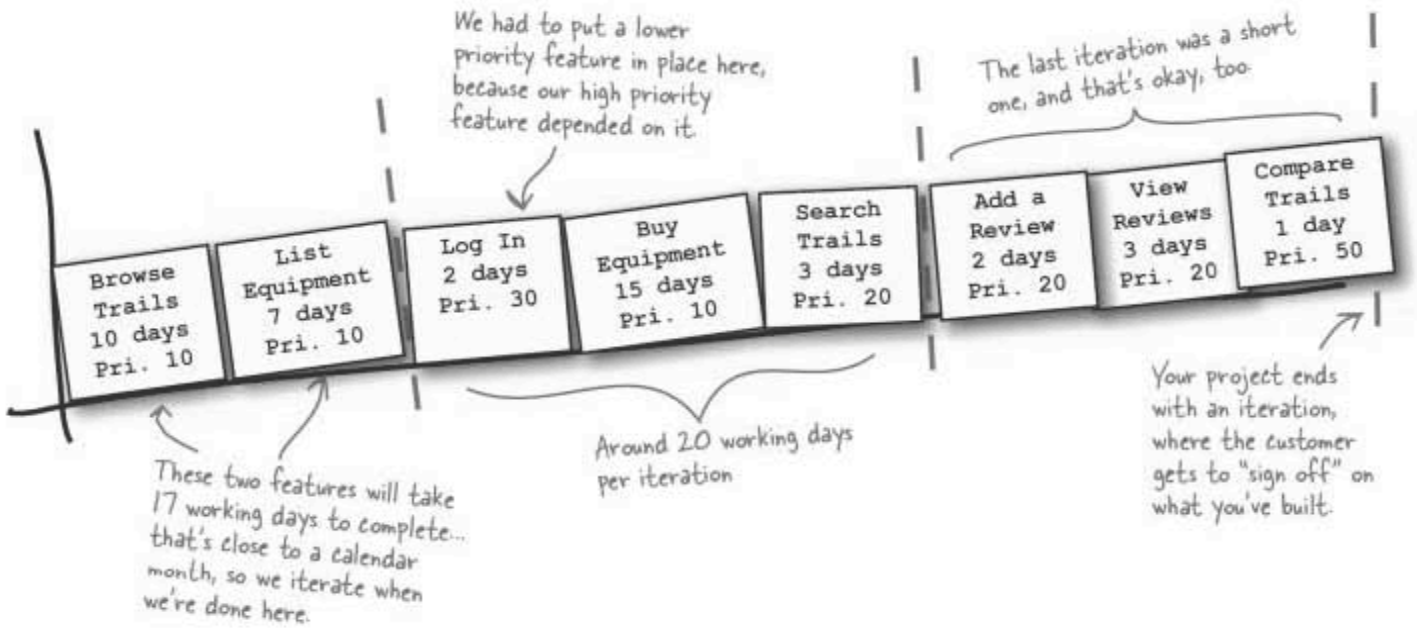
View Reviews
3 days
Customer Priority 20

Search Trails
3 days
Customer Priority 20

10 is high priority, 50 is low. We'll look at why these priorities are in increments of 10 in Chapter 3.

Don't forget to add as many iterations as you think will be useful.

Your job was to build an iteration plan for Tom's Trails. You should have come up with something like we did, below:



This is probably the only plan that serves the customer's priorities, keeps iterations at a manageable length, and gets the job done. If you came up with something different, take a hard look at why you made different choices than we did.

Your iteration length should be at the right tempo for YOUR project

An iteration helps you stay on track, and so you might decide to have iterations that are shorter or longer than 30 days. Thirty days might seem like a long time, but factor in weekends, and that means you're probably going to get 20 days of actual productive work per iteration. If you're not sure, try 30 calendar days per iteration as a good starting point, and then you can tweak for your project as needed.

The key here is to iterate often enough to catch youself when you're deviating from the goal, but not so often that you're spending all your time preparing for the end of an iteration. It takes time to show the customer what you've done and then make course corrections, so make sure to factor this work in when you are deciding how long your iterations should be.

there are no Dumb Questions

Q: The last feature scheduled for my iteration will push the time needed to way over a month. What should I do?

A: Consider shifting that feature into the next iteration. Your features can be shuffled around within the boundaries of a 20-day iteration until you are confident that you can successfully build an iteration within the time allocated. Going longer runs the risk of getting off course.

Q: Ordering things by customer priority is all fine and good, but what happens when I have features that need to be completed before other features?

A: When a feature is dependent on another feature, try to group those features together, and make sure they are placed within the same iteration. You can do this even if it means doing a lower-priority feature before a high-priority one, if it makes the high-priority feature possible.

This occurred in the previous exercise where the "Log In" feature was actually a low customer priority, but needed to be in place before the "Buy Equipment" feature could be implemented.

Q: If I add more people to the project, couldn't I do more in each of my iterations?

A: Yes, but be very careful. Adding another person to a project doesn't halve the time it takes to complete a feature. We'll talk more about how to factor in the overhead of multiple people in Chapter 2, when we talk about velocity.

Q: What happens when a change occurs and my plan needs to change?

A: Change is unfortunately a constant in software development, and any process needs to handle it. Luckily, an iterative process has change baked right in...turn the page and see what we mean.

The customer WILL change things up

Tom signed off on your plan, and Iteraton 1 has been completed. You're now well into your second iteration of development and things are going great. Then Tom calls...

Things are really starting to look great, but I had some thoughts after that last iteration. I think it's really important that Tom's Trails Online has a mailing list, so my customers can communicate with each other.

Remember, if your software doesn't do what the customer wants, you're not going to go very far in software development.

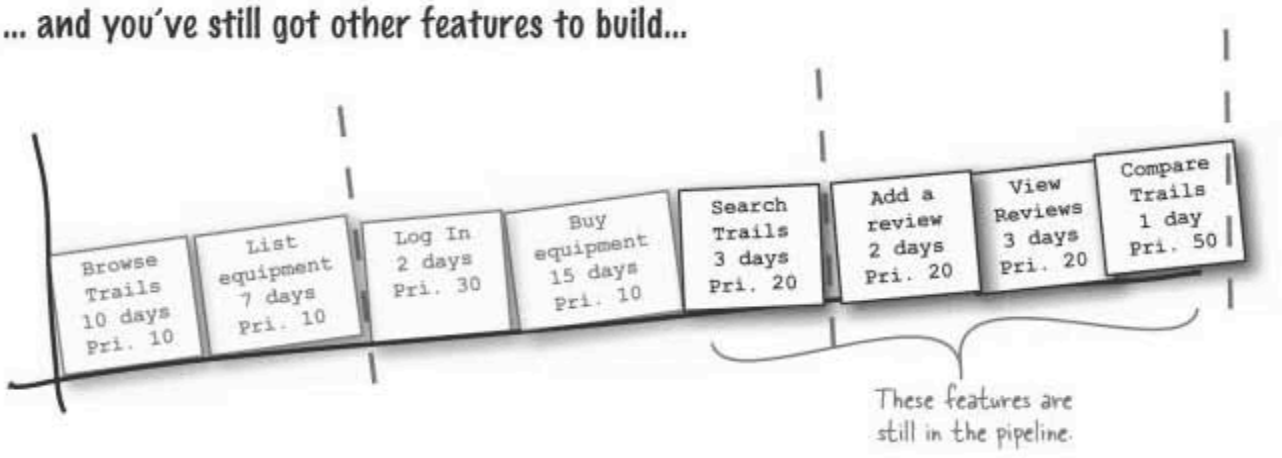
It's up to you to make adjustments

Tom's new idea means three new features, all high-priority. And we don't even know how long they'll take, either. But you've got to figure out a way to work these into your projects.

But there are some BIG problems...

You're already a long way into development...

... and you've still got other features to build...



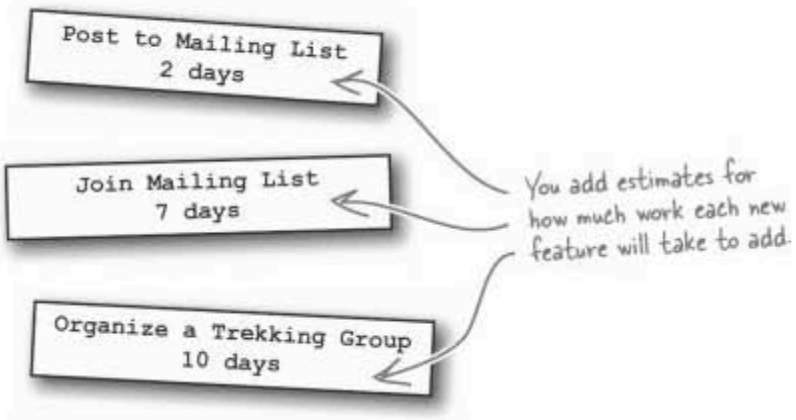
... and the deadline hasn't changed.

Iteration handles change automatically (well, sort of)

Your iteration plan is already structured around short cycles, and is built to handle lots of individual features. Here's what you need to do:

1 **Estimate the new features**

First, you need to estimate how long each of the new features is going to take. We'll talk a lot more about estimation in a few chapters, but for now, let's say we came up with these estimates for the three new features:

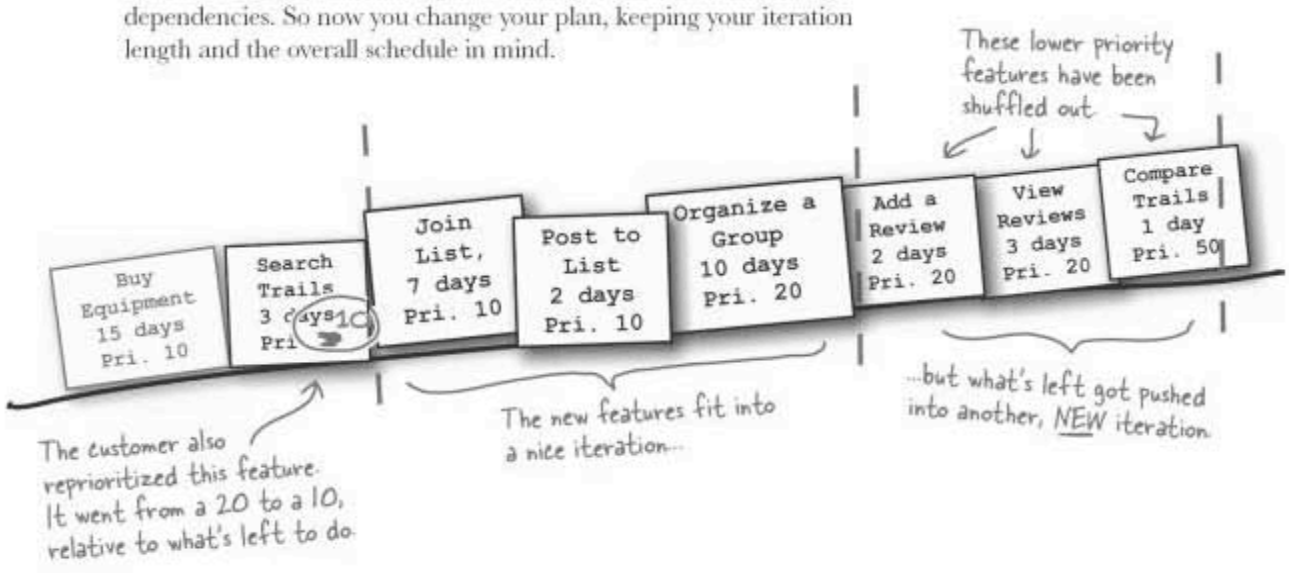


2 **Have your customer prioritize the new features**

Tom already gave everything a priority of "20," right? But you really need him to look at the other features left to implement as well, and prioritize in relation to those.

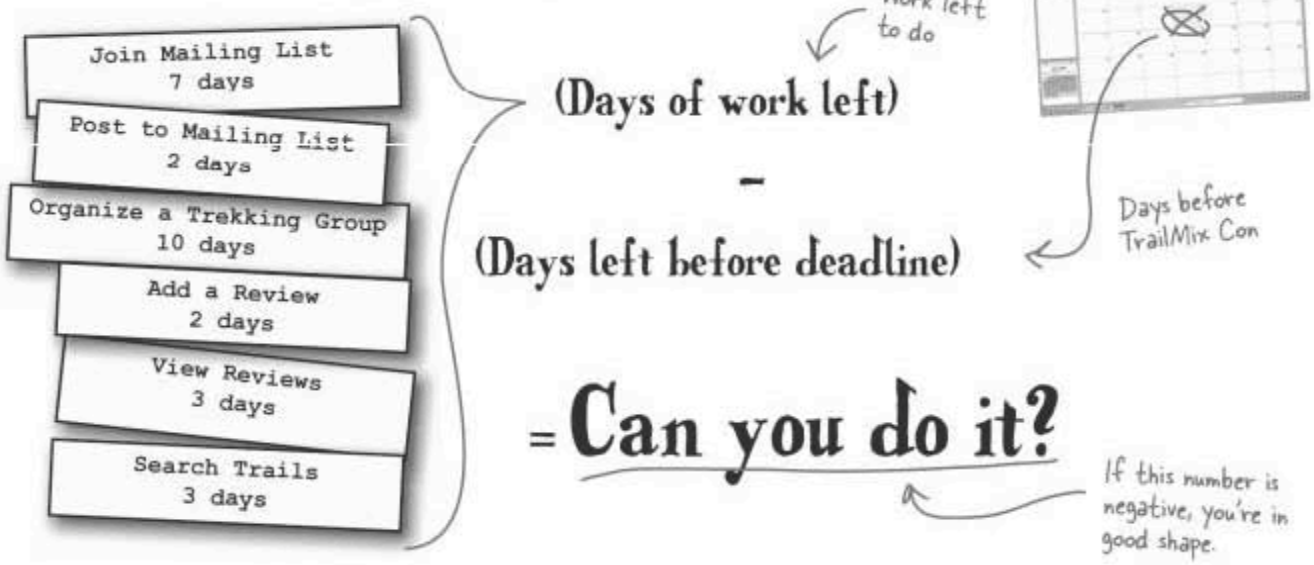
3 Rework your iteration plan

The ordering is set based on prioritization, and there aren't any dependencies. So now you change your plan, keeping your iteration length and the overall schedule in mind.



4 Check your project deadline

Remember the TrailMix Conference? You need to see if the work you've got left, including the new features, still can get done in time. Otherwise, Tom's got to make some hard choices.



A process is really just a sequence of steps

Process, particularly in software development, has gotten a bit of a bad name. A process is just a sequence of steps that you follow in order to do something—in our case, develop software. So when we've been talking about iteration, prioritization, and estimation, we've really been talking about a software development process.

Rather than being any formal set of rules about what diagrams, documentation, or even testing you should be doing (although testing is something we'd definitely recommend!), a process is really just what to do, and when to do it. And it doesn't need an acronym...it just has to work.

We don't really care what process you use, as long as it has the components that ensure you get great, quality software at the end of your development cycle.

The right software development process for YOU is one that helps YOU develop and deliver great software, on time and on budget.

Iteration is more than a process

Regardless of the actual steps involved in the process you choose, iteration is a best practice. It's an approach that can be applied to ***any*** process, and it gives you a better chance of delivering what is needed, on time and on budget. Whatever process you end up using, iteration should be a major part.

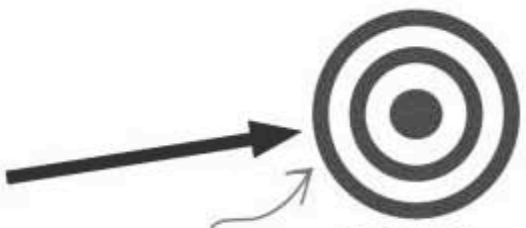
Your software isn't complete until it's been RELEASED

You added the new features, and now you and your team have finished the project on time and on schedule. At every step of the way, you've been getting feedback from the customer at the end of each iteration, incorporating that feedback, and new features, into the next iteration. Now you can deliver your software, and ***then*** you get paid.

Tom isn't talking about your software running on your machine... he cares about the software running in the real world.

Excellent. I'm already getting calls, and people love the new site. And our orders are up this week, mostly off of new customers that saw our online demo at TrailMix.

Nice work.



The Goal

Iteration helped you reach an achievable goal that captured what your customer needed.

there are no Dumb Questions

Q: What happens when the customer comes up with new requirements and you can't fit all the extra work into your current iteration?

A: This is when customer priority comes into play. Your customer needs to make a call as to what really needs to be done for this iteration of development. The work that cannot be done then needs to be postponed until the next iteration. We'll talk a lot more about iteration in the next several chapters.

Q: What if you don't *have* a next iteration? What if you're already on the last iteration, and then a top priority feature comes in from the customer?

A: If a crucial feature comes in late to your project and you can't fit it into the last iteration, then the first thing to do is explain to the customer why the feature won't fit. Be honest and show them your iteration plan and explain why, with the resources you have, the work threatens your ability to deliver what they need by the due date.

The best option, if your customer agrees to it, is to factor the new requirement into another iteration on the end of your project, extending the due date. You could also add more developers, or make everyone work longer hours, but be wary of trying to shoehorn the work in like this. Adding more developers or getting everyone to work longer hours will often blow your budget and rarely if ever results in the performance gains you might expect (see Chapter 3).

Tools for your Software Development Toolbox

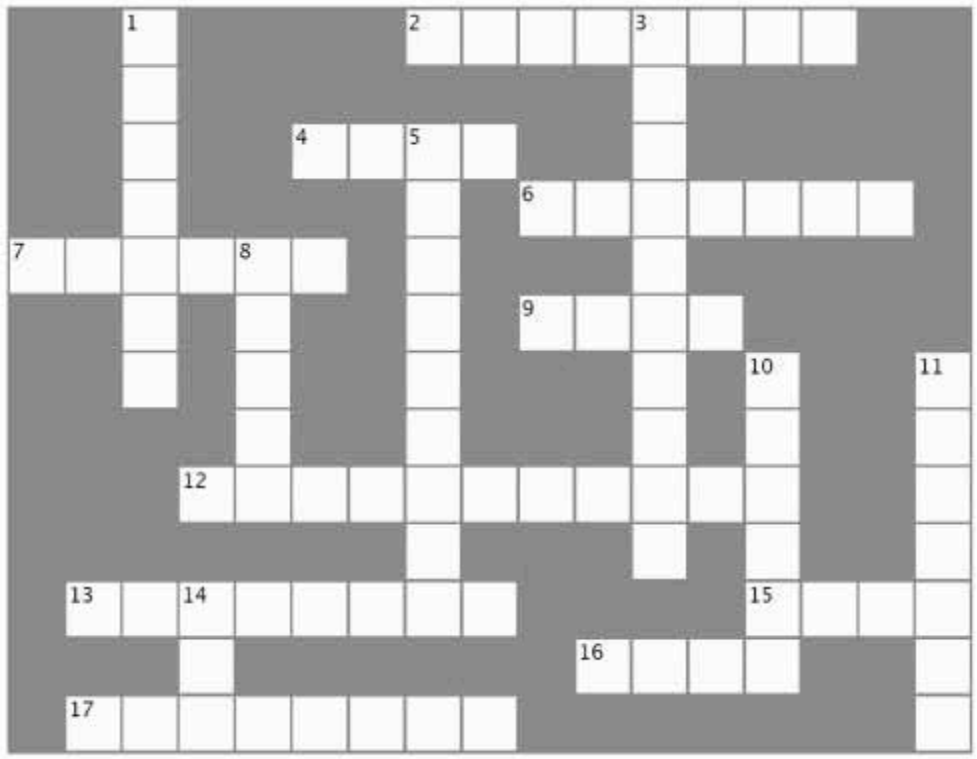
Software Development is all about developing and delivering great software. In this chapter, you learned about several techniques to keep you on track. For a complete list of tools in the book, see Appendix ii.

BULLET POINTS

- The **feedback** that comes out of each **iteration** is the best tool for ensuring that your software meets the needs of your customers.
- An iteration is a complete project in **miniature**.
- Successful software is not developed in a vacuum. It needs **constant feedback** from your customer using iterations.
- Good software development delivers great software, **on time** and **on budget**.
- It's always better to deliver **some** of the features **working perfectly** than all of the features that don't work properly.
- Good developers develop software; **great developers** ship software!

Software Development Cross

Let's put what you've learned to use and stretch out your left brain a bit. All of the words below are somewhere in this chapter. Good luck!



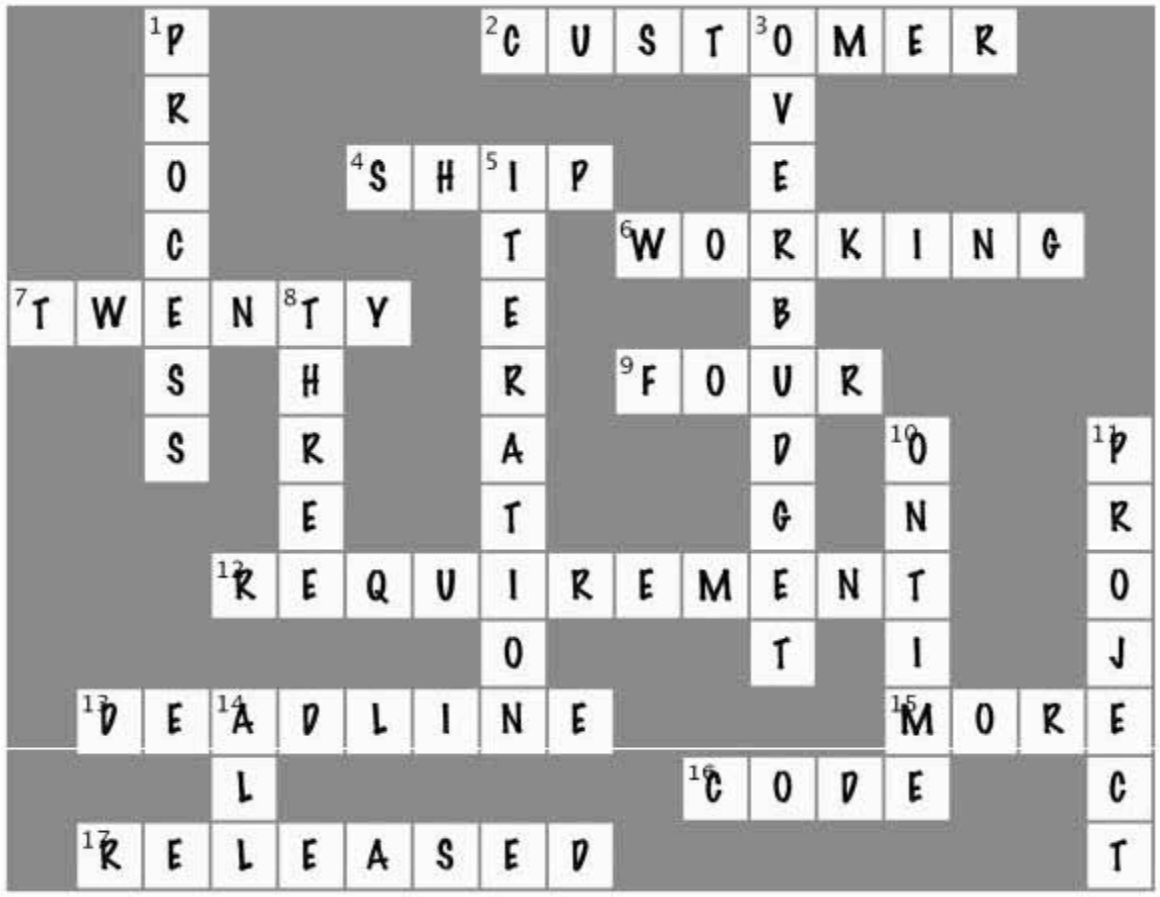
Across

2. I'm the person or company who ultimately decides if your software is worth paying for.
4. Good Developers develop, great developers .....
6. An iteration produces software that is .....
7. Aim for ..... working days per iteration.
9. The number of development stages that are executed within an iteration.
12. I am one thing that your software needs to do.
13. The date that you need to deliver your final software on.
15. Iteration is ..... than a process.
16. The single most important output from your development process.
17. Software isn't complete until it has been .....

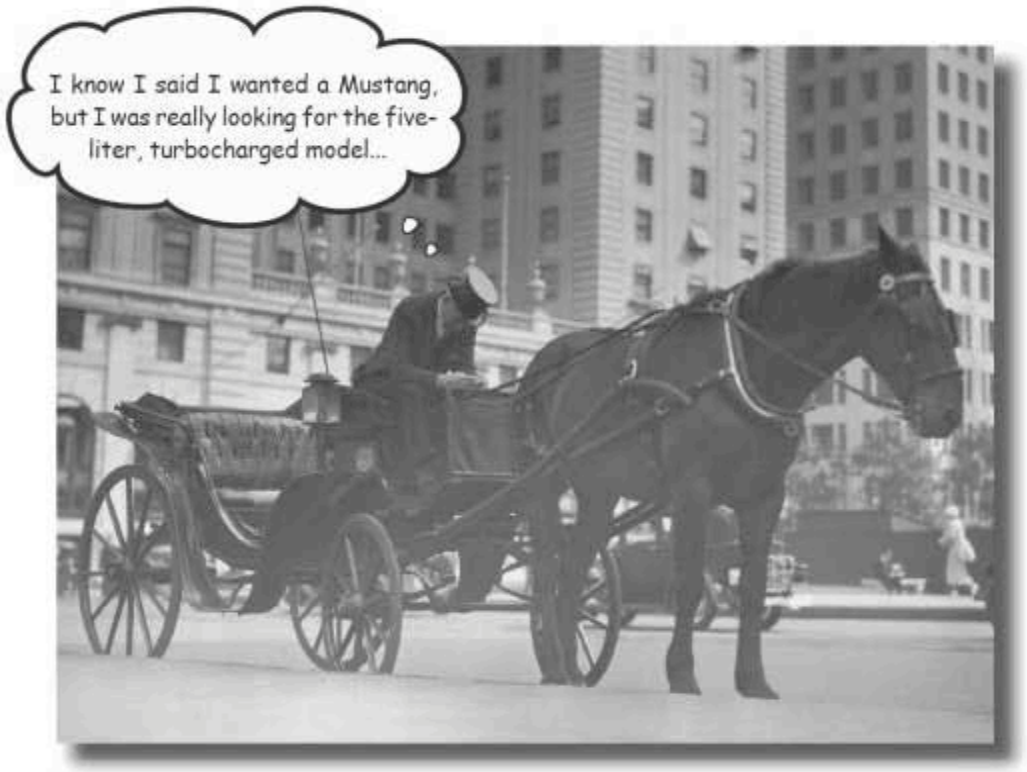
Down

1. A ..... is really just a sequence of steps.
3. When a project fails because it costs too much, it is .....
5. I contain every step of the software development process in micro and I result in runnable software.
8. The minimum number of iterations in a 3 month project.
10. Software that arrives when the customer needs it is .....
11. An iteration is a complete mini-.....
14. The types of software development projects where you should use iteration.

Software Development Cross Solution



Knowing what the customer wants



You can't always get what you want...but the customer should!

Great software development delivers **what the customer wants**. This chapter is all about **talking to the customer** to figure out what their **requirements** are for your software. You'll learn how **user stories**, **brainstorming**, and the **estimation game** help you get inside your customer's head. That way, by the time you finish your project, you'll be confident you've built what your customer wants...and not just a poor imitation.

Orion's Orbits is modernizing

Orion's Orbits provides quality space shuttle services to discerning clients, but their reservation system is a little behind the times, and they're ready to take the leap into the 21st century. With the next solar eclipse just four weeks away, they've laid out some serious cash to make sure their big project is done right, and finished on time.

Orion's doesn't have an experienced team of programmers on staff, though, so they've hired you and your team of software experts to handle developing their reservation system. It's up to you to get it right and deliver on time.

BRAIN POWER

How close do you think your final software will be to what the CEO of Orion Orbits wants?

Sharpen your pencil

Your job is to analyze the Orion's CEO's statement, and build some initial requirements. A requirement is a single thing that the software has to do. Write down the things you think you need to build for Orion's Orbits on the cards below.

Here's one to get you started.

Title: Show current deals

Description: The web site will show current deals to Orion's Orbits users.

Title:

Description:

Title:

Description:

Title:

Description:

Title:

Description:

Title:

Description:

Remember, each requirement should be a single thing the system has to do.

If you've got index cards, they're perfect for writing requirements down.

Sharpen your pencil Solution

Let's start by breaking out the requirements from what the Orion's Orbits CEO is asking for. Take his loose ideas and turn them into snippets, with each snippet capturing one thing that you think the software will need to do...

Each card captures one thing that the software will need to provide

there are no Dumb Questions

Q: Should we be using a specific format for writing these down?

A: No. Right now you're just grabbing and sorting out the ideas that your customer has and trying to get those ideas into some sort of manageable order.

Q: Aren't these requirements just user stories?

A: You're not far off, but at the moment they are just ideas. In just a few more pages we'll be developing them further into full-fledged user stories. At the moment it's just useful to write these ideas down somewhere.

Q: These descriptions all seem really blurry right now. Don't we need a bit more information before we can call them requirements?

A: Absolutely. There are lots of gaps in understanding in these descriptions. To fill in those gaps, we need to go back and talk to the customer some more...

Talk to your customer to get MORE information

There are always gaps in your understanding of what your software is supposed to do, especially early in your project. Each time you have more questions, or start to make assumptions, you need to go back and **talk with the customer** to get answers to your questions.

Here are a few questions you might have after your first meeting with the CEO:

1. How many different types of shuttles does the software have to support?
2. Should the software print out receipts or monthly reports (and what should be on the reports)?
3. Should the software allow reservations to be canceled or changed?
4. Does the software have an administrator interface for adding new types of shuttles, and/or new packages and deals?
5. Are there any other systems that your software is going to have to talk to, like credit card authorization systems or Air/Space Traffic Control?
6. .....

.....

Can you come up with another question you might want to ask the CEO?

OK, thanks for coming back to me. I'll get to those questions in just a bit, but I thought of something else I forgot to mention earlier...

Try to gather additional requirements.

Talking to the customer doesn't just give you a chance to get more details about *existing* requirements. You also want to find out about **additional requirements** the customer didn't think to tell you about earlier. There's nothing worse than finishing a project and the customer saying they forgot some important detail.

So how do you get the customer to think of everything you need to know, ***before*** you start building their software?

Bluesky with your customer

When you iterate with the customer on their requirements, **THINK BIG**. Brainstorm with other people; two heads are better than one, and ten heads are better than two, as long as everyone feels they can contribute without criticism. Don't rule out any ideas in the beginning—just capture *everything*. It's OK if you come up with some wild ideas, as long as you're all still focusing on the core needs that the software is trying to meet. This is called ***blueskying*** for requirements.

We call this blue skying because the sky's the limit

Watch it!

Avoid office politics.

Nothing will stifle creative bluesky thinking like a boss that won't let people speak up. Try as much as possible to leave job descriptions and other baggage at the door when blueskying requirements. Everyone should get an equal say to ensure you get the most out of each brainstorming session.

Sharpen your pencil

Take four of the ideas from the bluesky brainstorm and create a new card for each potential requirement. Also, see if you can come up with two additional requirements of your own.

We can refer to each requirement easily by using its title.

Title: Pay with Visa/MC/PayPal

Description: Users will be able to pay for their bookings by credit card.

Title:

Description:

Title:

Description:

Title:

Description:

Make these two your own.

Title:

Description:

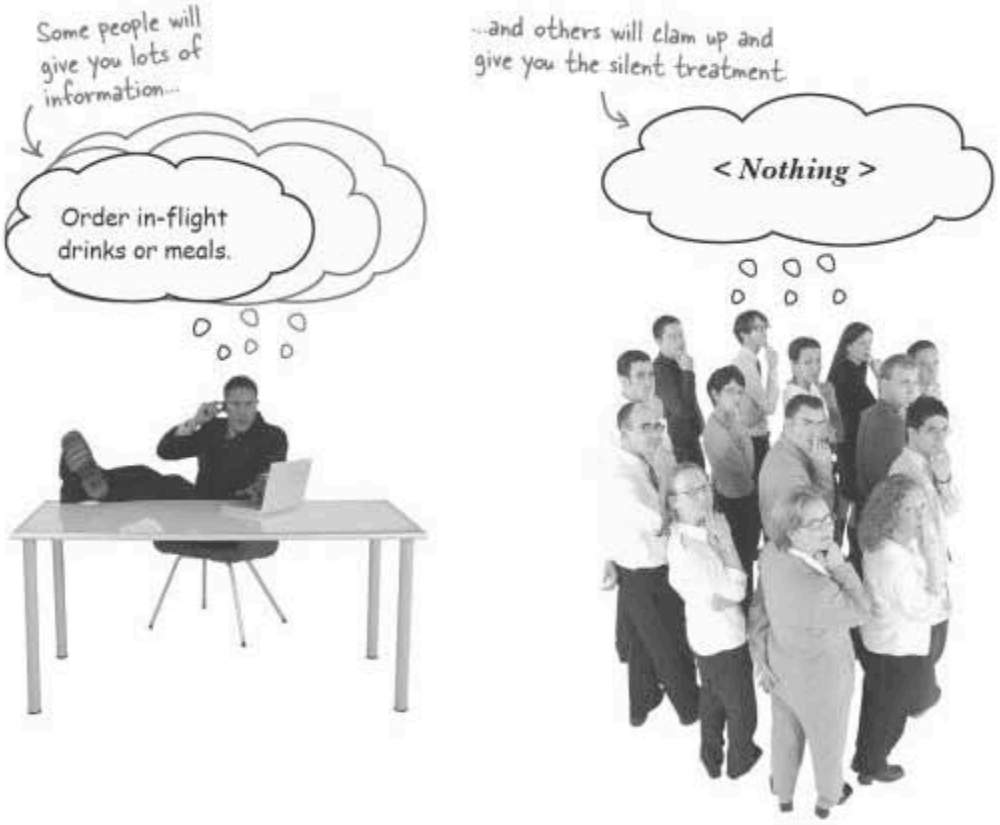
Title:

Description:

Answers on page 38.

Sometimes your bluesky session looks like this...

Sometimes, no matter how hard you try, your bluesky sessions can be as muffled as a foggy day in winter. Often the people that know what the software should really do are just not used to coming out of their shell in a brainstorming environment, and you end up with a long, silent afternoon.



The zen of good requirements

The key to capturing good requirements is to get as many of the stakeholders involved as possible. If getting everyone in the same room is just not working, have people brainstorm individually and then come together and put all their ideas on the board and brainstorm a bit more. Go away and think about what happened and come back together for a second meeting.

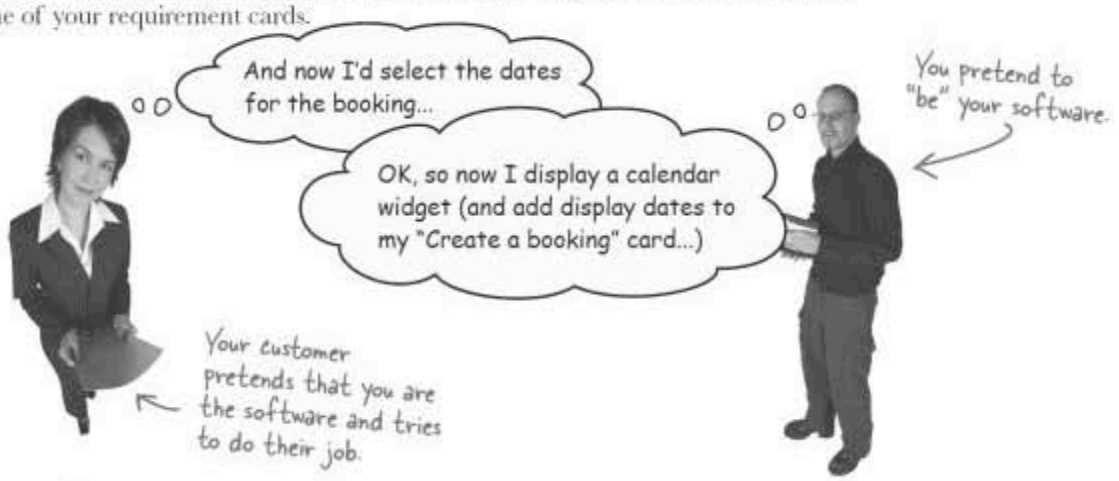
There are LOTS of ways to gather good requirements. If one approach doesn't work, simply TRY ANOTHER.

Find out what people REALLY do

Everything (that's ethical and legal) is pretty much fair game when you're trying to get into your customer's head to understand their requirements. Two particularly useful techniques that help you understand the customer are **role playing** and **observation**.

Role playing

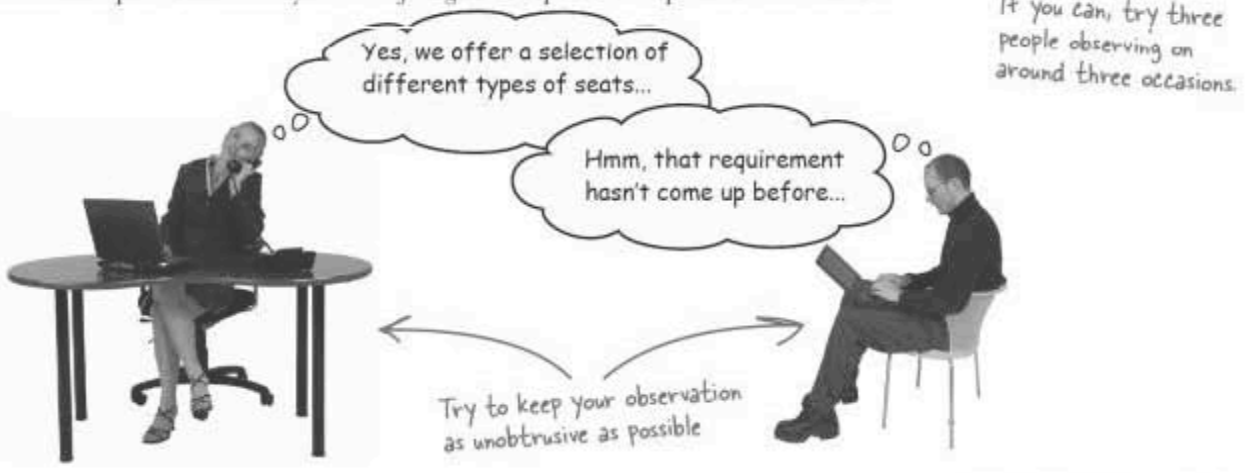
If your customer is finding it hard to visualize how they need their software to work, act it out. You pretend to be the software and your customer attempts to instruct you in what they would like you to do. Then write down each thing the software needs to do on one of your requirement cards.



Observation

Sometimes the best way to understand how people will work with your software is to watch them, and figure out where your software will fit in. Nothing beats firsthand evidence, and observation can really help to bring out constraints and details that might have been missed in bluesky brainstorming or even in role playing. Also, try to observe the same interactions more than once with multiple observers so you don't just gain one person's impression of an event.

If you can, try three people observing on around three occasions.



Our Sharpen your pencil Solution

Your job was to take each of the ideas from the bluesky session on page 35 and create a new card for each potential requirement.

You should also role play and observe.

A nonfunctional constraint, but it is still captured as a user story

Title: Support 3000 concurrent users
Description: The traffic for Orion's Orbits is expected to reach 3000 users, all using the site at the same time.

Title: Pay with Visa/MC/PayPal
Description: Users will be able to pay for their bookings by credit card or PayPal.

Title: Review flight
Description: A user will be able to leave a review for a shuttle flight they have been on.

We've added to our cards from page 32 after the brainstorming with the customer.

Title: Order Flight DVD
Description: A user will be able to order a DVD of a flight they have been on.

Title: Order in-flight meals
Description: A user will be able to specify the meals and drinks they want during a flight.

Title: Book a shuttle
Description: A user will be able to book a shuttle specifying the date and time of the flight.

These were the requirements we came up with; yours could have been different.

Title: Choose seating
Description: A user will be able to choose aisle or window seating.

Title: Use Ajax for the UI
Description: The user interface will use Ajax technologies to provide a cool and slick online experience.

And we've added more detail where it was uncovered through brainstorming, role playing, or observation.

The boss isn't sure he understands what this requirement is all about.

Your requirements must be CUSTOMER-oriented

A great requirement is actually written **from your customer's perspective** describing what the software is going to do **for the customer**. Any requirements that your customer doesn't understand are an immediate red flag, since they're not things that the customer could have possibly asked for.

A requirement should be written in the customer's language and read like a **user story**: a story about how their users interact with the software you're building. When deciding if you have good requirements or not, judge each one against the following criteria:

User stories SHOULD...

You should be able to check each box for each of your user stories.

- ☐ ... describe **one thing** that the software needs to do for the customer.
- ☐ ... be written using language that **the customer understands**.
- ☐ ... be **written by the customer**.
- ☐ ... be **short**. Aim for no more than three sentences.

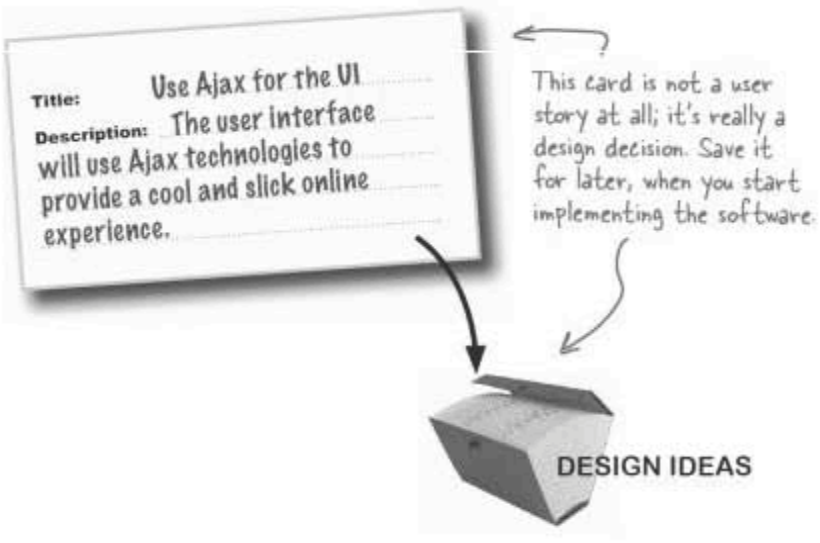
Think "by the customer, for the customer"

This means the customer drives each one, no matter who scribbles on a notecard.

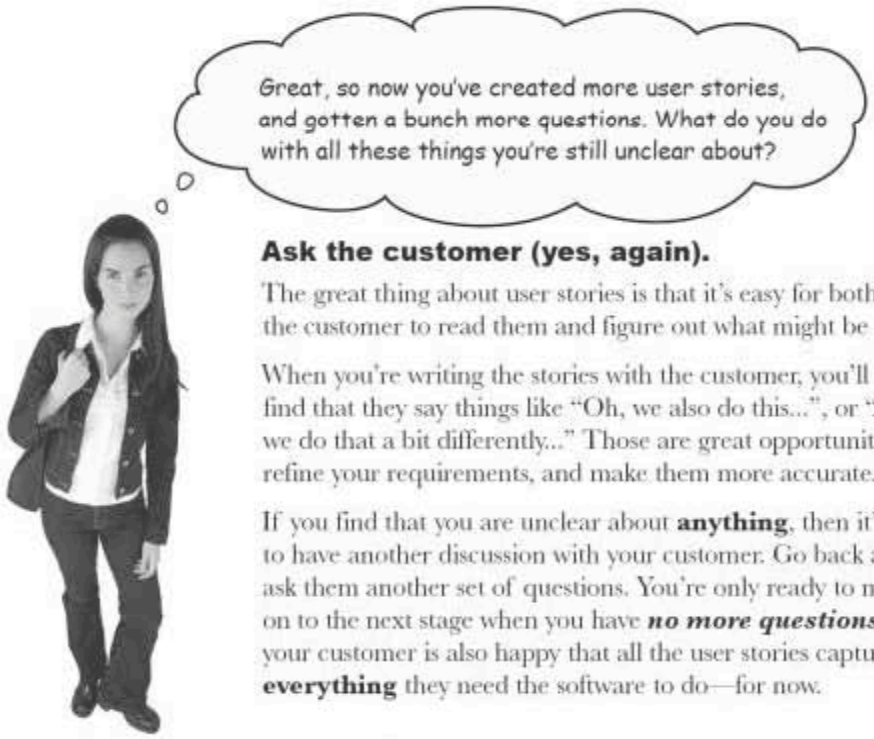
If a user story is long, you should try and break it up into multiple smaller user stories (see page 54 for tips).

User stories SHOULD NOT...

- ☐ ... be a long essay.
- ☐ ... use technical terms that are unfamiliar to the customer.
- ☐ ... mention specific technologies.



A user story is written from the CUSTOMER'S PERSPECTIVE. Both you AND your customer should understand what a user story means.



Ask the customer (yes, again).

The great thing about user stories is that it's easy for both you and the customer to read them and figure out what might be missing.

When you're writing the stories with the customer, you'll often find that they say things like "Oh, we also do this...", or "Actually, we do that a bit differently..." Those are great opportunities to refine your requirements, and make them more accurate.

If you find that you are unclear about **anything**, then it's time to have another discussion with your customer. Go back and ask them another set of questions. You're only ready to move on to the next stage when you have ***no more questions*** and your customer is also happy that all the user stories capture **everything** they need the software to do—for now.

there are no Dumb Questions

Q: What's the "Title" field on my user stories for? Doesn't my description field have all the information I need?

A: The title field is just a handy way to refer to a user story. It also gives everyone on the team the *same* handy way to refer to a story, so you don't have one developer talking about "Pay by PayPal," another saying, "Pay with credit card," and find out they mean the same thing later on (after they've both done needless work).

Q: Won't adding technical terms and some of my ideas on possible technologies to my user stories make them more useful to me and my team?

A: No, avoid tech terms or technologies at this point. Keep things in the language of the customer, and just describe what the software needs to do. Remember, the user stories are written from the customer's perspective. The customer has to tell you whether you've gotten the story right, so a bunch of tech terms will just confuse them (and possibly obscure whether your requirements are accurate or not).

If you do find that there are some possible technical decisions that you can start to add when writing your user stories, note those ideas down on another set of cards (cross referencing by title). When you get to coding, you can bring those ideas back up to help you at that point, when it's more appropriate.

Q: And I'm supposed to do all this refining of the requirements as user stories with the customer?

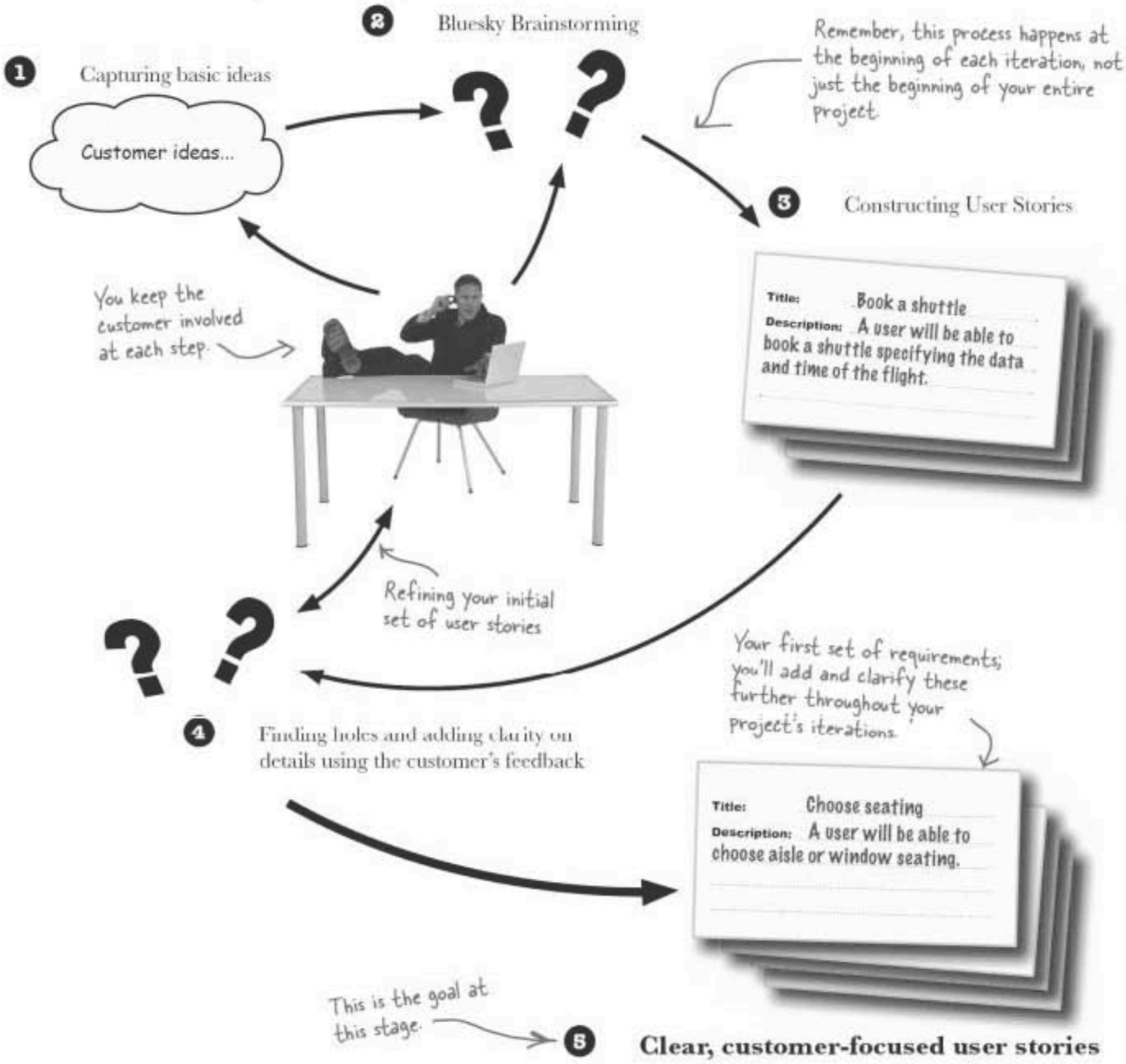
A: Yes, absolutely. After all, you're only ready for the next step when both you **and** the customer finally decide that you completely understand the software requirements. You can't make that decision on your own, so keeping your customer in the loop is essential.

Q: This seems like a lot of requirements work up front at the beginning of the project. What about when things change?

A: The work you've done so far is just your first attempt at gathering requirements at the beginning of your prject. You'll continue to refine and capture new requirements throughout your project, feeding those requirements where necessary into your project's iterations.

Develop your requirements with customer feedback

The steps we've followed so far have been all about coming to grips with the customer's ideas and refining those ideas into user stories. You execute these steps, in one form or another, at the beginning of each iteration to make sure that you always have the right set of features going into the next iteration. Let's see how that process currently looks...



User Story Exposed

This week's interview:
The many faces of a User Story

Head First: Hello there, User Story.

User Story: Hi! Sorry it's taken so long to get an interview, I'm a bit busy at the moment...

Head First: I can imagine, what with you and your friends capturing and updating the requirements for the software at the beginning of each iteration, you must have your hands pretty full.

User Story: Actually, I'm a lot busier than that. I not only describe the requirements, but I'm also the main technique for bridging the gap between what a customer wants in his head and what he receives in delivered software. I pretty much drive everything from here on in.

Head First: But don't you just record what the customer wants?

User Story: Man, I really wish that were the case. As it turns out, I'm pretty much at the heart of an entire project. Every bit of software a team develops has to implement a user story.

Head First: So that means you're the benchmark against which every piece of software that is developed is tested?

User Story: That means if it's not in a user story somewhere, it ain't in the software, period. As you can imagine, that means I'm kept busy all the way through the development cycle.

Head First: Okay, sure, but your job is essentially done after the requirements are set, right?

User Story: I wish. If there's anything I've learned, requirements never stay the same in the real world. I might change right up to the end of a project.

Head First: So how do you handle all this pressure and still keep it together?

User Story: Well, I focus on one single thing: describing what the software needs to do from the customer's perspective. I don't get distracted by the rest of the noise buzzing around the project, I just keep that one mantra in my head. Then everything else tends to fall into place.

Head First: Sounds like a big job, still.

User Story: Ah, it's not too bad. I'm not very sophisticated, you know? Just three lines or so of description and I'm done. The customers like me because I'm simple and in their language, and the developers like me because I'm just a neat description of what their software has to do. Everyone wins.

Head First: What about when things get a bit more formal, like with use cases, main and alternate flows, that sort of thing? You're not really used then, are you?

User Story: Heck, I can smarten myself up with some more details to be a use case if that's what you need, and lots of people do dress me up that way for their bosses. The important thing is that we all describe what a piece of software needs to do, no matter how we look. Use cases are more or less user stories in a tuxedo.

Head First: Well, you heard it here first folks. Next week we'll be catching up with Test to see how he guarantees that software does what a user story requires. Until then, take care and remember, always do only what your user story says, and not an ounce more!

User stories define the WHAT of your project... estimates define the WHEN

After your initial requirement-capture stage you will have a set of clear, customer-focused user stories that you and the customer believe capture **WHAT** it is you're trying to build, at least for the first iteration or so. But don't get too comfortable, because the customer will want to know **WHEN** all those stories will be built.

This is the part where the customer asks the big question: **How long will it all take?**

Your project estimate is the sum of the estimates for your user stories

To figure out how long it will take to complete all of the requirements captured in your user stories, you need to use a two-step process.

You need to:

If you can get this figured out...

- ☐ Add an estimate to each user story for how long you think it will take to develop (that is, design, code, test, and deliver) that functionality.
- ☐ Add up all the estimates to get a total estimate for how long your project will take to deliver the required software.

...then this will be a piece of cake.

Welcome to the Orion's Orbits Development Diner. Below is the menu...your job is to choose your options for each dish, and come up with an estimate for that dish—ahem—user story. You'll also want to note down any assumptions you made in your calculations.

Entrées

Pay Credit Card or Paypal

Visa	**2 days**
Mastercard	**2 days**
PayPal	**2 days**
American Express	**5 days**
Discover	**4 days**

Order Flight DVD

Stock titles with standard definition video	**2 days**
Provide custom titles	**5 days**
High Definition video	**5 days**

Choose Seating

Choose aisle or window seat	**2 days**
Choose actual seat on shuttle	**10 days**

Order In-Flight Meals

Select from list of three meals & three drinks	**5 days**
Allow special dietary needs (Vegetarian, Vegan)	**2 days**

Desserts

Create Flight Review

Create a review online	**3 days**
Submit a review by email	**5 days**

Estimate for each user story in days

Assumptions?

Title: Pay with Visa/MC/PayPal
Description: Users will be able to pay for their bookings by credit card or PayPal.

Write your estimate for the user story here.

Title: Order Flight DVD
Description: A user will be able to order a DVD of a flight they have been on.

Jot down any assumptions you think you're making in your estimate.

Title: Choose seating
Description: A user will be able to choose aisle or window seating.

Title: Order in-flight meals
Description: A user will be able to specify the meals and drinks they want during a flight.

Title: Review flight
Description: A user will be able to leave a review for a shuttle flight they have been on.

What did you come up with? Rewrite your estimates here. Bob and Laura also did estimates...how did yours compare to theirs?

Put your estimates here.

	Your estimates	Bob's estimates	Laura's estimates
Title: Pay with Visa/MC/PayPal		15	10
Title: Order Flight DVD		20	2
Title: Choose seating		12	2
Title: Order in-flight meals		2	7
Title: Review flight		3	3

Well, at least we seem to agree here...

It looks like everyone has a different idea for how long each user story is going to take. Which estimates do you think are **RIGHT**?

Cubicle conversation

Laura: Well, let's start with the first user story. How did you come up with 10 days?

Bob: That's easy, I just picked the most popular credit cards I could think of, and added time to support PayPal...

Laura: But lots of high-end executives only use American Express, so my assumption was that we'd have to cope with that card, too, not just Visa and MasterCard.

Bob: Okay, but I'm still not feeling entirely happy with that. Just that one assumption is making a really big difference on how long it will take to develop that user story...

Laura: I know, but what can you do, we don't know what the customer expects...

Bob: But look at this...you came up with 20 days for "Ordering a Flight DVD," but even with all the options, that should be 14 days, max!

Laura: I was actually being on the conservative side. The problem is that creating a DVD is a completely new feature, something I haven't done before. I was factoring in overhead for researching how to create DVDs, installing software, and getting everything tested. Everything I thought I'd need to do to get that software written. So it came out a lot higher.

Bob: Wow, I hadn't even thought of those things. I just assumed that they'd been thought of and included. I wonder if the rest of the estimates included tasks like research and software installation?

Laura: In my experience, probably not. That's why I cover my back.

Bob: But then ***all*** of our estimates could be off...

Laura: Well, at least we agree on the "Create a Flight Review" story. That's something.

Bob: Yeah, but I even had assumptions I made there, and that still doesn't take into account some of that overhead you were talking about.

Laura: So all we have are a bunch of estimates we don't feel that confident about. How are we going to come up with a number for the project that we believe when we don't even know what everyone's assumptions are?

Getting rid of assumptions is the most important activity for coming up with estimates you believe in.

Playing planning poker

To come up with accurate estimates, you need to get rid of all those assumptions that put your estimates at risk of being wrong. You want a set of estimates that **everyone believes in** and are confident that they can deliver, or at the very least you want a set of estimates that let you know what assumptions everyone is making before you sign on the dotted line. It's time to grab everyone that's going to be involved in estimating your user stories, sit them around a table, and get ready to play "planning poker."

1

Place a user story in the middle of the table
This focuses everyone on a specific user story so they can get their heads around what their estimates and assumptions might be.

Title: Pay with Visa/MC/PayPal

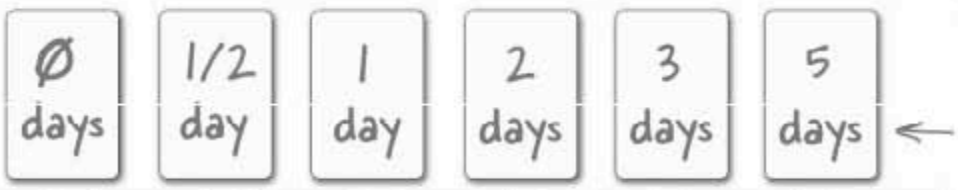
Description: Users will be able to pay for their bookings by credit card or PayPal.

We want a solid estimate for how long it will take to develop this story. Don't forget that development should include designing, coding, testing, and delivering the user story.

2

Everyone is given a deck of 13 cards. Each card has an estimate written on one side.
You only need a small deck, just enough to give people several options:

This card means "It's already done."



All of these estimates are developer-days (for instance, two man-days split between two workers is still two days).

Everyone has each of these cards.

Hmmm...any thoughts on what it means if someone plays one of these cards for their estimate?

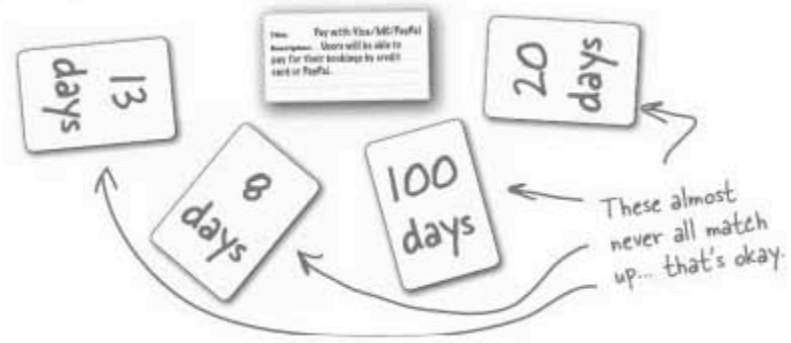
Don't have enough info to estimate? You might consider using this card.

If any player uses this card, you need to take a break from estimating for a bit

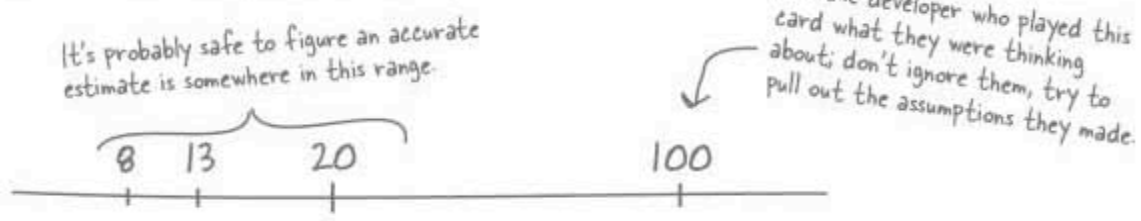
3 **Everyone picks an estimate for the user story and places the corresponding card face down on the table.**
You pick the card that you think is a reasonable estimate for the user story. Don't discuss that estimate with anyone else, though.

Make sure your estimate is for the whole user story, not just a part of it

4 **Everyone then turns over their cards at exactly the same time.**
Each player at the table shows their hand, which gives their honest estimate for the user story.



5 **The dealer marks down the spread across each of the estimates.**
Whoever is running the game notes the spread across each of the estimates that are on the cards. Then you do a little analysis:



The larger the difference between the estimates, the less confident you are in the estimate, and the more assumptions you need to root out.

How does this help with assumptions? And what about that guy who chose 100? We can't just ignore him, can we?

Large spreads can be a misunderstanding

When you see large gaps between the estimates on a particular user story's spread, something is probably missing. It could be that some of your team misunderstood the user story, in which case it's time to revisit that story. Or it could be that some members of your team are just unsure of something that another part of your team is completely happy with.

In either case, it's time to look at the assumptions that your team is making and decide if you need to go back and speak to the customer to get some more feedback and clarification on your user stories—and the assumptions you're making about them.

In fact, even if everyone's estimate is within the same narrow range, it's worth asking for everyone's assumptions to make sure that EVERYONE is not making the same wrong assumption. It's unlikely that they are, but just in case, always discuss and document your assumptions after every round of planning poker.

Try writing your assumptions on the back of your user story cards.

Put assumptions on trial for their lives

When it comes to requirements, ***no assumption is a good assumption***. So whenever planing poker turns up your team's assumptions, don't let that assumption into your project without first doing everything you can to ***beat it out*** of your project...

Put every assumption on trial

You're aiming for as few assumptions as possible when making your estimates. When an assumption rears its head in planning poker, even if your entire team shares the assumption, expect that assumption to be wrong ***until it is clarified by the customer****.*

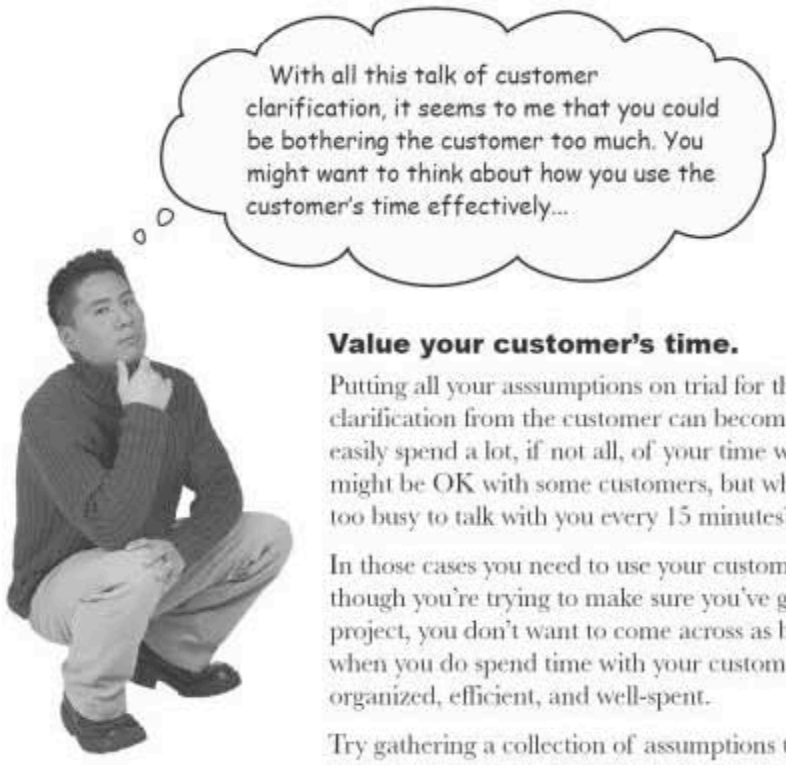
As opposed to not knowing what you don't know...

At least you know what you don't know

No matter how hard you try, some assumptions really will survive clarification with the customer. That's OK. Sometimes the customer doesn't have a great answer to a particular assumption at the beginning of a project, and in those cases you need to live with the assumption. The important thing is that you know that there is an assumption being made, and you can write it down as a risk for that user story (like on the back of your user story card). This helps you keep an eye on and track your risks, knocking them out at a later stage in your project.

Depending on customer priority, you might even decide to delay the development of a user story that has a number of surviving assumptions until they can be clarified.

While you can't always get rid of all assumptions, the goal during estimation is to eliminate as many assumptions as possible by clarifying those assumptions with the customer. Any surviving assumptions then become risks.



Value your customer's time.

Putting all your asssumptions on trial for their life and seeking clarification from the customer can become a lot of work. You can easily spend a lot, if not all, of your time with your customer. That might be OK with some customers, but what about the ones that are too busy to talk with you every 15 minutes?

In those cases you need to use your customer's time carefully. Even though you're trying to make sure you've gotten things right on their project, you don't want to come across as being not up to the job. So when you do spend time with your customer, make sure that time is organized, efficient, and well-spent.

Try gathering a collection of assumptions together and then clarifying those all at once with the customer. Rather than bothering the customer at the end of every round of planing poker, schedule an **assumption-busting session** where you take in the collected assumptions and try to blast as many of them away as possible.

Once you have your answers, head back for a final round of planning poker.

Once you've gotten a significant number of your assumptions beaten out in your assumption-busting session with the customer, it's time to head back and play a final round of planning poker so that you and your team can come up with estimates that factor in the new clarifications.

there are no Dumb Questions

Q: Why is there a gap between 40 and 100 days on the planning poker cards?

A: Well, the fact is that 40 is a pretty large estimate, so whether you feel that the estimate should be 41 or even 30 days is not really important at this point. 40 just says that you think there's a lot to do in this user story, and you're just on the boundary of not being able to estimate this user story at all...

Q: 100 days seems *really* long; that's around half a year in work time! Why have 100 days on the cards at all?

A: Absolutely, 100 days is a **very** long time. If someone turns up a 100-days card then there's something seriously misunderstood or wrong with the user story. If you find that it's the user story that's simply too long, then it's time to break that user story up into smaller, more easily estimatable stories.

Q: What about the question-mark card? What does that mean?

A: That you simply don't feel that you have enough information to estimate this user story. Either you've misunderstood something, or your assumptions are so big that you don't have any confidence that *any* estimate you place down on the table could be right.

Q: Some people are just bound to pick nutty numbers. What do I do about them?

A: Good question. First, look at the trends in that individual's estimates to see if they really are being "nutty," or whether they in fact tend to be right! However, some people really are inclined to just pick extremely high or very low numbers most of the time and get caught up in the game. However, every estimate, particularly ones that are out of whack with the rest of the player's estimates, should come under scrutiny after every round to highlight the assumptions that are driving those estimates.

After a few rounds where you start to realize that those wacky estimates are not really backed up by good assumptions, you can either think about removing those people from the table, or just having a quiet word with them about why they always insist on being off in left field.

Q: Should we be thinking about who implements a user story when coming up with our estimates?

A: No, every player estimates how long they think it will take for them to develop and deliver the software that implements the user story. At estimation time you can't be sure who is going to actually implement a particular user story, so you're trying to get a feel for the capability of anyone on your team to deliver that user story.

Of course, if one particular user story is perfect for one particular person's skills, then they are likely to estimate it quite low. But this low estimate is balanced by the rest of your team, who should each assume that they are individually going to implement that user story.

In the end, the goal is to come up with an estimate that states "We as a team are all confident that this is how long it will take any one of us to develop this user story."

Q: Each estimate is considering more than just implementation time though, right?

A: Yes. Each player should factor in how much time it will take them to develop and deliver the software including any other deliverables that they think might be needed. This could include documentation, testing, packaging, deployment—basically everything that needs to be done to develop and deliver the software that meets the user story.

If you're not sure what other deliverables might be needed, then that's an assumption, and might be a question for the customer.

Q: What if my team all agree on exactly the same estimate when the cards are turned over. Do I need to worry about assumptions?

A: Yes, for sure. Even if everyone agrees, it's possible that everyone is making the same wrong assumptions. A large spread of different estimates indicates that there is more work to be done and that your team is making different and possibly large assumptions in their estimates. A tiny spread says that your team might be making the same assumptions in error, so examining assumptions is critical regardless of the output from planning poker.

It's important to get any and all assumptions out in the open ***regardless*** of what the spread says, so that you can clarify those assumptions right up front and keep your confidence in your estimates as high as possible.

Don't make assumptions about your assumptions... talk about EVERYTHING.

A BIG user story estimate is a BAD user story estimate

We all agree, we don't need any more information. This user story will take 40 days to develop...

Your user story is too big.

Remember this from Chapter 1?

40 days is a long time, and lots can change. Remember, 40 days is ***2 months*** of work time.

An ***entire iteration*** should ideally be around **1 calendar month** in duration. Take out weekends and holidays, and that's about 20 working days. If your estimate is 40 days for just *one* user story, then it won't even fit in one iteration of development unless you have two people working on it!

If you have to have long estimates like this, then you need to be talking as a team as often as possible. We'll get to that in a few pages.

As a rule of thumb, estimates that are longer than 15 days are *much less likely* to be accurate than estimates below 15 days.

In fact, some people believe that estimates longer than seven days should be double-checked.

When a user story's estimate breaks the 15-day rule you can either:

1. **Break your stories into smaller, more easily estimated stories**
   Apply the AND rule. Any user story that has an "and" in its title or description can probably be split into two or more smaller user stories.

2. **Talk to your customer...again.** Starting to sense a pattern?
   Maybe there are some assumptions that are pushing your estimate out. If the customer could clarify things, those assumptions might go away, and cut down your estimates significantly.

Estimates greater than 15 days per user story allow too much room for error.

When an estimate is too long, apply the AND rule to break the user story into smaller pieces.

Exercise

The two user stories below resulted in estimates that broke the 15-day rule. Take the two user stories and apply the AND rule to them to break them into smaller, more accurately estimatable stories.

Title: Choose seating

Description: A user will choose aisle or window seating, be able to select the seat they want, and change that seat up to 24 hours before the flight.

Title:

Description:

Title:

Description:

Title: Order in-flight meals

Description: A user will choose which meal option they want, from a choice of three, and be able to indicate if they are vegetarian or vegan.

Title:

Description:

Title:

Description:

Title:

Description:

Title:

Description:

Exercise Solution

Your job was to take the longer user stories at the top of each column and turn them into smaller, easily estimatable user stories.

Title: Choose seating
Description: A user will choose aisle or window seating, be able to select the seat they want, and change that seat up to 24 hours before the flight.

Title: Order in-flight meals
Description: A user will choose which meal option they want, from a choice of three, and be able to indicate if they are vegetarian or vegan.

Title: Choose aisle/window seat
Description: A user can choose either aisle or window seating.

Title: Select from meal options
Description: A user can choose the meal they want from a set of three meal options.

Title: Choose specific seat
Description: A user can choose the actual seat that they want for a shuttle flight.

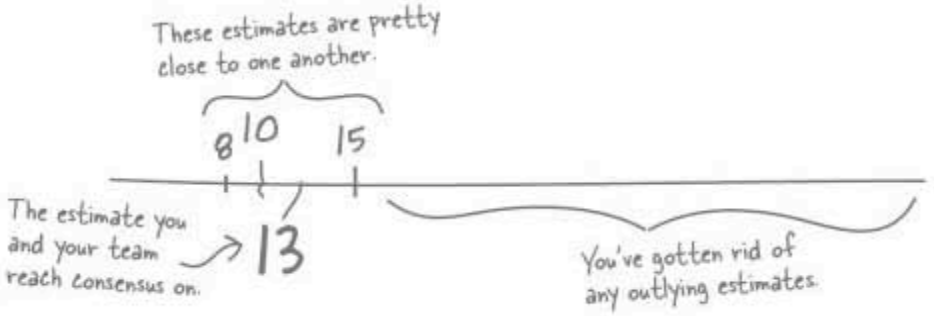
Title: Specify vegetarian meal
Description: A user will be able to indicate that they are vegetarian when selecting their meal options.

Title: Change seating
Description: A user can change their seat up to 24 hours before launch, provided other seat options are available.

Title: Specify vegan meal
Description: A user will be able to indicate that they are vegan when selecting their meal options.

The goal is convergence

After a solid round of planning poker, you should not only have estimates for each user story but be *confident* in those estimates. The goal now is to get rid of as many assumptions as possible, and to **converge** all of the points on each user story's spread of estimates.



Run through this cycle of steps till you reach a consensus:

1. **Talk to the customer**
   First and foremost, get as much information and remove as many assumptions and misunderstandings as possible by talking to your customer.

2. **Play planning poker**
   Play planning poker with each of your user stories to uproot any hidden assumptions. You'll quickly learn how confident you are that you can estimate the work that needs to be done.

3. **Clarify your assumptions**
   Using the results of planning poker, you'll be able to see where your team may have misunderstood the user stories, and where additional clarification is needed.

4. **Come to a consensus**
   Once everyone's estimates are close, agree on a figure for the user story's estimate.

Head back to Step 1 if you find assumptions that only the customer can answer.

It can also be useful to note the low, converged, and high estimates to give you an idea of the best and worst case scenarios.

How close is "close enough"?

Deciding when your estimates are close enough for consensus is really up to you. When you feel **confident in an estimate**, and you're **comfortable with the assumptions** that have been made, then it's time to write that estimate down on your user story card and move on.

there are no Dumb Questions

Q: How can I tell when my estimates are close enough, and have really converged?

A: Estimates are all about confidence. You have a good estimate if you and your team are truly confident that you can deliver the user story's functionality within the estimate.

Q: I have a number of assumptions, but I still feel confident in my estimate. Is that okay?

A: Really, you should have no assumptions in your user stories or in you and your team's understanding of the customer's requirements.

Every assumption is an opportunity to hit unexpected problems as you develop your software. Worse than that, every assumption increases the chances that your software development work will be delayed and might not even deliver what was required.

Even if you're feeling relatively confident, knock out as many of those assumptions as you possibly can by speaking to your team and, most importantly, speaking to your customer.

With a ***zero-tolerance attitude to assumptions***, you'll be on a much more secure path to delivering your customer the software that's needed, on time and on budget. However, you will probably always have some assumptions that survive the estimation process. This is OK, as assumptions are then turned into risks that are noted and tracked, and at least you are aware of those risks.

Your estimates are your PROMISE to your customer about how long it will take you and your team to DELIVER.

Q: I'm finding it hard to come up with an estimate for my user story, is there a way I can better understand a user story to come up with better initial estimates?

A: First, if your user story is complicated, then it may be too big to estimate confidently. Break up complex stories into simpler ones using the AND rule or common sense.

Sometimes a user story is just a bit blurry and complicated. When that happens, try breaking the user story into tasks in your head—or even on a bit of paper—you've got next to you at your planning poker sessions.

Think about the jobs that will be needed to be done to build that piece of software. Imagine you are doing those jobs, figure out how long you would take to do each one, and then add them all up to give you an estimate for that user story.

Q: How much of this process should my customer actually see?

A: Your customer should only see and hear your questions, and then of course your user stories as they develop. In particular, your customer is ***not*** involved in the planning poker game. Customers will want lower-than-reasonable estimates, and can pressure you and your team to get overly aggressive.

When there is a question about what a piece of the software is supposed to do in a given situation, or when an assumption is found, then involving the customer is absolutely critical. When you find a technical assumption being made by your team that you can clarify without the customer, then you don't have to go back and bother them with details they probably won't understand anyway.

But when you're playing planning poker, you are coming up with estimates of how long ***you*** believe that ***your team*** will take to develop and deliver the software. So it's ***your*** neck on the line, and ***your*** promise. So the customer shouldn't be coming up with those for you.

Who am I?

A bunch of techniques for working with requirements, in full costume, are playing a party game, "Who am I?" They'll give you a clue and then you try to guess who they are based on what they say. Assume they always tell the truth about themselves. Fill in the blanks next to each statement with the name (or names) of each attendee that the statement is true for. Attendees may be used in more than one answer

Tonight's attendees:

Blueskying – Role playing – Observation
User story – Estimate – Planning poker

You can dress me up as a use case for a formal occasion. .....

The more of me there are, the clearer things become. .....

I help you capture EVERYTHING. .....

I help you get more from the customer. .....

In court, I'd be admissible as firsthand evidence. .....

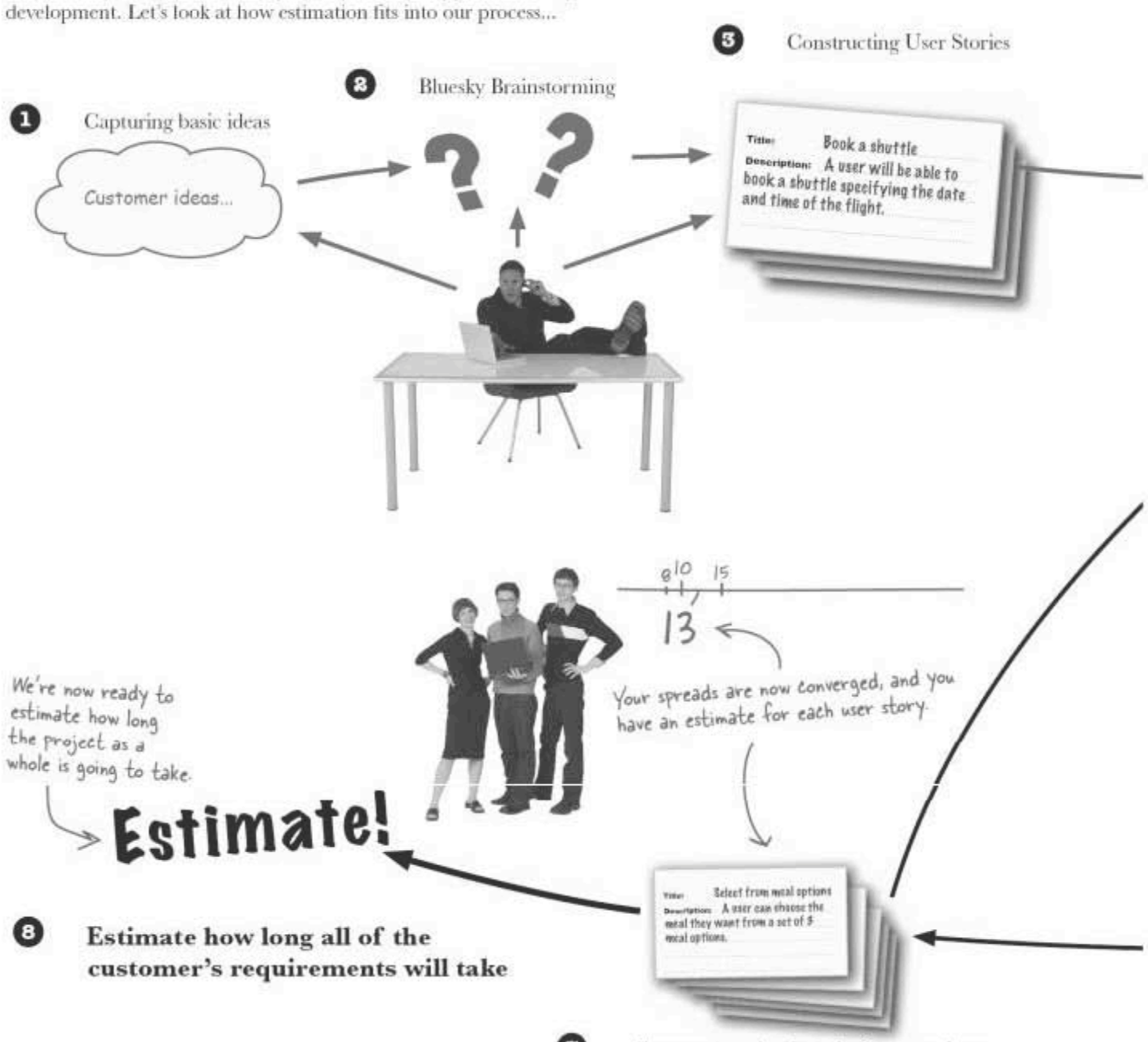
Some people say I'm arrogant, but really I'm just about confidence. .....

Everyone's involved when it comes to me. .....

→ Answers on page 62.

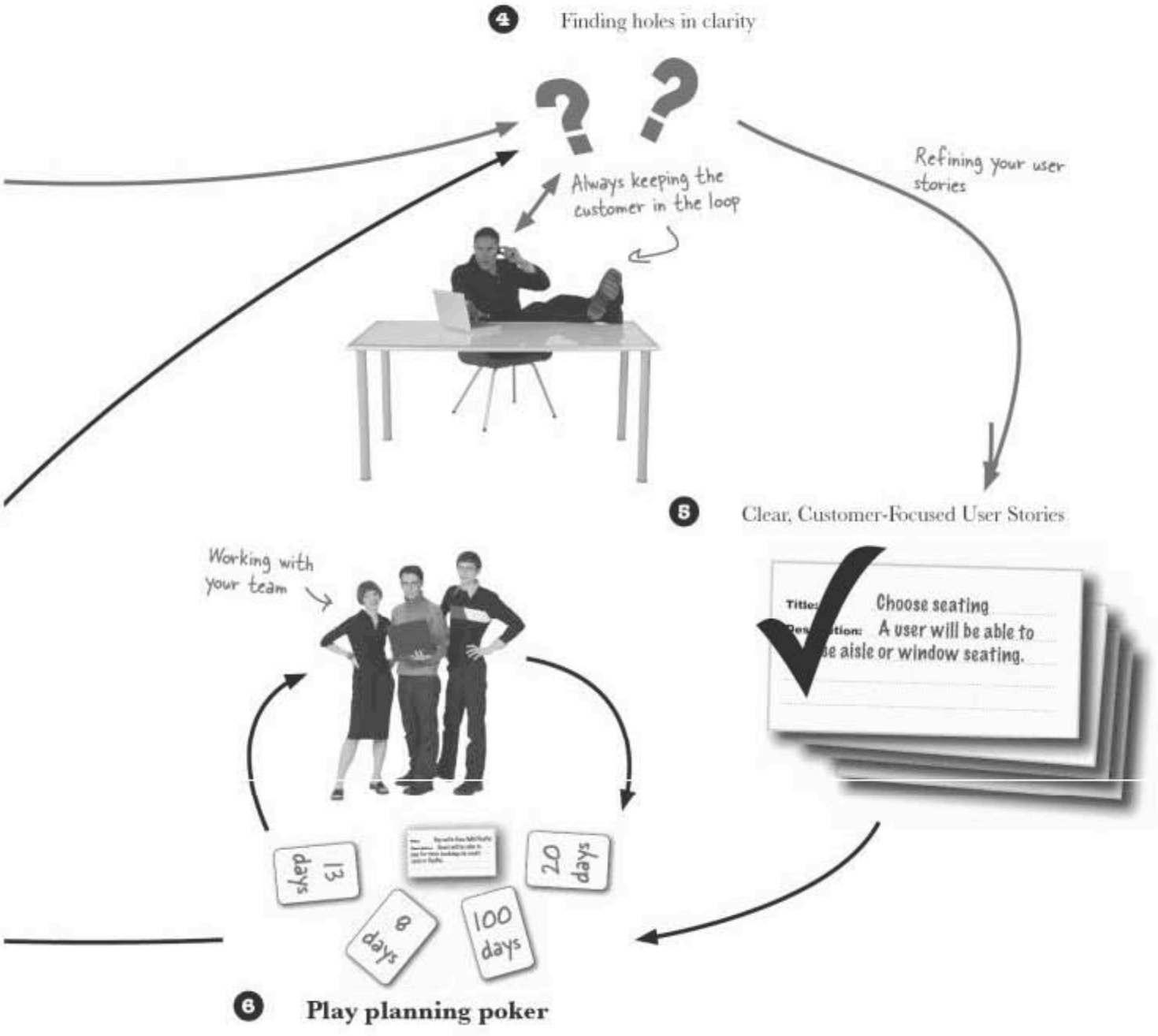
The requirement to estimate iteration cycle

We've now added some new steps in our iterative approach to requirements development. Let's look at how estimation fits into our process...



8 **Estimate how long all of the customer's requirements will take**

7 **Get any missing information from the customer, and break up large user stories**


4 Finding holes in clarity
Always keeping the customer in the loop
Refining your user stories
5 Clear, Customer-Focused User Stories
Title: Choose seating
Description: A user will be able to
be aisle or window seating.
Working with your team
13 days
8 days
100 days
20 days
6 Play planning poker

Who am I? Solutions

A bunch of techniques for working with requirements, in full costume, are playing a party game, "Who am I?" They'll give you a clue and then you try to guess who they are based on what they say. Assume they always tell the truth about themselves. Fill in the blanks next to each statement with the name (or names) of each attendee that the statement is true for. Attendees may be used in more than one answer

Tonight's attendees:

Blueskying – Role playing – Observation
User story – Estimate – Planning poker

You can dress me up as a use case for a formal occasion. ..... User Story .....

The more of me there are, the clearer things become. ..... User Story .....

I help you capture EVERYTHING. ..... Blueskying, Observation .....

I help you get more from the customer. ..... Role playing, Observation .....

In court, I'd be admissible as firsthand evidence. ..... Observation .....

Some people say I'm arrogant, but really I'm just about confidence. ..... Estimate .....

Everyone's involved when it comes to me. ..... Blueskying .....

Did you say planning poker? Customers aren't involved in that activity.

Finally, you're ready to estimate the whole project...

You've got short, focused user stories, and you've played planning poker on each story. You've dealt with all the assumptions that you and your team were making in your estimates, and now you have a set of estimates that you all believe in. It's time to get back to the customer with your total project estimate...

You've got an estimate for each story now.

- [x] Add an estimate to each user story for how long you think it will take to develop that functionality.
- [ ] Add up all the estimates to get a **total estimate** for how long your project will take to deliver the required software.

Now you can get a total estimate.

And the total project estimate is...

Add up the each of the converged estimates for your user stories, and you will find the total duration for your project, if you were to develop everything the customer wants.

15 16 20 19 12 15

Sum of user story estimates

= 489 days!

What do you do when your estimates are WAY too long?

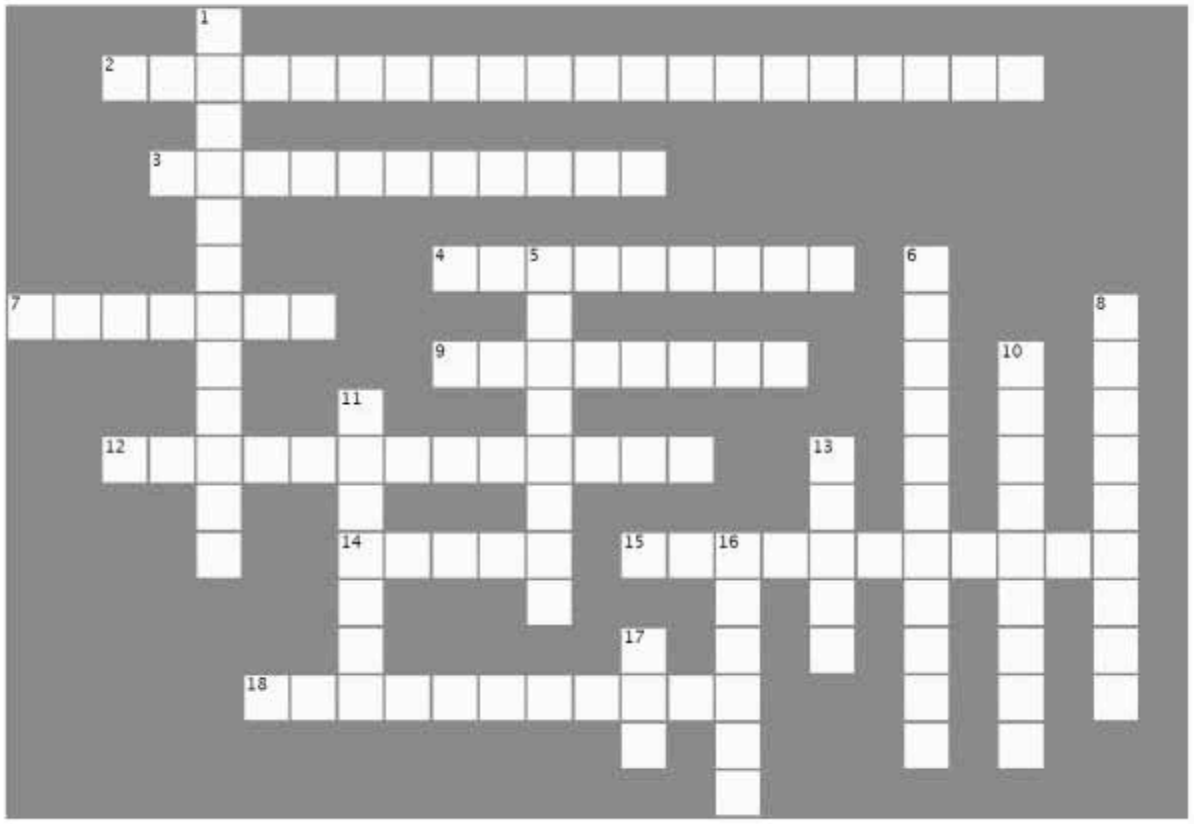
You've finally got an estimate you believe in, and that takes into account all the requirements that the customer wants. But you've ended up with a monster project that is just going to take too long.

Is it time to go back to the drawing board? Do you admit defeat and hand the work over to someone else? Or do you just ask the customer how long he thinks would work, forgetting about all your hard work to come up with you estimates in the first place?

You'll have to solve a crossword puzzle and work your way to Chapter 3 to find out how to get Orion's Orbits back on track.

Requirements and Estimation Cross

Let's put what you've learned to use and stretch out your left brain a bit. All of the words below are somewhere in this chapter: Good luck!



Across

2. When you and the customer are really letting your ideas run wild you are ....
3. When coming up with estimates, you are trying to get rid of as many ..... as possible.
4. None of this language is allowed in a user story.
7. If a requirement is the what, an estimate is the ....
9. Requirements are oriented towards the .....
12. The best way to get honest estimates and highlight assumptions.
14. A User Story is made up of a ..... and a description.
15. ..... is a great way of getting first hand evidence of exactly how your customer works at the moment.
18. The goal of estimation is .....

Down

1. When you just have no idea how to estimate a user story, use a card with this on it.
5. User stories are written from the perspective of the .....
6. When you and the customer act out a particular user story, you are ....
8. When everyone agrees on an estimate, it is called a .....
10. An estimate is good when eveyone on your team is feeling .....
11. The maximum number of days that a good estimate should be for one user story.
13. A great user story is about ..... lines long.
16. After a round of planning poker, you plot all of the estimates on a .....
17. You can use the ..... rule for breaking up large user stories.

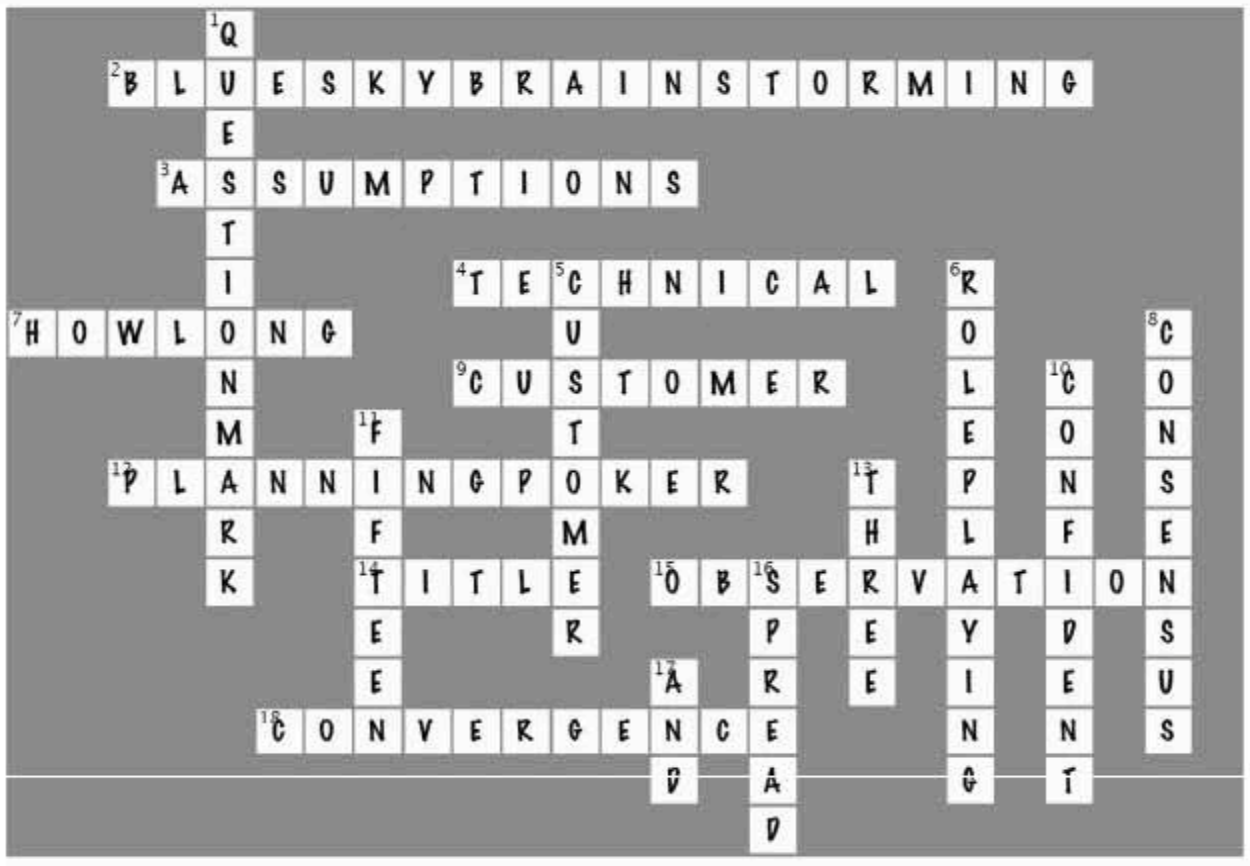
Tools for your Software Development Toolbox

Software Development is all about developing and delivering the software that the customer actually wants. In this chapter, you learned about several techniques to help you get inside the customer's head and capture the requirements that represent what they really want... For a complete list of tools in the book, see Appendix ii.

BULLET POINTS

- **Blueskying** gets your customer to **think big** when coming up with their requirements.
- A **user story** captures one interaction with the software from the customer's perspective.
- User stories should be **short**, around three sentences in length.
- A short user story is an **estimatable user story**.
- A user story should not take one developer more than 15 days to deliver.
- **Iteratively develop** your requirements with your customer to keep them in the loop at every step of the process.

Requirements and Estimation Cross Solution



Planning for success

Every great piece of software starts with a great plan.

In this chapter you're going to learn how to create that plan. You're going to learn how to work with the customer to **prioritize their requirements**. You'll **define iterations** that you and your team can then work toward. Finally you'll create an achievable **development plan** that you and your team can confidently **execute** and **monitor.** By the time you're done, you'll know exactly how to get from requirements to your first deliverable.

Customers want their software NOW!

Customers want their software **when they need it**, and not a moment later. You've come to grips with the customer's ideas using brainstorming, you've got a set of user stories that describe everything the customer might need the software to do, and you've even added an estimate to each user story that helped you figure out how long it will take to deliver everything the customer wants. The problem is, developing **everything** the customer said they needed will take **too long**...

Our Estimate

489 days

The total after summing up all the estimates for your user stories

What the customer wants

90 days!

Orion's Orbits still wants to modernize their booking system; they just can't wait almost two years for the software to get finished. Take the following snippets from the Orion's Orbits user stories, along with their estimates, and circle the ones you think you should develop to come up with a chunk of work that will take **no longer than 90 days**.

Title: Book a shuttle
Estimate: 15 days

Title: Pay with Visa/MC/PayPal
Estimate: 15 days

Title: Review flight
Estimate: 13 days

Title: Order in-flight meals
Estimate: 13 days

Title: Order Flight DVD
Estimate: 12 days

Title: Book Segway in spaceport transport
Estimate: 15 days

Title: View Space Miles Account
Estimate: 14 days

Title: Choose seating
Estimate: 12 days

Title: Apply for "frequent astronaut" card
Estimate: 14 days

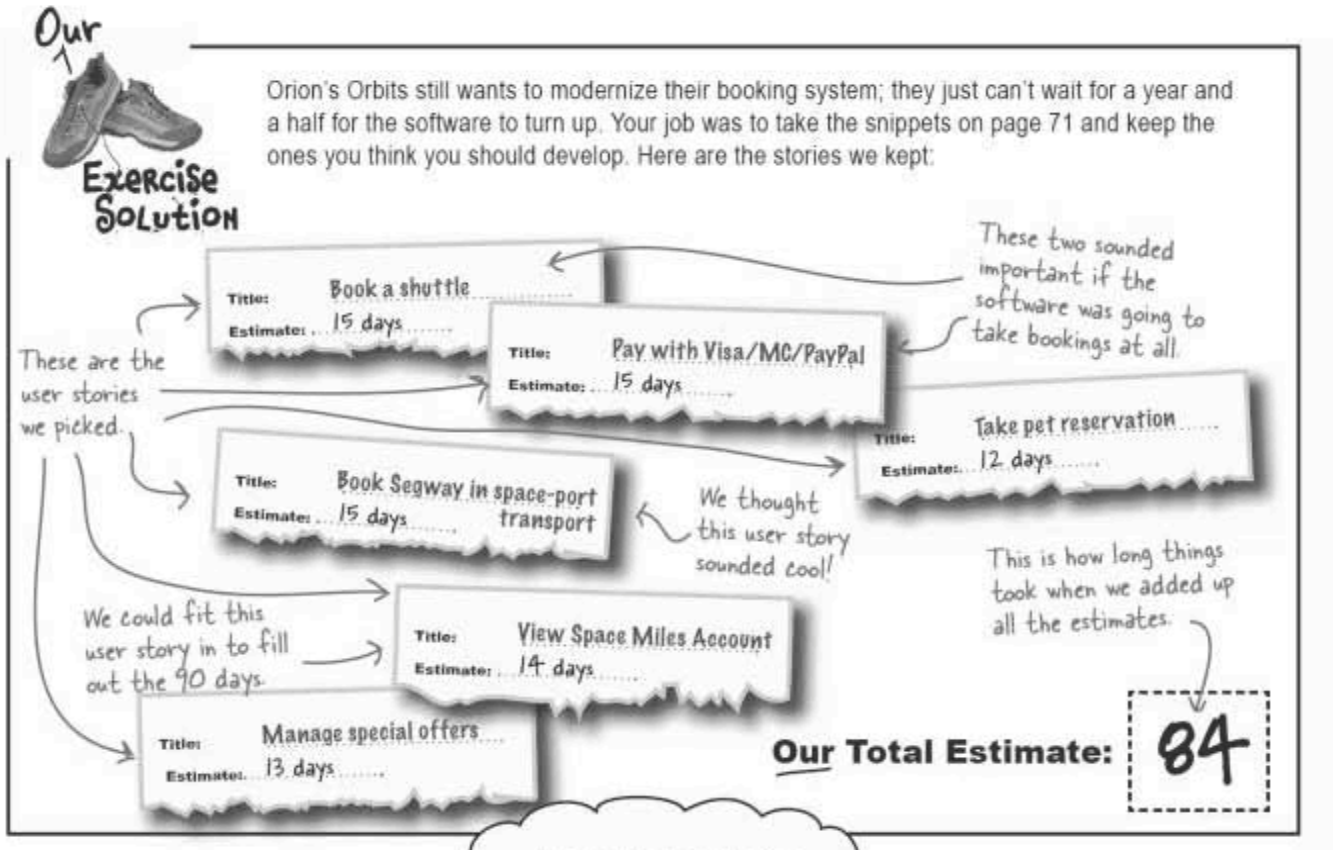
Title: Take pet reservation
Estimate: 12 days

Title: Manage special offers
Estimate: 13 days

Total Estimate: ☐ ← Total estimate for all of the user stories you've circled

Problems? See any problems with this approach? Write them down here.

Assumptions? Note down any assumptions you think you are making here.



The customer sets the priorities

Seems like the CEO of Orion's Orbits is not happy, and can you blame him? After all that hard work to figure out what he needs, we've ignored him completely when deciding which user stories take priority for the project.

When user stories are being prioritized, you need to **stay customer-focused**. Only the customer knows what is really needed. So when it comes to **deciding what's in and what's out,** you might be able to provide some expert help, but **ultimately it's a choice that the customer has to make**.

Prioritize with the customer

It's your **customer's call** as to what user stories take priority. To help the customer make that decision, shuffle and lay out all your user story cards on the table. Ask your customer to order the user stories by priority (the story most important to them first) and then to select the set of features that need to be delivered in Milestone 1.0 of their software.

What is "Milestone 1.0"?

Milestone 1.0 is your **first major release** of the software to the customer. Unlike smaller iterations where you'll show the customer your software for feedback, this will be the first time you actually **deliver your software** (and expect to get paid for the delivery). Some Do's and Don'ts when planning Milestone 1.0:

Do... balance functionality with customer impatience

Help the customer to understand **what can be done in the time available**. Any user stories that don't make it into Milestone 1.0 are not ignored, just postponed until Milestone 2, or 3...

Don't... get caught planning nice-to-haves

Milestone 1.0 is about **delivering what's needed**, and that means a set of functionality that meets the most important needs of the customer.

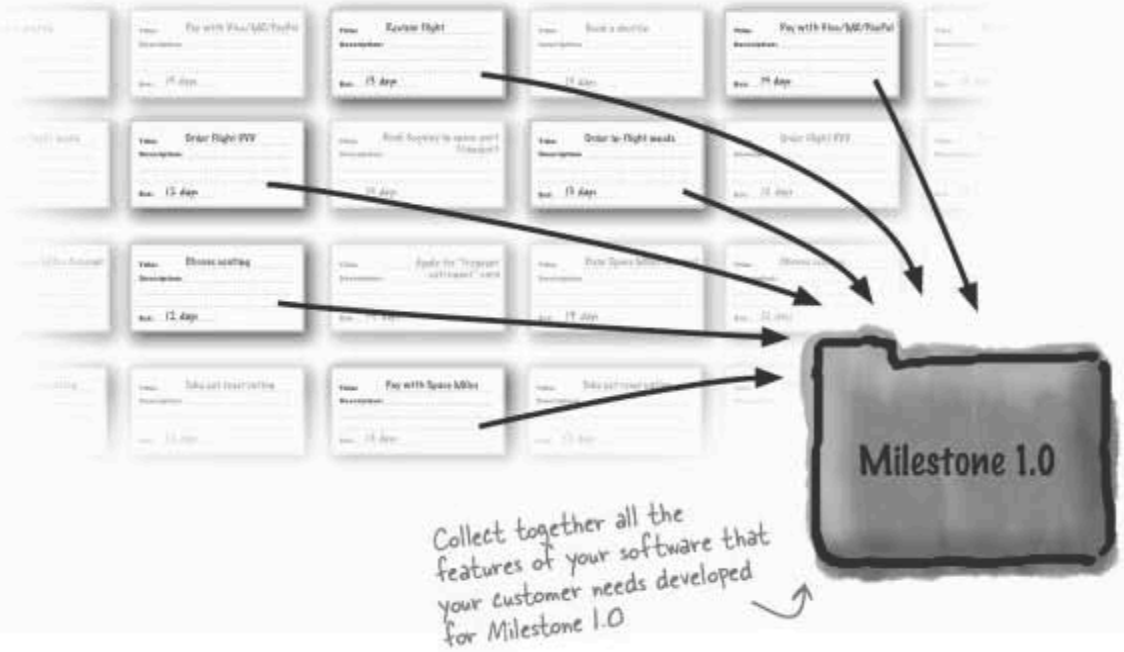
Don't... worry about length (yet)

At this point you're just asking your customer which are the most important user stories. **Don't get caught up on how long** those user stories will take to develop. You're just trying to understand the customer's priorities.

Don't worry, we're not ignoring estimation. We'll come right back to this.

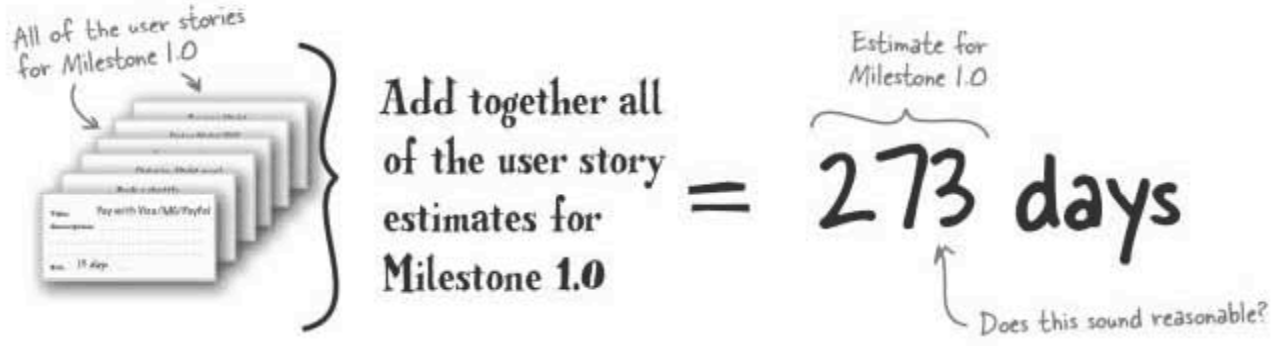
We know what's in Milestone 1.0 (well, maybe)

From all of the user stories developed from the customer's ideas, organized into prority order, the customer then selects the user stories that they would like to be a part of Milestone 1.0 of the software...



Sanity-check your Milestone 1.0 estimate

Now that you know what features the customer wants in Milestone 1.0, it's time to find out if you now have a reasonable length of project if you develop and deliver all of those most important features...



If the features don't fit, reprioritize

You've got 273 days of work for Milestone 1.0, and Orion's Orbits want delivery in **90 days**. Don't worry, this is pretty common. Customers usually want more than you can deliver, and it's your job to go back to them and reprioritize until you come up with a workable feature set.

To reprioritize your user stories for Milestone 1.0 with the customer...

1. **Cut out more FUNCTIONALITY**

   The very first thing you can look at doing to shorten the time to delivering Milestone 1.0 is to cut out some functionality by removing user stories that are not ***absolutely crucial*** to the software working.

   Once you explain the schedule, most customers will admit they don't really need everything they originally said they did.

2. **Ship a milestone build as early as possible**

   Aim to deliver a significant milestone build of your software as early as possible. This keeps your development momentum up by allowing you and your team to focus on a deadline that's not too far off.

   Don't let customers talk you into longer development cycles than you're comfortable with. The sooner your deadline, the more focused you and your team can be on it.

3. **Focus on the BASELINE functionality**

   Milestone 1.0 is all about delivering ***just*** the functionality that is needed for a working version of the software. Any features beyond that can be scheduled for later milestones.

there are no Dumb Questions

Q: What's the difference between a milestone and a version?

A: Not much. In fact you could call your first milestone "Version 1" if you like. The big difference between a milestone and a version is that a milestone marks a point at which you deliver signficant software and get paid by your customer, whereas a version is more of a simple descriptive term that is used to identify a particular release of your software.

The difference is really quite subtle, but the simple way to understand it is that "Version" is a label and doesn't mean anything more, whereas "Milestone" means you deliver signficant functionality and you get paid. It could be that Version 1.0 coincides with Milestone 1.0, but equally Milestone 1.0 could be Version 0.1, 0.2 or any other label you pick.

Q: So what exactly is my software's baseline functionality?

A: The **baseline functionality** of your software is the smallest set of features that it needs to have in order for it to be at all useful to your customer and their users.

Think about a word processing application. Its core functionality is to let you load, edit, and save text to a file. Anything else is beyond core functionality, no matter how useful those features are. Without the ability to load, edit, and save a document with text in it, a word processor simply ***is not useful***.

That's the rule of thumb: If you can get by without a feature, then it isn't really baseline functionality, and it's probably a good candidate for pushing out to a later milestone than Milestone 1.0 if you don't have time to get everything done.

Q: I've done the math and no matter how I cut the user stories up, I just can't deliver what my customer wants when they want me to. What can I do?

A: It's time to confess, unfortunately. If you really can't build the software that is required in the time that it's needed by, and your customer simply won't budge when it comes to removing some user stories from the mix, then you might need to walk away from the project and know that at least you were honest with the customer.

Another option is to try to beef up your team with new people to try and get more work done quicker. However, adding new people to the team will up the costs considerably, and won't necessarily get you all the advantages that you'd think it might.

If it takes you 273 days, with 2 more people like you, that would reduce the overall development time by a factor of 3, right?

It's about more than just development time

While adding more people can look really attractive at first, it's really not as simple as "double the people, halve the estimate."

Every new team member needs to **get up to speed on the project**; they need to **understand the software**, the **technical decisions**, and **how everything fits together**, and while they're doing that ***they can't be 100% productive***.

Then you need to get that new person set up with the right tools and equipment to work with the team. This could mean buying new licenses and purchasing new equipment, but even if it just means downloading some free or open source software, ***it all takes time*** and that time needs to be factored in as you reassess your estimates.

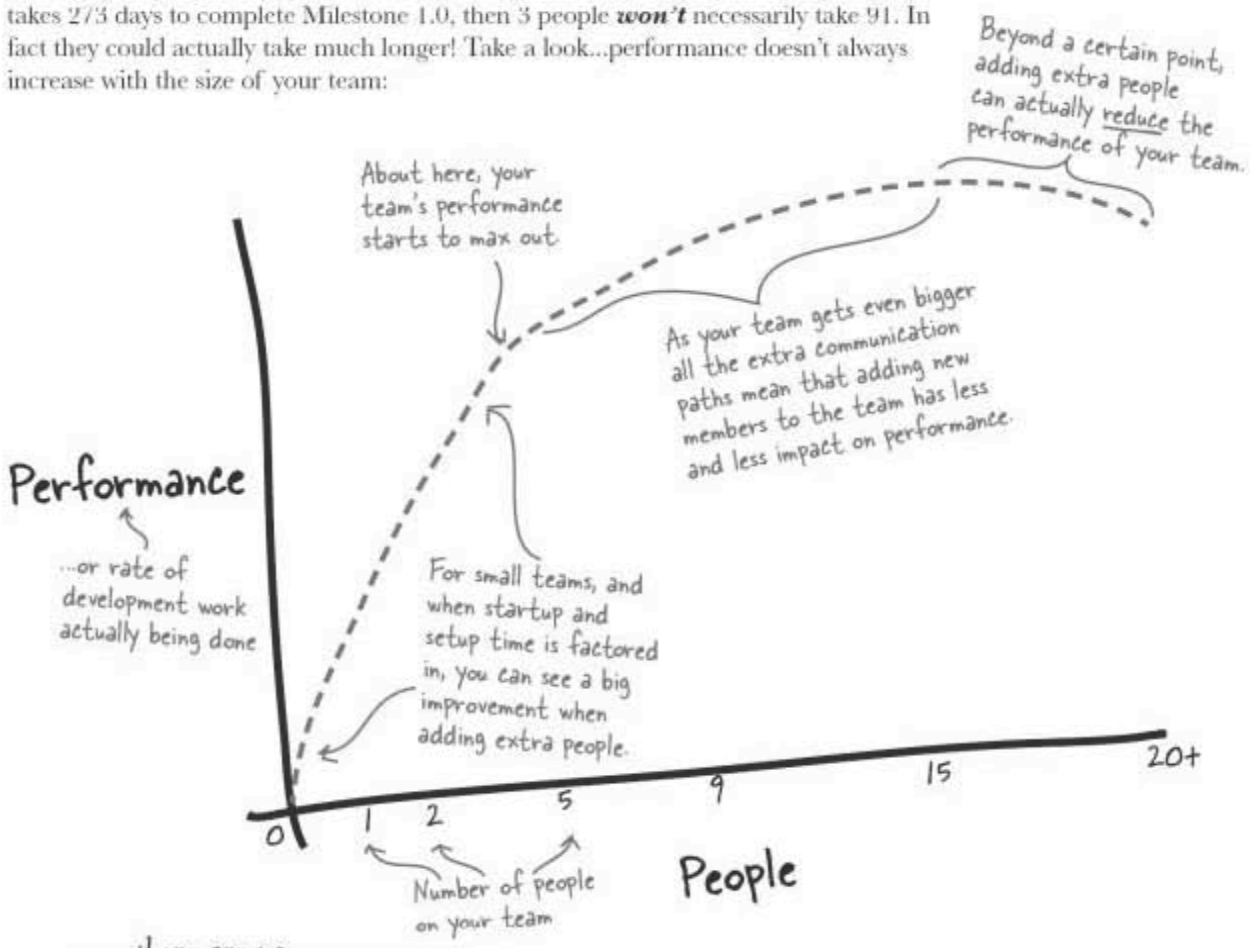
Finally, every person you add to your team makes the job of keeping everyone focused and knowing what they are doing harder. Keeping everyone moving in the same direction and on the same page can become a full-time job, and as your team gets larger you will find that this complex communication can start to hit your team's overall ability to be productive and develop great software.

In fact, there is a maximum number of people that your team can contain and still be productive, but it will depend very much on your project, your team, and who you're adding. The best approach is to monitor your team, and if you start to see your team actually get ***less productive***, even though you have ***more people***, then it's time to re-evaluate the amount of work you have to do or the amount of time in which you have to do it.

Later on in this chapter you'll be introduced to the burn-down rate graph. This is a great tool for monitoring the performance of your team.

More people sometimes means diminishing returns

Adding more people to your team doesn't always work as you'd expect. If 1 person takes 273 days to complete Milestone 1.0, then 3 people ***won't*** necessarily take 91. In fact they could actually take much longer! Take a look...performance doesn't always increase with the size of your team:



there are no Dumb Questions

Q: Is there a maximum team size that I should never go over?

A: Not really. Depending on your experience you may find that you can happily handle a 20-person team, but that things become impossible when you hit 21. Alternatively you might find that any more than three developers, and you start to see a dip in productivity. The best approach is to monitor performance closely and make amendments based on your observations.

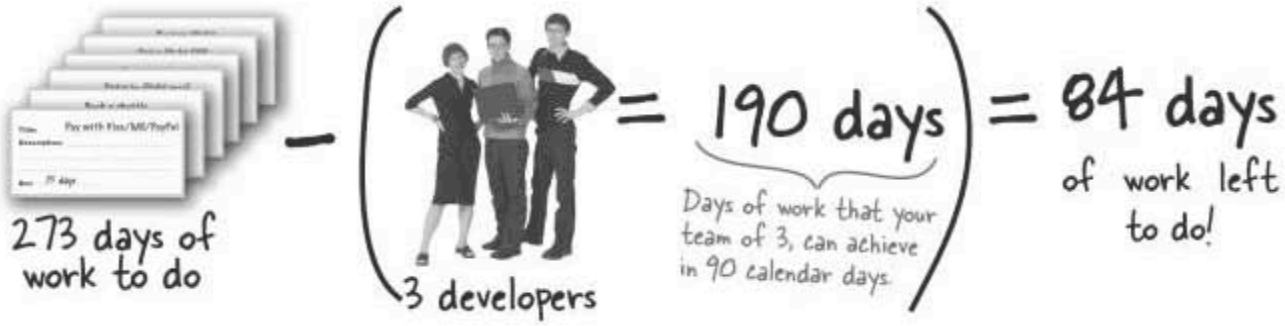
Do you think the size of your project affects this graph? What about if you broke your project up into smaller sub-projects?

Work your way to a reasonable Milestone 1.0

With Orion's Orbits, going from one person to three—by adding two more developers—can have a positive impact. So let's see how that works out:

First you add two new people to your team...

Adding two developers to your team (that's three including you) helps, but it's not a magical solution. Two developers can add a lot of work time to your project, but there's still work left:



...then you reprioritize with the customer

Now you've got a nice way to figure out what has to be removed. We've got 189 days of work time, and 273 days of work. So we need to talk to the customer and remove around 84 days of work by shifting out some user stories from Milestone 1.0.

there are no Dumb Questions

Q: But 190 days of work is less than the 190 days that our three-developer team can produce, shouldn't we add some more features with the customer?

A: The overall estimate doesn't actually have to be exactly 189 days. Given that we're dealing with estimates anyway, which are rarely 100% accurate, and that we tend to be slightly optimistic in our estimates then 165 days is close enough to the 189-day mark to be reasonably confident of delivering in that time.

Q: How did you come up with 190 days when you added two new developers?

A: At this point this number is a guesstimate. We've guessed that adding two people to build a team of three will mean we can do around 190 days of work in 90 calendar days. There are ways to back up this guess with some evidence using something called "team velocity," but we'll get back to that later on in this chapter.

BE the Customer

Think about baseline functionality. If a feature isn't essential, it's probably not a 10.

Now it's your chance to be the customer. You need to build a plan for when you are going to develop each of the user stories for Milestone 1.0, and to do that you need to ask the customer what features are most important so that you can develop those first. Your job is to play the customer by assigning a priority to the Milestone 1.0 user stories. For each user story, assign it a ranking in the square provided, depending on how important you think that feature is using the key at the bottom of the page.

Title: Pay using "Space Miles"
Est: 15 days
Priority:

Title: Order In-flight meals
Est: 13 days
Priority:

Title: Login to "Frequent Astronaut" account
Est: 15 days
Priority:

Title: Review flight
Est: 13 days
Priority:

Title: Manage special offers
Est: 13 days
Priority:

Title: Book a shuttle
Est: 15 days
Priority:

Title: Pay with Visa/ MC/PayPal
Est: 15 days
Priority:

Title: View "Space Miles" account
Est: 14 days
Priority:

Title: View flight reviews
Est: 12 days
Priority:

Title: Choose seating
Est: 12 days
Priority:

Title: Apply for "frequent astronaut" card
Est: 14 days
Priority:

Title: View shuttle deals
Est: 12 days
Priority:

For each user story, specify what priority it is in the box provided.

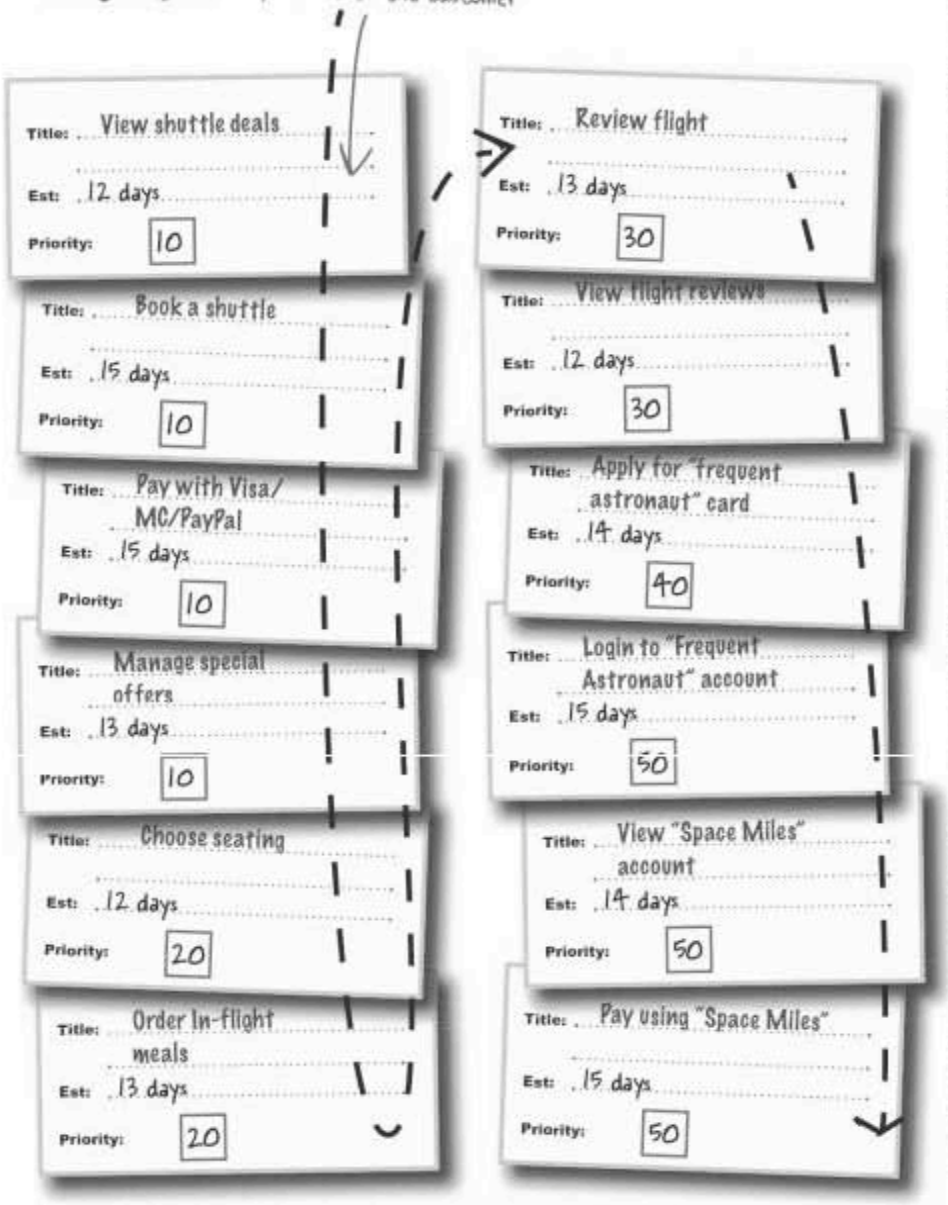
Priorities Key

10 - **Most Important**
20
30
40
50 - **Least Important**

Our BE the Customer Solution

Your job was to play the customer and prioritize the Milestone 1.0 user stories. Here are the priorities that we assigned to each of the user stories. We also laid out the user stories in order of

Order of priority, most to least important to the customer



there are no Dumb Questions

Q: Why are the priorities 10, 20, 30, 40, and 50?

A: Powers of ten get the brain thinking about groupings of features, instead of ordering each and every feature separately with numbers like 8 or 26 or 42. You're trying to get the customer to decide what is most important, but not get too hung up on the exact numbers themselves. Also, powers of ten allow you to occasionally specify, say, a 25 for a particular feature when you add something in later, and need to squeeze it between existing features.

Q: If it's a 50, then maybe we can leave it out, right?

A: No, 50 doesn't mean that a user story is a candidate for leaving out. At this point, we're working on the user stories for Milestone 1.0, and so these user stories have ***already*** been filtered down to the customer's most important features. The goal here is to prioritize, not figure out if any of these features aren't important. So a 50 just says it can come later, not that it's not important to the customer

Q: What if I have some non–Milestone 1.0 user story cards?

A: Assign a priority of 60 to those cards for now, so they don't get mixed in with your Milestone 1.0 features.

Q: And the customer does all this work?

A: You can help out and advise, maybe mentioning dependencies between some of the user stories. But the final decision on priorities is ***always*** the customer's to make.

Exercise

Now that you have your user stories for Milestone 1.0 in priority order, it's time to build some iterations. Lay out the user stories so they make iterations that make sense to you. Be sure and write down the total days of work, and how long that will take for your team of three developers.

First one added for you

Iteration 1

Title: View shuttle deals

Est: 12 days

Priority: 10

Total Days: ☐ Divide by 3 developers: ☐

Iteration 2

Total Days: ☐ Divide by 3 developers: ☐

Iteration 3

Total Days: ☐ Divide by 3 developers: ☐

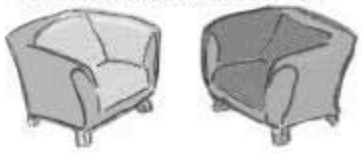
Bonus question

What do you think you should do at the end of an iteration?

Answers on page 84.

milestone = paid, iteration = on track

Fireside Chats



Tonight's talk: **A sit-down discussion between an iteration and a milestone.**

Milestone:

Hello there, iteration, seems like it's only been a month since I saw you last.

So how are things going on the project? It seems like you're always showing up, and I just arrive for the big finish. Actually, what's your purpose?

Naive? Look, just because I've had a few customer run-ins before doesn't mean I'm not important. I mean, without me, you wouldn't have software at all ,let alone get paid! Besides, just because I've shown up and surprised the occasional customer from time to time...

I used to try that, too. I'd try and soften the blow by explaining to the customer that all of their problems would be fixed in the next version of the software, but that wasn't what they wanted to hear. Lots of yelling, and I'd slink off, ready to go back to work for a year or so, and see if the customer liked me better next time.

Iteration:

Almost exactly a month. And you'll see me again next month, I can guarantee it. About three times, and we're ready for you, Milestone 1.0.

To make sure things go great, of course. That's my job really, to make sure that every step of the way from day 1 to day 90, the project stays on track. What, you thought you could just show up three months into the project and everything would be just like the customer wants it? A bit naive, aren't you?

Oh, I really sympathize with you there. I hate it when the customer isn't happy with me. But then again, there's a lot more time to fix things. I mean, we get together, you know, me and the customer, at least once a month. And, if things are bad, I just let the customer know it'll be better next time.

But you're shorter than a year now, right?

Milestone:

Well, I try to be, but sometimes that's just how long it takes, although I just love seeing the customer more often. At least once a quarter seems to line up with their billing cycles. And not so long that I get forgotten about; there's nothing worse than that.

Are you kidding? You're not even an alpha or a beta....just some code glued together, probably an excuse for everyone to wear jeans to work and drink beer on Friday afternoon.

Ha! Where would I be? Same place I am right now, getting ready to show the customer some real...

...software. Hey, wait. Hopefully? I've got a few hopes for you, you little...

Ungrateful little punk...release this!

Iteration:

Yeah, nobody forgets about me. Around every month, there I am, showing up, putting on a song and dance, pleasing the customer. Really, I can't imagine how you ever got by without me.

Oh, it's a little more than that, don't you think? Where would you be without me paving the way, making sure we're on track, handling changes and new features, and even removing existing features that aren't needed any more.

...***hopefully*** working?

Well, you got the little part right. Why don't you just shuffle off for another 30 days or so, we'll call you when all the work's done. Then we'll see who Friday beers are on, OK?

Sure thing, and since I do my job, I'm sure you'll work just fine. I'm outta here, plenty of work left to be done...

Exercise Solution

Your job was to lay out the user stories so they make iterations that make sense. Here's what we came up with... note that all our iterations are within one calendar month, about 20 working days (or less).

Your answers could be different, but make sure you went in order of priority...

Iteration 1

Title	Est	Priority
Manage special offers	13 days	10
Book a shuttle	15 days	10
Pay with Visa/ MC/PayPal	15 days	10
View Shuttle deals	12 days	10

...and make sure you kept your iterations short

Total Days: 57 Divide by 3 developers: 19

Iteration 2

Title	Est	Priority
Choose seating	12 days	20
Order In-flight meals	13 days	20
Review flight	13 days	30
View flight reviews	12 days	30

Total Days: 50 Divide by 3 developers: 17

Iteration 3

Title	Est	Priority
Apply for "frequent astronaut" card	14 days	40
Login to "Frequent Astronaut" account	15 days	50
View "Space Miles" account	14 days	50
Pay using "Space Miles"	15 days	50

Total Days: 58 Divide by 3 developers: 20

What do you think you should do at the end of an iteration? Show the customer and get their feedback

there are no Dumb Questions

Q: What if I get to the end of an iteration, and I don't have anything to show my customer?

A: The only way you should end up at the end of an iteration and not have something to show the customer is if no user stories were completed during the iteration. If you've managed to do this, then your project is out of control and you need to get things back on track as quickly as possible.

Keep your software continuously building and your software always runnable so you can always get feedback from the customer at the end of an iteration.

Iterations should be short and sweet

So far Orion's Orbits has focussed on ***30-day iterations***, with 3 iterations in a 90-day project. You can use different size iterations, but make sure you keep these basic principles in mind:

Keep iterations short

The shorter your iterations are, the more chances you get to find and deal with change and unexpected details ***as they arise***. A short iteration will get you feedback earlier and bring changes and extra details to the surface sooner, so you can adjust your plans, and even change what you're doing in the next iteration, before you release a faulty Milestone 1.0.

Keep iterations balanced

Each iteration should be a balance between dealing with change, adding new features, beating out bugs, and accounting for real people working. If you have iterations every month, that's not really 30 days of work time. People take weekends off (at least once in a while), and you have to account for vacation, bugs, and things that come up along the way. A 20-work-day iteration is a safe bet of work time you can handle in an actual 30-day, calendar-month iteration.

30-day iterations are basically 30 CALENDAR days...

...which you can assume turn into about 20 WORKING days of productive development

SHORT iterations help you deal with change and keep you and your team motivated and focused.

WHO DOES WHAT?

Below is a particular aspect of a user story, iteration, milestone...or perhaps two, or even all three! Your job is to check off the boxes for the different things that each aspect applies to.

	User story	Iteration	Milestone
I result in a buildable and runnable bit of software.	☐	☐	☐
I'm the smallest buildable piece of software.	☐	☐	☐
In a full year, you should deliver me a maximum of four times.	☐	☐	☐
I contain an estimate set by your team.	☐	☐	☐
I contain a priority set by the customer.	☐	☐	☐
When I'm done, you deliver software to the customer and get paid.	☐	☐	☐
I should be done and dusted in 30 days.	☐	☐	☐

→ Answers on page 88.

Comparing your plan to reality

Bob: Oh, just so you know, Nick is coming in at 11 today, he's got a doctor's appointment...

Laura: What?

Bob: And while we're talking, the IT guys are installing Oracle 9 on my machine this afternoon, so you might want to keep that in mind, too.

Laura: Oh great, any other nasty surprises in there that I should be aware of?

Bob: Well, I have got a week of vacation this month, and then there's Labor Day to take into account...

Laura: Perfect, how can we come up with a plan that factors all these overheads in so that when we go get signoff from the CEO of Orion's Orbits we know we have a plan we can deliver?

Do you think our current 20-work-day iterations take these sorts of issues into account?

Sharpen your pencil

See if you can help Bob out. Check all the things that you need to account for when planning your iterations.

- ☐ Paperwork
- ☐ Equipment failure
- ☐ Holidays
- ☐ Sickness
- ☐ Software upgrades
- ☐ Frank winning the lottery

WHO DOES WHAT?

Below is a particular aspect of a user story, iteration, milestone...or perhaps two, or even all three! Your job is to check off the boxes for the different things that each aspect applies to.

	User Story	Iteration	Milestone
I result in a buildable and runnable bit of software.	☐	☒	☒
I'm the smallest buildable piece of software.	☐	☒	☐
In a full year, you should deliver me a maximum of four times.	☐	☐	☒
I contain an estimate set by your team.	☒	☒	☒
I contain a priority set by the customer.	☒	☐	☐
When I'm done, you deliver software to the customer and get paid.	☐	☐	☒
I should be done and dusted in 30 days.	☐	☒	☐

Sharpen your pencil

See if you can help Bob out. Check all the things that you need to account for when planning your iterations.

Things like this always occur... so we have to plan for them.

- ☒ Paperwork
- ☐ Equipment failure
- ☒ Holidays
- ☒ Sickness
- ☒ Software upgrades
- ☐ Frank winning the lottery

You really can't factor in complete surprises

Velocity accounts for overhead in your estimates

It's time to add a little reality to your plan. You need to factor in all those annoying bits of overhead by looking at how fast you and your team actually develop software. And that's where **velocity** comes in. Velocity is a percentage: given X number of days, how much of that time is productive work?

Start with a velocity of 0.7.

On the first iteration with a new team it's fair to assume that your team's working time will be about 70% of their available time. This means your team has a velocity value of 0.7. In other words, for every 10 days of work time, about 3 of those days will be taken up by holidays, software installation, paperwork, phone calls, and other nondevelopment tasks.

That's a conservative estimate, and you may find that over time, your team's actual velocity is higher. If that's the case, then, at the end of your current iteration, you'll adjust your velocity and use that new figure to determine how many days of work can go into the next iteration.

Yet another reason to have short iterations: you can adjust velocity frequently.

Best of all, though, you can apply velocity to your amount of work, and get a **realistic estimate** of how long that work will actually take.

Take the days of work it will take you to develop a user story, or an iteration, or even an entire milestone...

...and DIVIDE that number by your velocity, which should be between 0 and 1.0. Start with 0.7 on a new project as a good conservative estimate.

$$\frac{\text{days of work}}{\text{velocity}} = \text{days required to get work done}$$

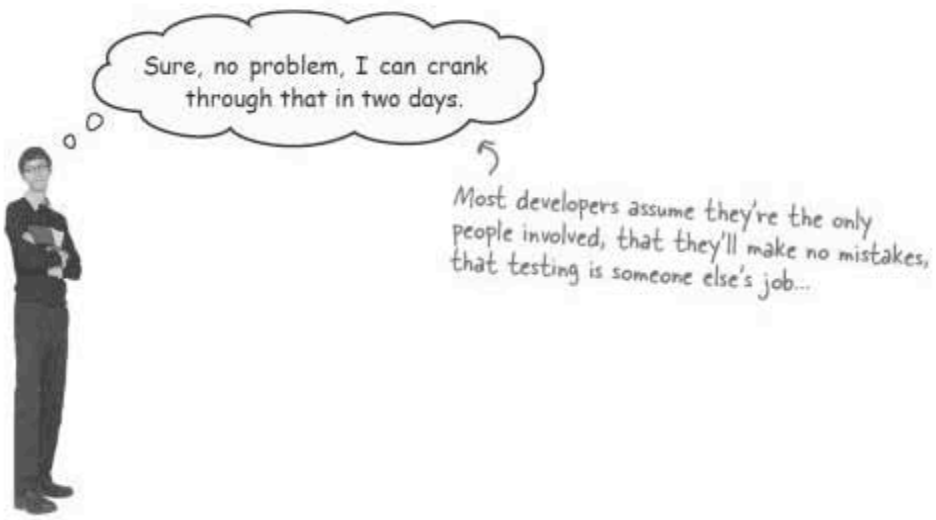
The result should always be BIGGER than the original days of work, to account for days of administration, holidays, etc.

Seeing a trend? 30 days of a calendar month was really 20 days of work, and 20 days of work is really only about 15 days of productive time.

Programmers think in UTOPIAN days...

Ask a programmer how long it takes to get something done, like writing a PHP interface to a MySQL database, or maybe screen-scraping World Series scores from *espn.com*. They're going to give you a ***better-than-best-case estimate***.

Here's what a programmer SAYS...



...but here's what he's really THINKING

Developers think in REAL-WORLD days...

To be a software developer, though, you have to deal with reality. You've probably got a team of programmers, and you've got a customer who won't pay you if you're late. On top of that, you may even have other people depending on you—so your estimates are more conservative, and take into account real life:

Sharpen your pencil

Take your original estimates for each iteration from the solution on page 84 and apply a 70% velocity so that you can come up with a more confident estimate for all the work in Milestone 1.0.

Iteration 1
57 days of work / 0.7 = ...

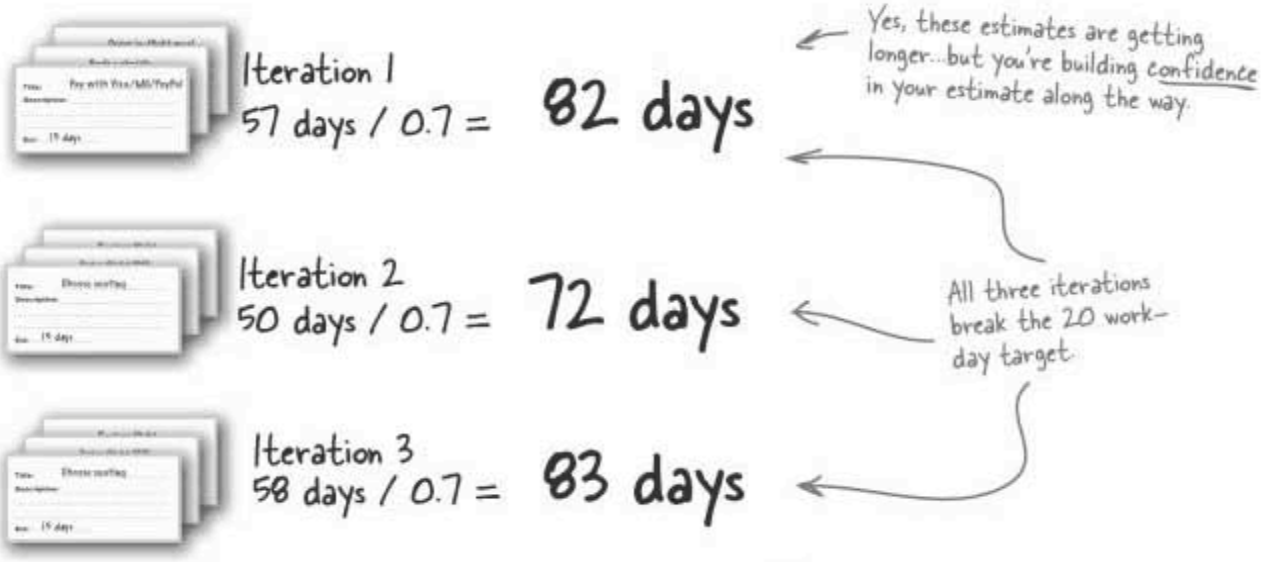
Iteration 2
50 days of work / 0.7 = ...

Iteration 3
58 days of work / 0.7 = ...

Milestone 1.0 = ...

When is your iteration too long?

Suppose you have three developers on your team who are working at a velocity of 0.7. This means that to calculate **how long an iteration will really take *your team***, you need to apply your velocity to the iteration's estimate:



So if you have 3 developers, each of them has to work 79 days in 3 months... but there are only 60 working days.

BRAIN POWER

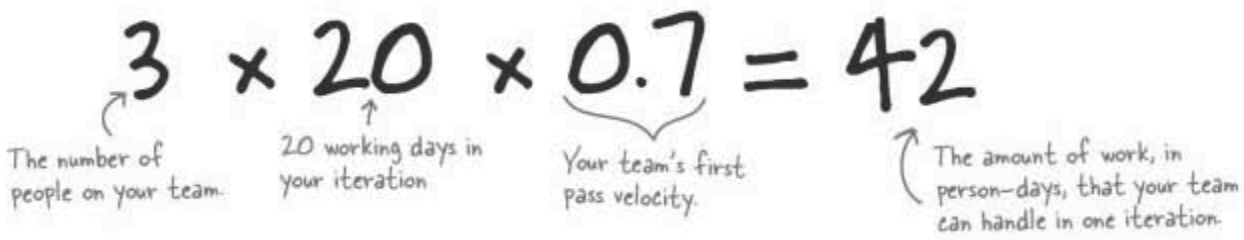
How would you bring your estimates back to 20 work-day cycles so you can deliver Milestone 1.0 on time, without working weekends?

Deal with velocity BEFORE you break into iterations

A lot of this pain could actually have been avoided if you'd applied velocity at the ***beginning*** of your project. By applying velocity up front, you can calculate how many days of work you and your team can produce in each iteration. Then you'll know exactly what you can ***really*** deliver in Milestone 1.0.

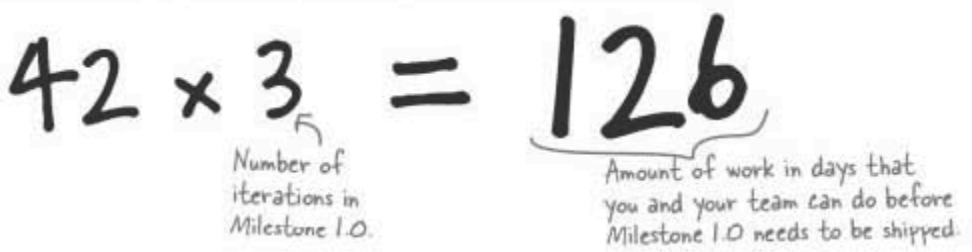
First, apply your team velocity to each iteration

By taking the number of people in your team, multiplied by the number of actual working days in your iteration, multiplied finally by your team's velocity, you can calculate how many **days of actual work** your team can produce in one iteration:



Add your iterations up to get a total milestone estimate

Now you should estimate the number of iterations you need for your milestone. Just multiply your days of work per iteration by the number of iterations, and you've got the number of working days you can devote to user stories for your milestone:



there are no Dumb Questions

Q: That sucks! So I only have 14 days of actual productive work per iteration if my velocity is 0.7?

A: 0.7 is a conservative estimate for when you have new members of your team coming up to speed and other overheads. As you and your team complete your iterations, you'll keep coming back to that velocity value and updating it to reflect how productive you really are.

Q: With velocity, my Milestone 1.0 is now going to take 79 working days, which means 114 calendar days. That's much more than the 90-day/3-month deadline that Orion's Orbits set, isn't that too long?

A: Yes! Orion's Orbits need Milestone 1.0 in 90 calendar days, so by applying velocity, you've now got too much work to do to meet that deadline. You need to reassess your plan to see what you really can do with the time and team that you have.

Long Exercise

When your iterations contain too much work for your team, there's nothing else to do but reshuffle work until your iterations are manageable. Take the Orion's Orbits Milestone 1.0 user stories and organize them into iterations that each contain no more than 42 days of work.

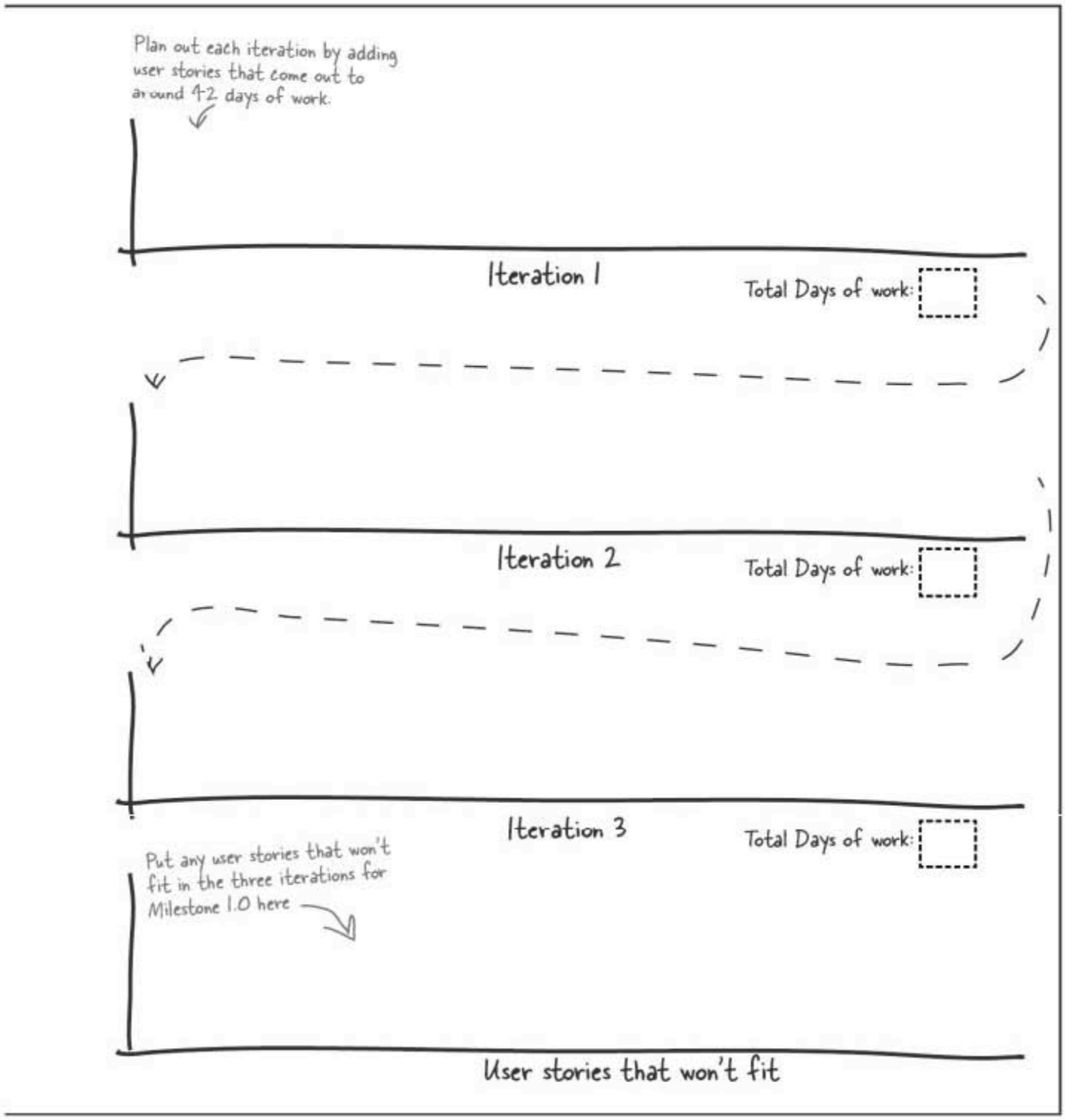
The maximum amount of work your team can do in a 20-day iteration, factoring in your velocity this time.

Title:	Est:	Priority:
View shuttle deals	12 days	10
Book a shuttle	15 days	10
Pay with Visa/ MC/PayPal	15 days	10
Manage special offers	13 days	10

Remember to respect the customer's original order of priority in your iterations.

Title:	Est:	Priority:
Choose seating	12 days	20
Order In-flight meals	13 days	20
Review flight	13 days	30
View flight reviews	12 days	30

Title:	Est:	Priority:
Apply for "frequent astronaut" card	14 days	40
Login to "Frequent Astronaut" accoun	15 days	50
View "Space Miles" account	14 days	50
Pay using "Space Miles"	15 days	50


Plan out each iteration by adding user stories that come out to around 1-2 days of work.
Iteration 1
Total Days of work:
Iteration 2
Total Days of work:
Iteration 3
Total Days of work:
Put any user stories that won't fit in the three iterations for Milestone 1.0 here
User stories that won't fit

Long Exercise Solution

Your job was to to take the Orion's Orbits user stories and aim for iterations that contain no more than 42 days of work each.

Iteration 1 — Total Days of work: 42

Title	Est	Priority
View shuttle deals	12 days	10
Book a shuttle	15 days	10
Pay with Visa/ MC/PayPal	15 days	10

Iteration 2 — Total Days of work: 38

Title	Est	Priority
Manage special offers	13 days	10
Choose seating	12 days	20
Order In-flight meals	13 days	20

Iteration 3 — Total Days of work: 39

Title	Est	Priority
Review flight	13 days	30
View flight reviews	12 days	30
Apply for Space Miles Loyalty Card	14 days	40

These user stories dropped off of the plan...

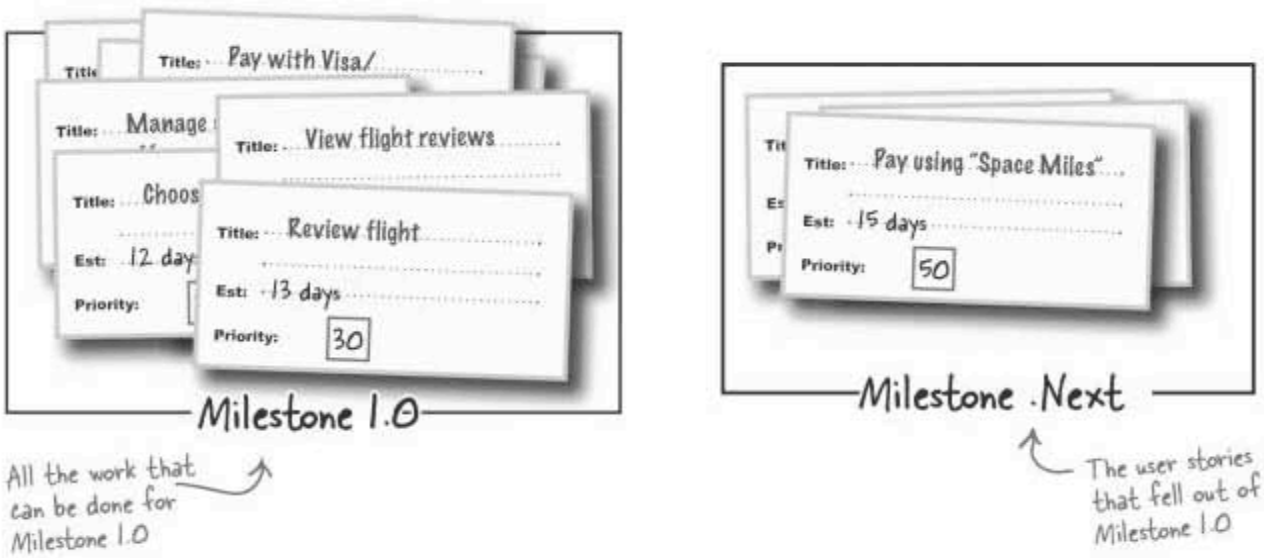
User stories that won't fit

Title	Est	Priority
Login to "Frequent Astronaut" account	15 days	50
View "Space Miles" account	14 days	50
Pay using "Space Miles"	15 days	50

Time to make an evaluation

So what's left? You've probably got a lot of user stories that still fit into Milestone 1.0...and maybe a few that don't. That's because we didn't figure out our velocity before our iteration planning.

Estimates without velocity can get you into real trouble with your customer.

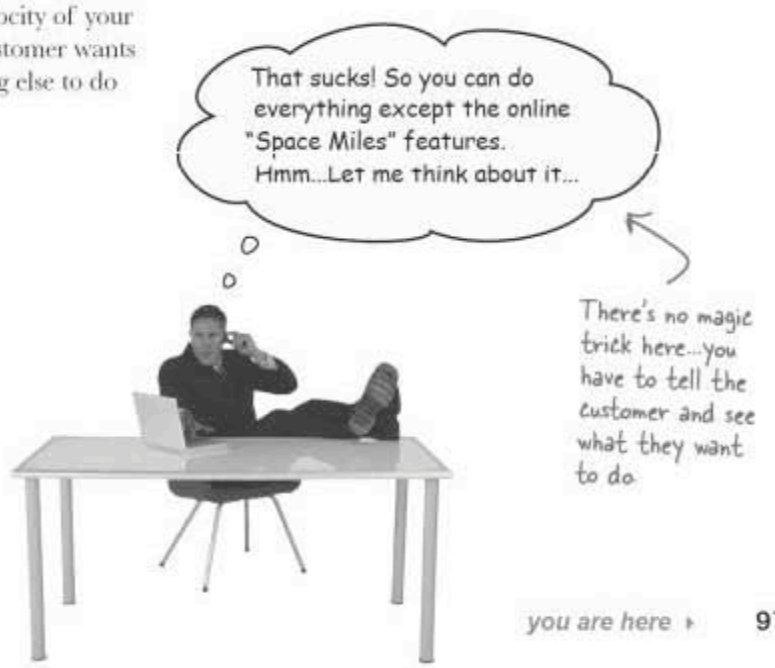


All the work that can be done for Milestone 1.0

The user stories that fell out of Milestone 1.0

Deliver the bad news to the customer

It's the time that every software developer dreads. You've planned out your iterations, factored in the velocity of your team, but you still can't get everything your customer wants done in time for their deadline. There's nothing else to do but come clean...



There's no magic trick here...you have to tell the customer and see what they want to do.

breaking** realistic **but** bad **news

[Note from human resources: we prefer the term unsympathetic customers.]

Managing ~~displeased~~ customers

Customers usually aren't happy when you tell them you can't get everything done in the time they want. Be honest, though; you want to come up with a plan for Milestone 1.0 that you can achieve, not a plan that just says what the customer wants it to say.

...and has you on a fast track to failure!

So what do you do when this happens?

It's almost inevitable that you're not going to be able to do everything, so it helps to be prepared with some options when you have to tell the customer the bad news...

1 **Add an iteration to Milestone 1.0**
Explain that the extra work can be done if an additional iteration is added to the plan. That means a longer development schedule, but the customer will get what they want in Milestone 1.0.

Another iteration gives your team plenty of time to develop all the customer's stories—but that pushes out the release date of Milestone 1.0, too.

2 **Explain that the overflow work is not lost, just postponed**
Sometimes it helps to point out that the user stories that can't make it into Milestone 1.0 are not lost; they are just put on the back burner until the next milestone.

These extra stories aren't trashed—they just fall into Milestone 2.0. Are space miles so important that they're worth starting over with a new development team?

3 **Be transparent about how you came up with your figures**
It sounds strange, but your customer only has your word that you can't deliver everything they want within the deadline they've given you, so it sometimes helps to explain where you're coming from. If you can, show them the calculations that back up your velocity and how this equates to their needs. And tell your customer you ***want*** to deliver them successful software, and that's why you've had to sacrifice some features to give yourself a plan that you are confident that you can deliver on.

there are no Dumb Questions

Q: If I'm close on my estimates, can I fudge a little and squeeze something in?

A: We REALLY wouldn't recommend this. Remember, your estimates are only educated guesses at this point, and they are actually more likely to take slightly longer than originally thought than shorter.

It's a much better idea to leave some breathing room around your estimates to really be confident that you've planned a successful set of iterations.

Q: I have a few days left over in my Milestone 1.0. Can't I add in a user story that breaks my day limit just a little bit?

A: Again, probably not a good idea. If your stories add up to leave you one or two days at the end of the iteration, that's OK. (In Chapter 9 we'll talk about what you can do to round those out.)

Q: OK, without squeezing my last user story in I end up coming under my work-day limit by a LONG way. I have 15 days free at the end of Milestone 1.0! Is there anything I can do about that?

A: To fit a story into that space, try and come up with two simpler stories and fit one of those into Milestone 1.0 instead.

Q: 0.7 seems to add up to a LOT of lost time. What sorts of activities could take up that sort of time?

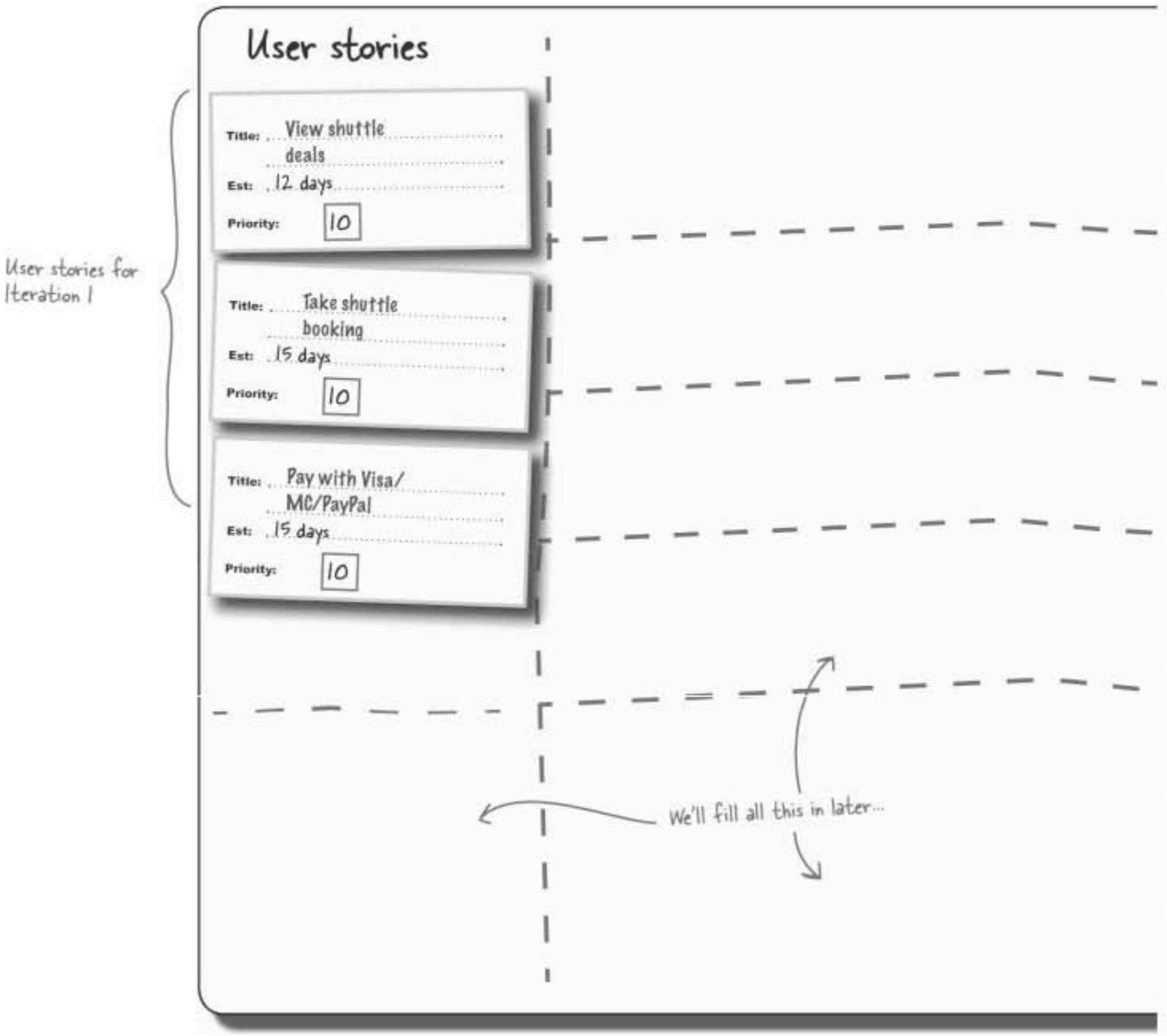
A: 0.7 is a safe first guess at a team's velocity. One example is where you are installing a new piece of software, like an IDE or a database (naming no specific manufacturers here, of course). In cases like these two hours of interrupted work can actually mean FOUR hours of lost time when you factor in how long it can take a developer to get back in "the zone" and developing productively.

It's also worth bearing in mind that velocity is recalculated at the end of every iteration. So even if 0.7 seems low for your team right now, you'll be able to correct as soon as you have some hard data. In Chapter 9 we'll be refining your velocity based on your team's performance during Iteration 1.

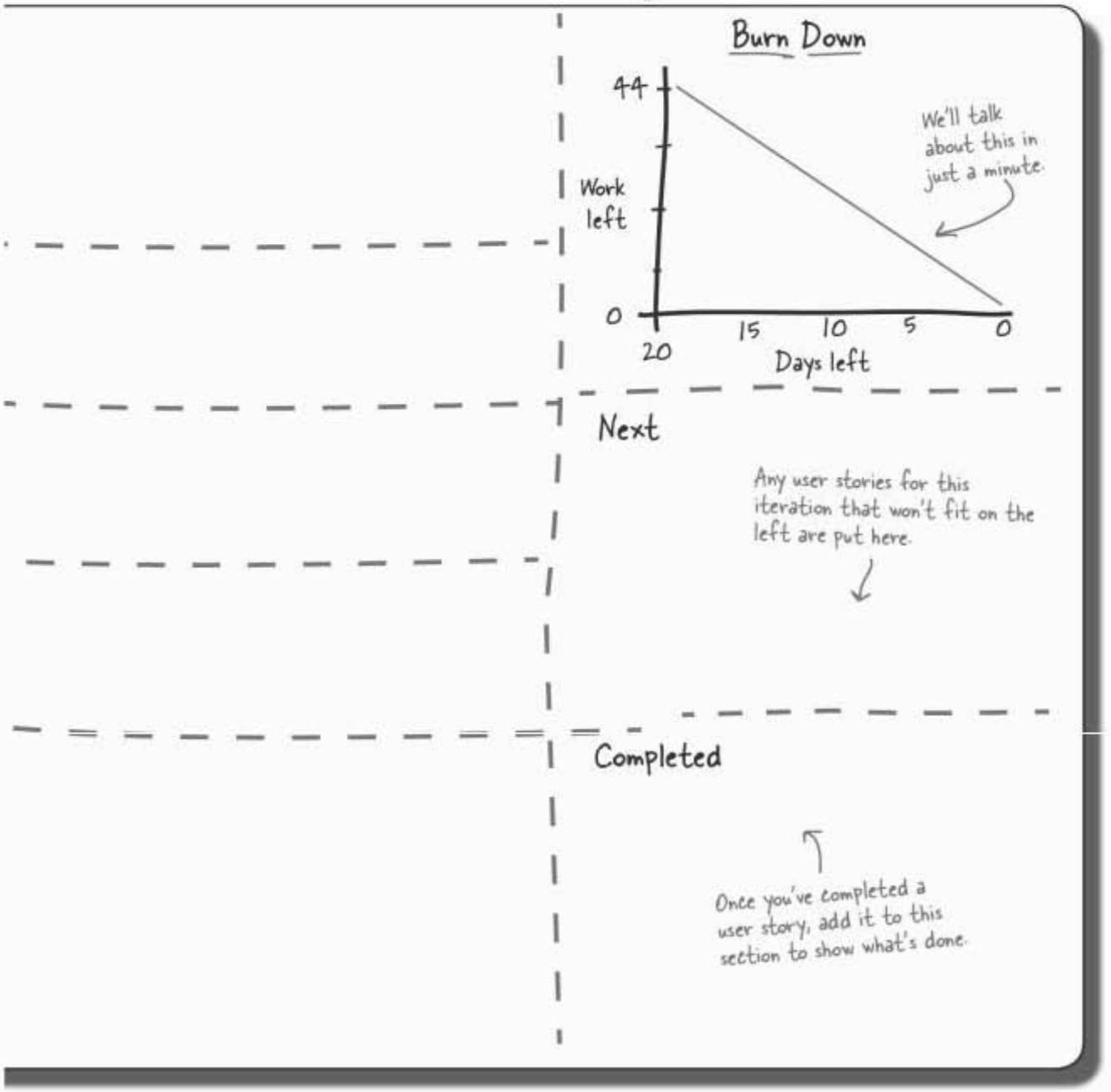
Stay confident that you can achieve the work you sign up for. You should promise and deliver rather than overpromise and fail.

The Big Board on your wall

Once you know exactly what you're building, it's time to set up your **software development dashboard** for Iteration 1 of development. Your dashboard is actually just a **big board** on the wall of your office that you can use to keep tabs on **what work is in the pipeline**, **what's in progress,** and **what's done**.

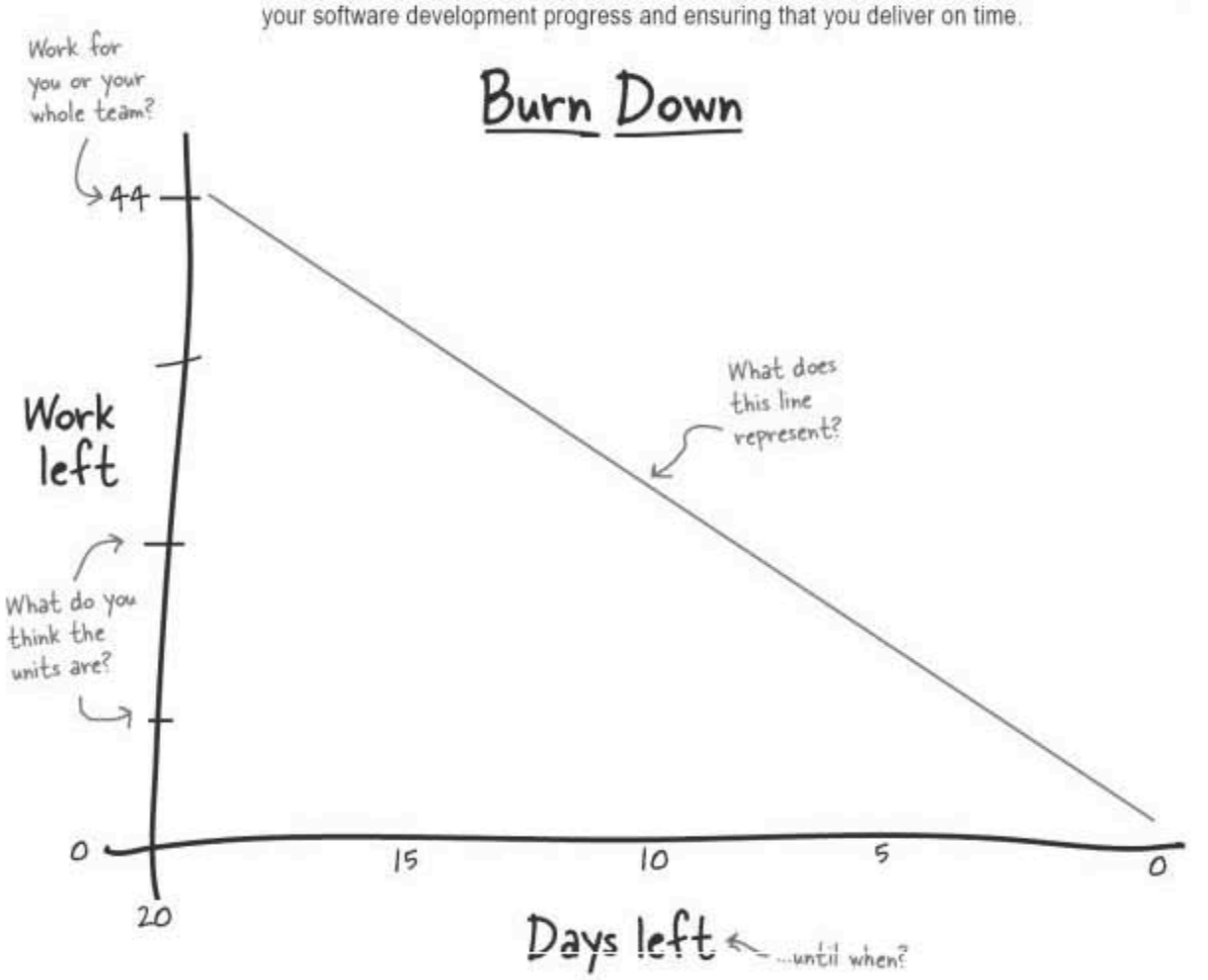


Usually your project board is a whiteboard, so you can use it again and again between iterations and projects.



Exercise

You may have noticed a graph at the top right of your development dashboard, but what is it for? Take a few minutes to glance over the burn-down graph below and write on it what you think the different parts of the graph are for and how it is one of the key tools for monitoring your software development progress and ensuring that you deliver on time.



→ Answers on page 104.

What do you think would be measured on this graph, and how?

.....

.....

.....

.....

How to ruin your team's lives

It's easy to look at those long schedules, growing estimates, and diminishing iteration cycles, and start to think, "**My team can work longer weeks!**" If you got your team to agree to that, then you're probably setting yourself up for some trouble down the line.

Personal lives matter

Long hours are eventually going to affect your personal life and the personal lives of the developers on your team. That might seem trite, but a happier team is a more productive team.

Fatigue affects productivity

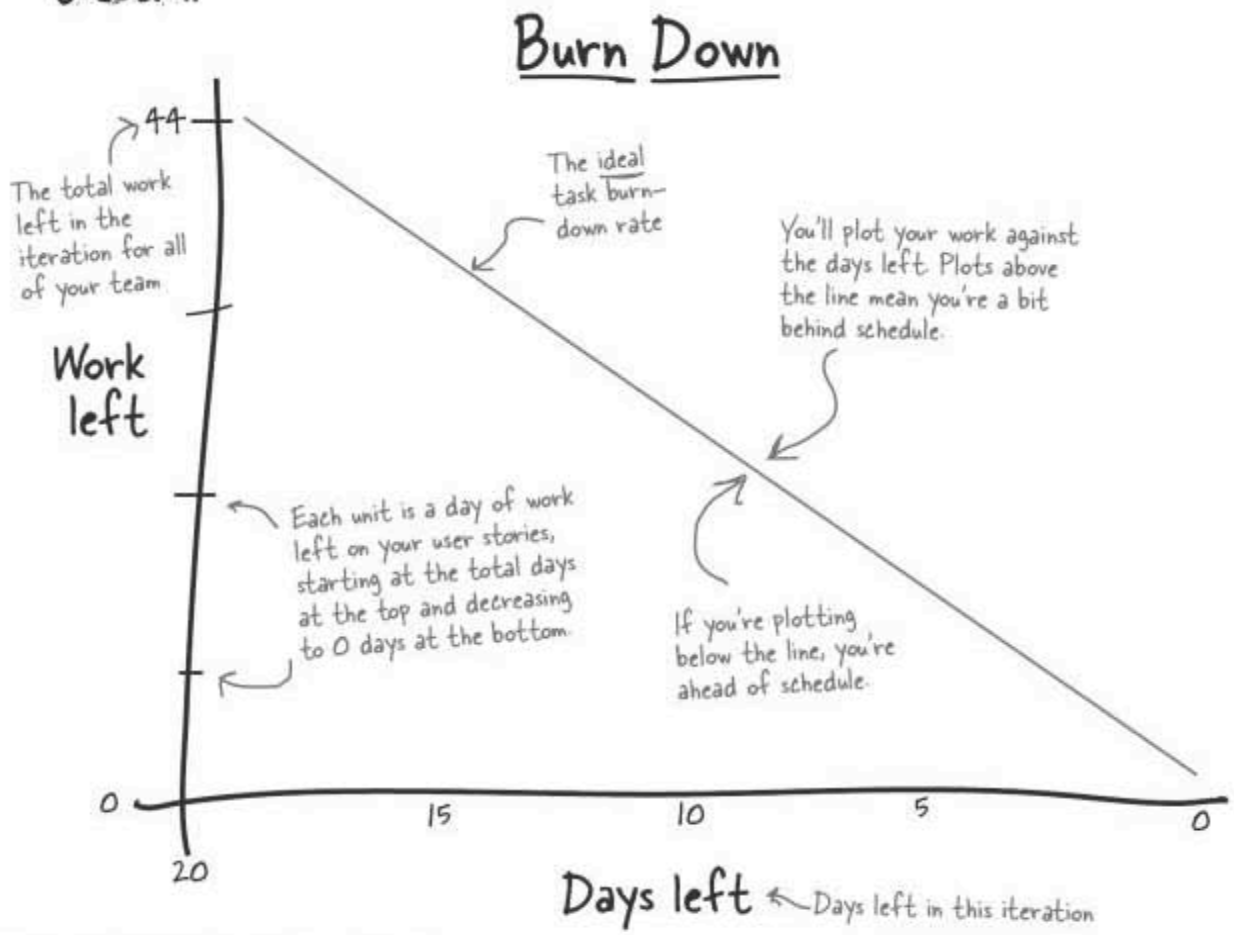
Tired developers aren't productive. Lots of studies suggest that developers are really only incredibly productive for about three hours a day. The rest of the day isn't a loss, but the more tired your developers are, the less likely they'll even get to that three hours of really productive time.

Be confident in your plans by applying velocity and not overworking yourself and your team.

BULLET POINTS

- The first step to planning what you are going to develop is to ask the customer to **prioritize their requirements**.
- **Milestone 1.0** should be delivered as **early** as you can.
- During Milestone 1.0 try to **iterate around once a month** to keep your development work on track.
- When you don't have enough time to build everything, ask the **customer** to **reprioritize**.
- Plan your iterations by factoring in your team's **velocity** from the **start**.
- If you really can't do what's needed in the time allowed, **be honest** and **explain why** to the customer.
- Once you have an agreed-upon and achievable set of user stories for Milestone 1.0, it's time to set up your **development dashboard** and get developing!

You were asked to take a few minutes to glance over the burn-down graph below and describe what you think the different parts of the graph are for and how it is one of the key tools for monitoring your software development progress and ensuring that you deliver on time.



What do you think would be measured on this graph, and how?

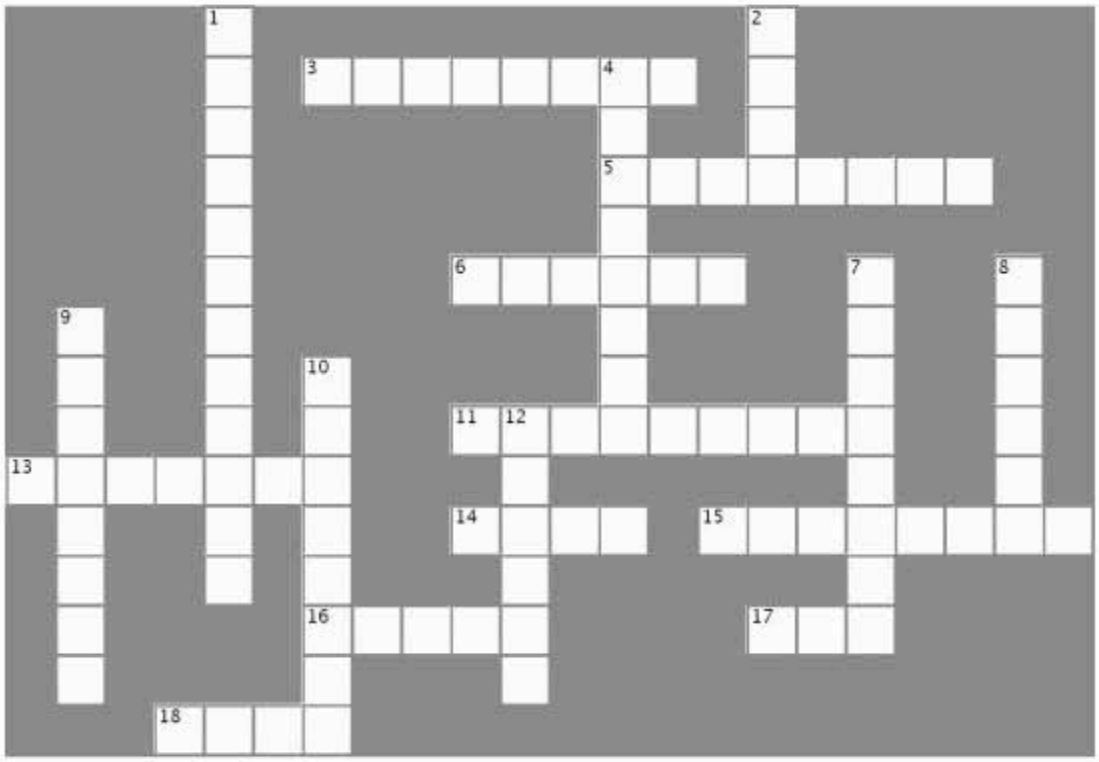
This graph monitors how quickly you and your team are completing your work, measured in days on the vertical axis. This chart then plots how quickly you tick off your work remaining against the number of days left in your iteration.

We'll talk a lot more about burn-down in the next few chapters.

Don't worry if you're still a little fuzzy on how burn-down rates work, and how to track it. You'll start creating a chart of your own in the next chapter, tracking your project's progress.

Software Development Planning Cross

Let's put what you've learned to use and stretch out your left brain a bit!
All of the words below are somewhere in this chapter: Good luck!



Across

3. At the end of an iteration you should get ..... from the customer.
5. Velocity does not account for ..... events.
6. Ideally you apply velocity ..... you break your Version 1.0 into iterations.
11. You should have one ..... per calendar month.
13. Every 3 iterations you should have a complete and running and releasable ..... of your software.
14. Velocity is a measuer of your .....'s work rate.
15. 0.7 is your first pass ..... for a new team.
16. At the end of an iteration your software should be .....
17. When priotitizing, the highest priority (the most important to the customer) is set to a value of .....
18. Any more than ..... people in a team and you run the risk of slowing your team down.

Down

1. Your customer can remove some less important user stories when ..... them.
2. Every 90 days you should ..... a complete version of your software.
4. The ..... sets the priority of each user stor.
7. The rate that you complete user stories across your entire project.
8. You should always try be ..... with the customer.
9. The set of features that must be present to have any working software at all is called the ..... functionality.
10. At the end of an iteration your software should be .....
12. You should assume ..... working days in a calendar month.

Tools for your Software Development Toolbox

Software Development is all about developing and delivering great software. In this chapter, you added several new techniques to your toolbox... For a complete list of tools in the book, see Appendix ii.

Development Techniques

Iterations should ideally be no longer than a month. That means you have 20 working calendar days per iteration

Applying velocity to your plan lets you feel more confident in your ability to keep your development promises to your customer

Use (literally) a big board on your wall to plan and monitor your current iteration's work

Get your customer's buy-in when choosing what user stories can be completed for Milestone 1.0, and when choosing what iteration a user story will be built in.

Development Principles

Keep iterations short and manageable

Ultimately, the customer decides what is in and what is out for Milestone 1.0

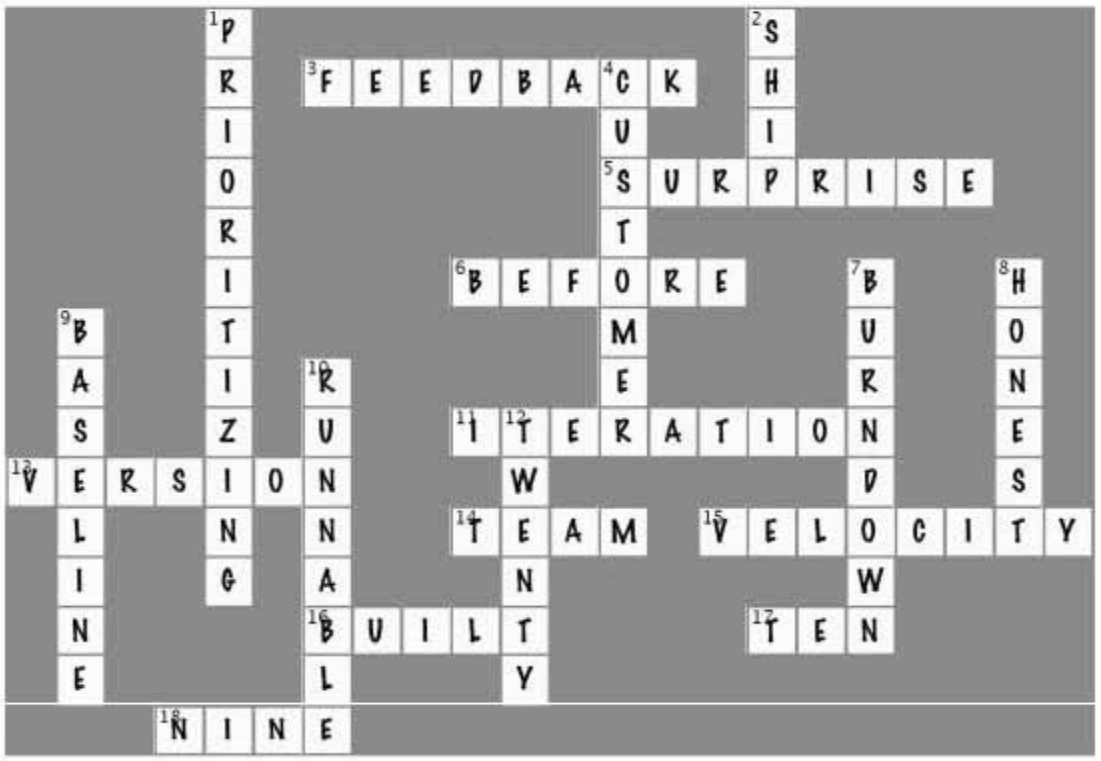
Promise, and deliver

ALWAYS be honest with the customer

BULLET POINTS

- Your **customer prioritizes** what is in and what is out for Milestone 1.0.
- Build **short iterations** of about 1 calendar month, **20 calendar days** of work.
- Throughout an iteration your software should be **buildable** and **runnable**.
- Apply your team's **velocity** to your estimates to figure out exactly how much work you can **realistically manage** in your first iteration.
- Keep your customers happy by coming up with a Milestone 1.0 that **you can achieve** so that you can be confident of delivering and getting paid. Then if you deliver more, they'll be even happier.

Software Development Planning Cross Solution



Getting to the real work

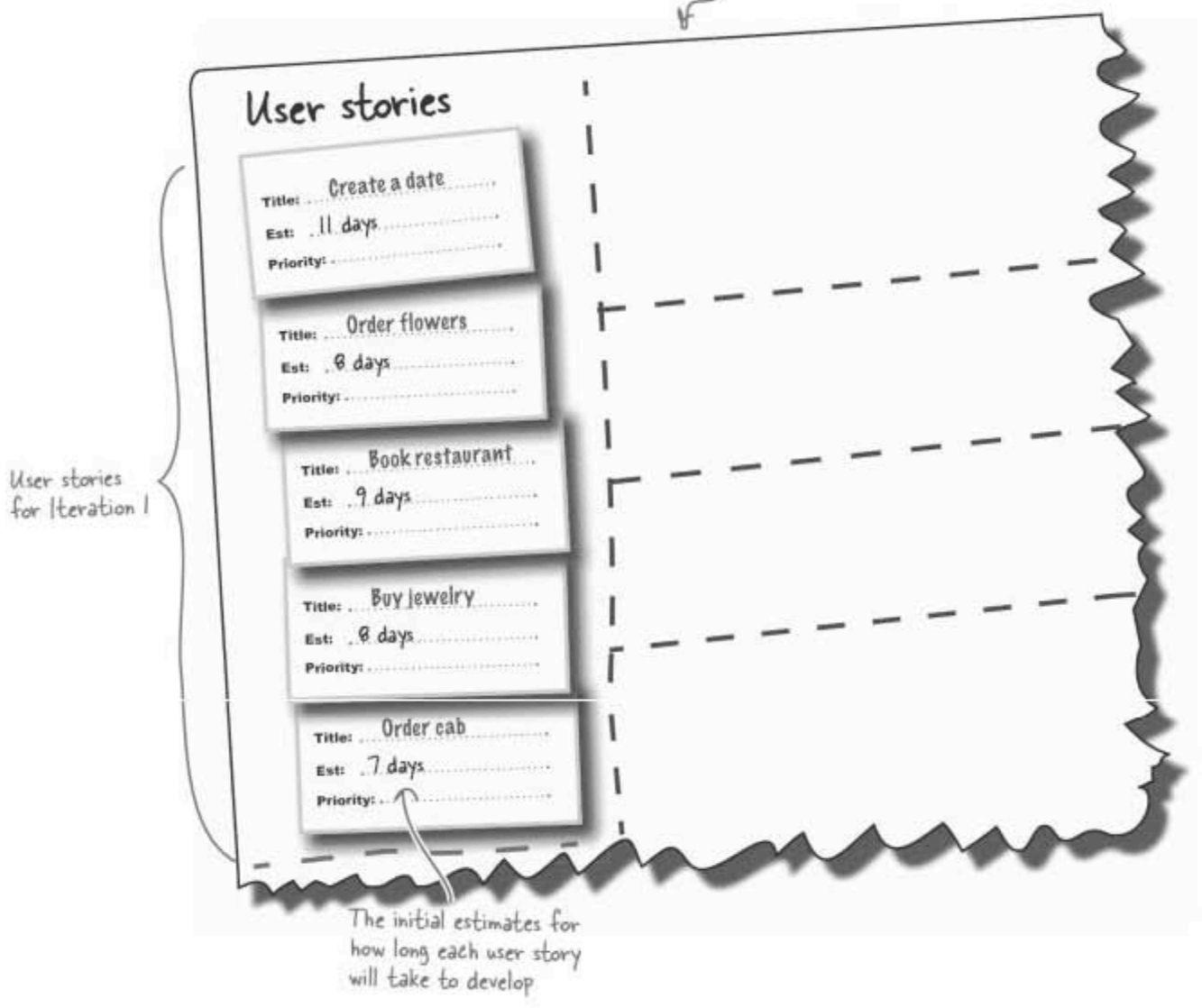
it's time to go to work. User stories capture what you need to develop, but now it's time to knuckle down and **dish out the work that needs to be done** so that you can bring those user stories to life. In this chapter you'll learn how to **break your user stories into tasks,** and how **task estimates** help you track your project from inception to completion. You'll learn how to update your board, moving tasks from in progress to complete, to finally **completing an entire user story**. Along the way, you'll handle and prioritize the inevitable **unexpected work** your customer will add to your plate.

Introducing iSwoon

Welcome to iSwoon, soon to be the world's finest desktop date planner! Here's the big board, already loaded with user stories broken down into 20-work-day iterations:

Date Rights Management (DRM) included

Part of the big board on your wall from Chapter 3



Exercise

It's time to get you and your team of developers working. Take each of the iSwoon user stories for Iteration 1 and assign each to a developer by drawing a line from the user story to the developer of your choice...

Title: Order flowers
Est: 8 days
Priority:

Title: Book restaurant
Est: 9 days
Priority:

Title: Create a date
Est: 11 days
Priority:

Title: Buy Jewelry
Est: 8 days
Priority:

Title: Order cab
Est: 7 days
Priority:

Wait a second, we can't just assign user stories to developers; things aren't that simple! Some of those user stories have to happen before others, and what if I want more than one developer on a single story?

Your work is more granular than your user stories.

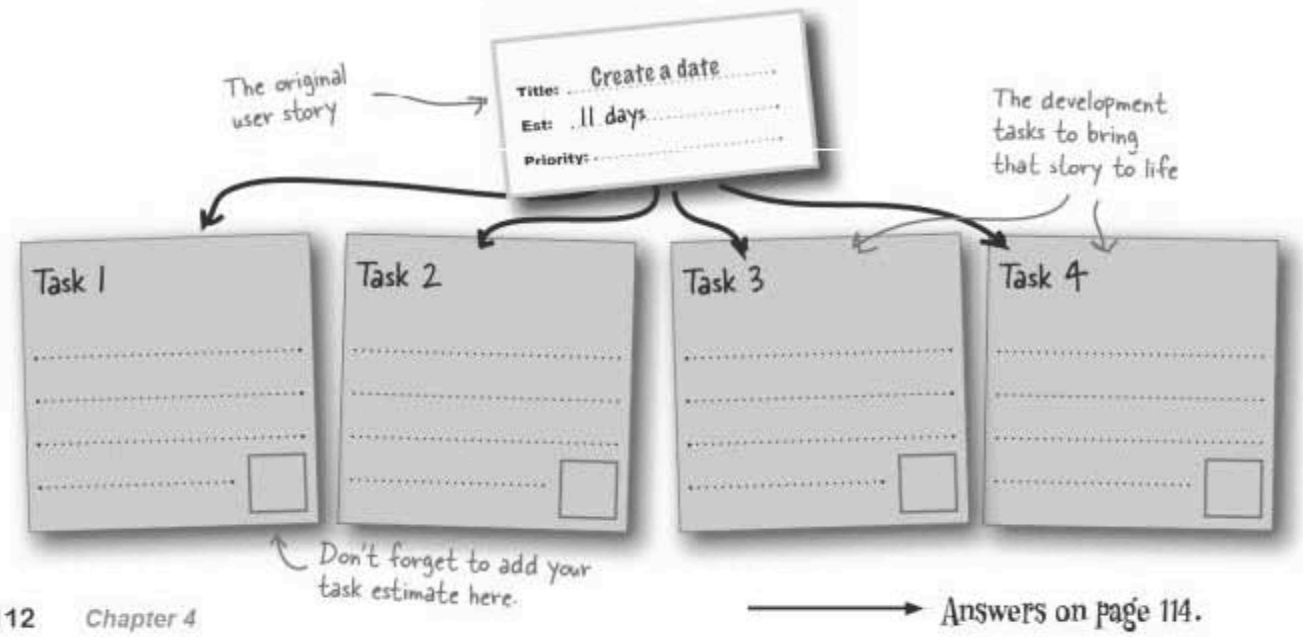
Your user stories were ***for your user***; they helped describe exactly what you software needed to do, from the customer's perspective. But now that it's time to start coding, you'll probably need to look at these stories differently. Each story is really a collection of specific **tasks**, small bits of functionality that can combine to make up one single user story.

A **task** specifies a piece of development work that needs to be carried out by ***one developer*** in order to construct part of a user story. Each task has a ***title*** so you can easily refer to it, a ***rough description*** that contains details about how the development of that task should be done, and an ***estimate***. Each task has its own estimate and—guess what—the best way to come up with those estimates is by playing planning poker again with your team.

We already used this to get estimates for user stories in Chapter 2, and it works for tasks, too.

Sharpen your pencil

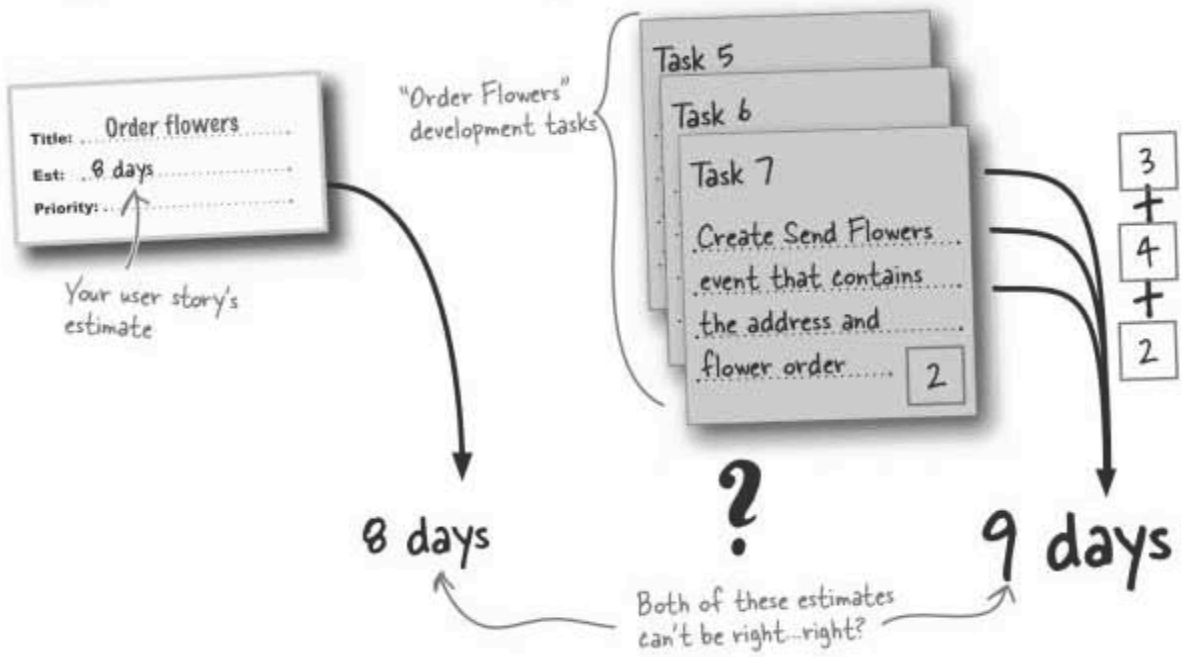
Now it's your turn. Take the user story of creating a date and break it into tasks you think you and your team need to execute. Write one task down on each of the sticky notes, and don't forget to add an estimate to each task.



Answers on page 114.

Do your tasks add up?

Did you notice a possible problem with your estimates? We've got a user story with an estimate, but now we're adding *new* estimates to our tasks. What happens when the two sets of estimates don't agree?



Task estimates add confidence to user story estimates

Your user story estimates kept you in the right ballpark when you were planning your iterations, but tasks really add another level of detail specific to the actual coding you'll do for a user story.

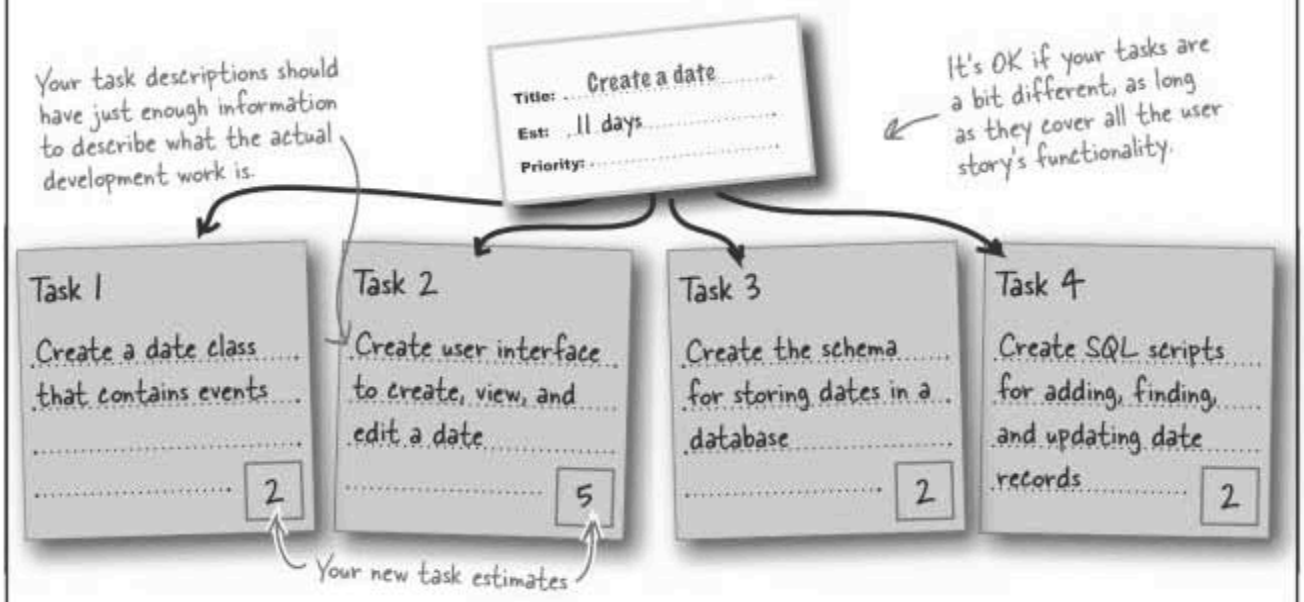
In fact, it's often best to break out tasks from your user stories right ***at the beginning*** of the estimation process, if you have time. This way you'll add even more confidence to the plan that you give your customer. ***It's always best to rely on the task estimates.*** Tasks describe the actual software development work that needs to be done and are far less of a guesstimate than a coarse-grained user story estimate.

Break user stories into tasks to add CONFIDENCE to your estimates and your plan.

And the earlier you can do this, the better.

Sharpen your pencil Solution

You were asked to take the user story of creating a date and break out the tasks that you think you and your team will need to execute to develop this user story, not forgetting to add task estimates...



there are no Dumb Questions

Q: My tasks add up to a new estimate for my user story, so were my original user story estimates wrong?

A: Well, yes and no. Your user story estimate was accurate enough in the beginning to let you organize your iterations. Now, with task estimates, you have a set of **more accurate data** that either backs up your user story estimates or conflicts with them.

You always want to rely on data that you trust, the estimates that you feel are most accurate. In this case, those are your task estimates.

Q: How big should a task estimate be?

A: Your task estimates should ideally be between 1/2 and 5 days in length. A shorter task, measured in hours, is too small a task. A task that is longer than five days spreads across more than one working week, and that gives the developer working on the task too much time to lose focus.

Q: What happens when I discover a big missing task?

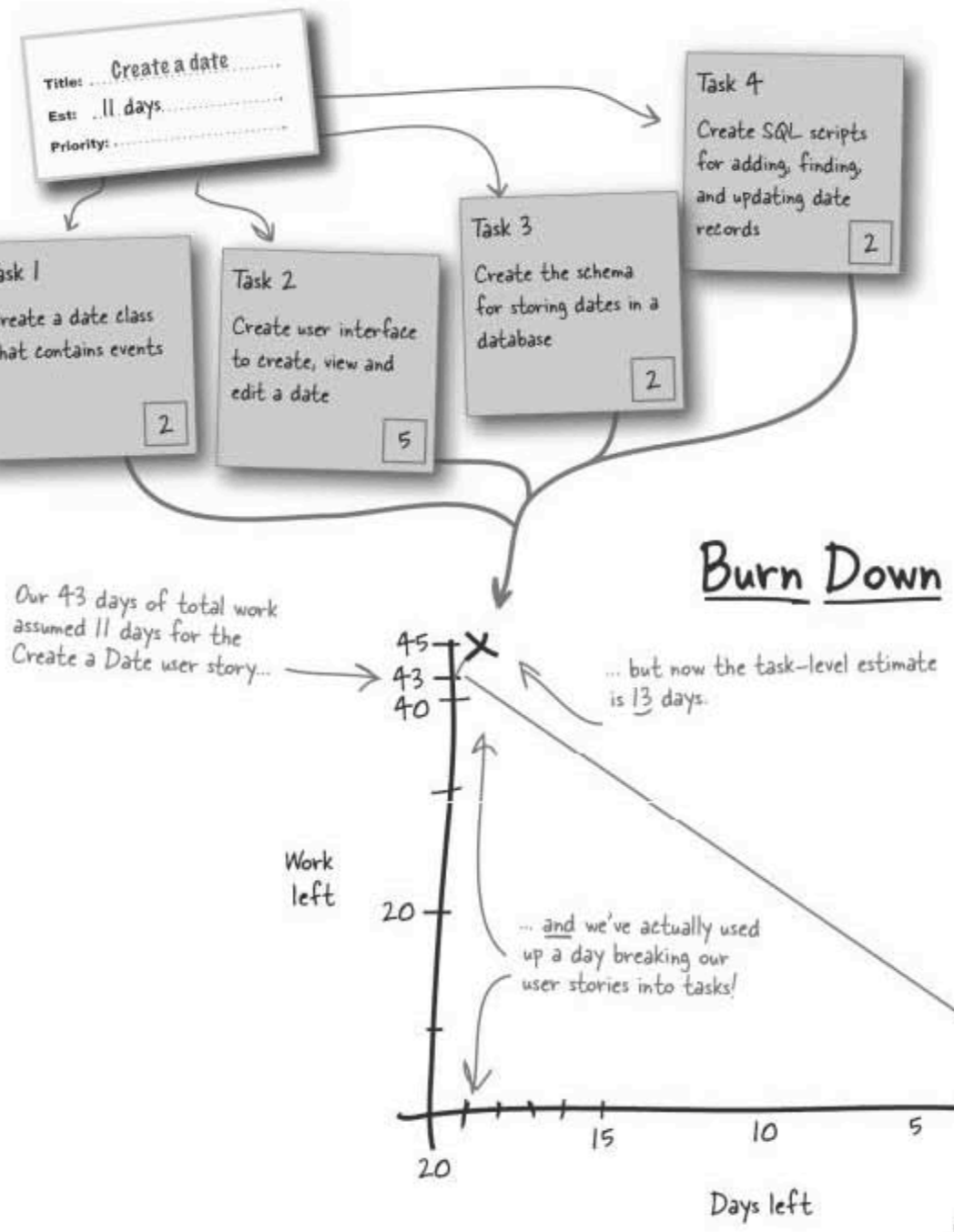
A: Sometimes—hopefully not too often—you'll come across a task that just breaks your user story estimate completely. You might have forgotten something important when first coming up with the user story estimates, and suddenly the devil in the details rears its ugly head, and you have a more accurate, task-based estimate that completely blows your user story estimate out of the water.

When this happens you can really only do one thing, and that's adjust your iteration. To keep your iteration within 20 working days, you can postpone that large task (and user story) until the next iteration, reshuffling the rest of your iterations accordingly.

To avoid these problem, you could break your user stories into tasks earlier. For instance, you might break up your user stories into tasks when you initially plan your iterations, always relying on your task estimates over your original user story estimates as you balance out your iterations to 20 working days each.

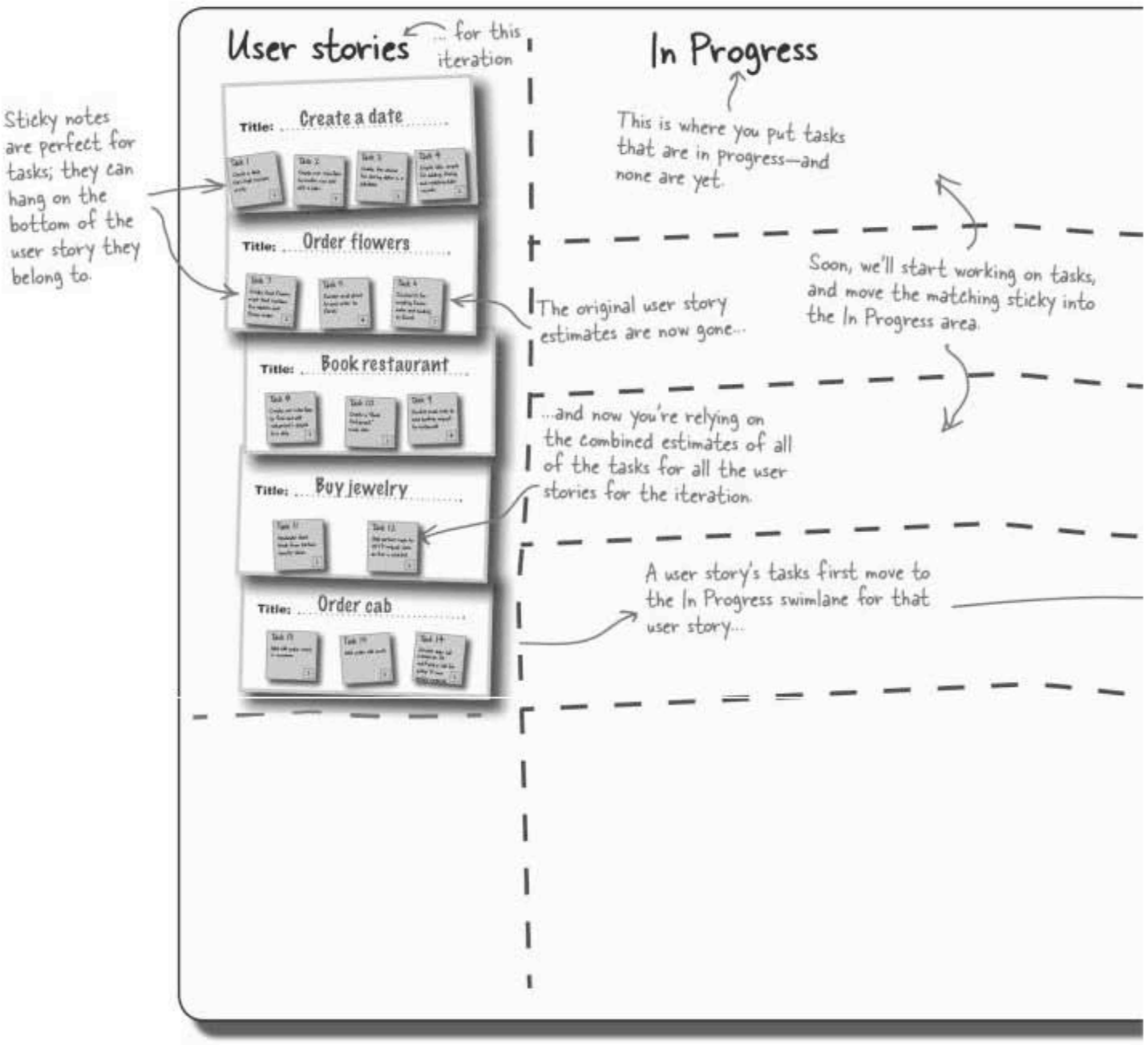
Plot just the work you have left

Remember that burn-down rate chart from Chapter 3? Here's where it starts to help us track what's going on in our project. Every time we do any work or review an estimate, we update our new estimates, and the time we have left, on our burn-down chart:



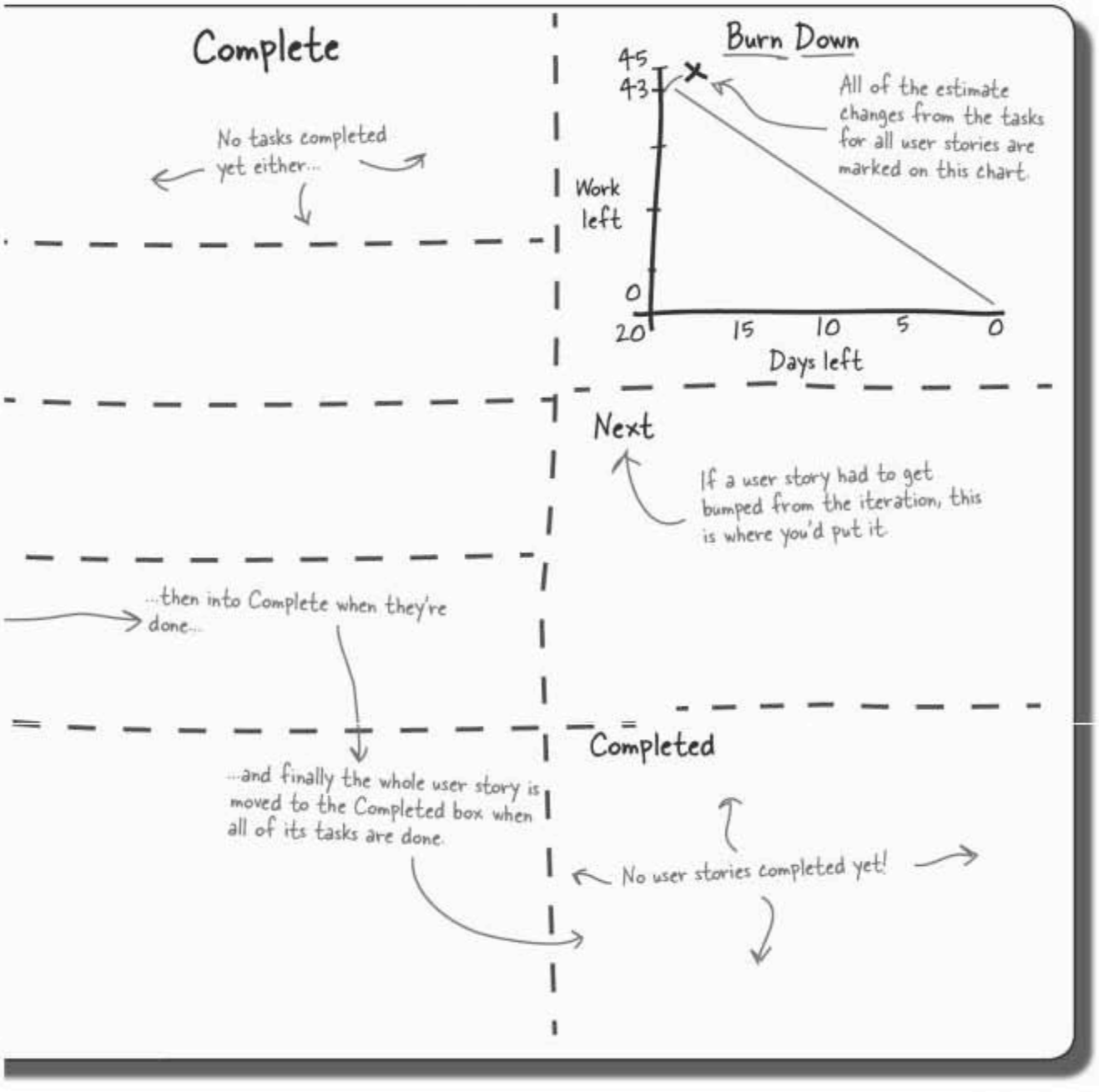
Add your tasks to your board

You and your team are now almost ready to start working on your tasks, but first you need to update the big board on your wall. Add your task sticky notes to your user stories, and also add an In Progress and Complete section for tracking tasks and user stories:



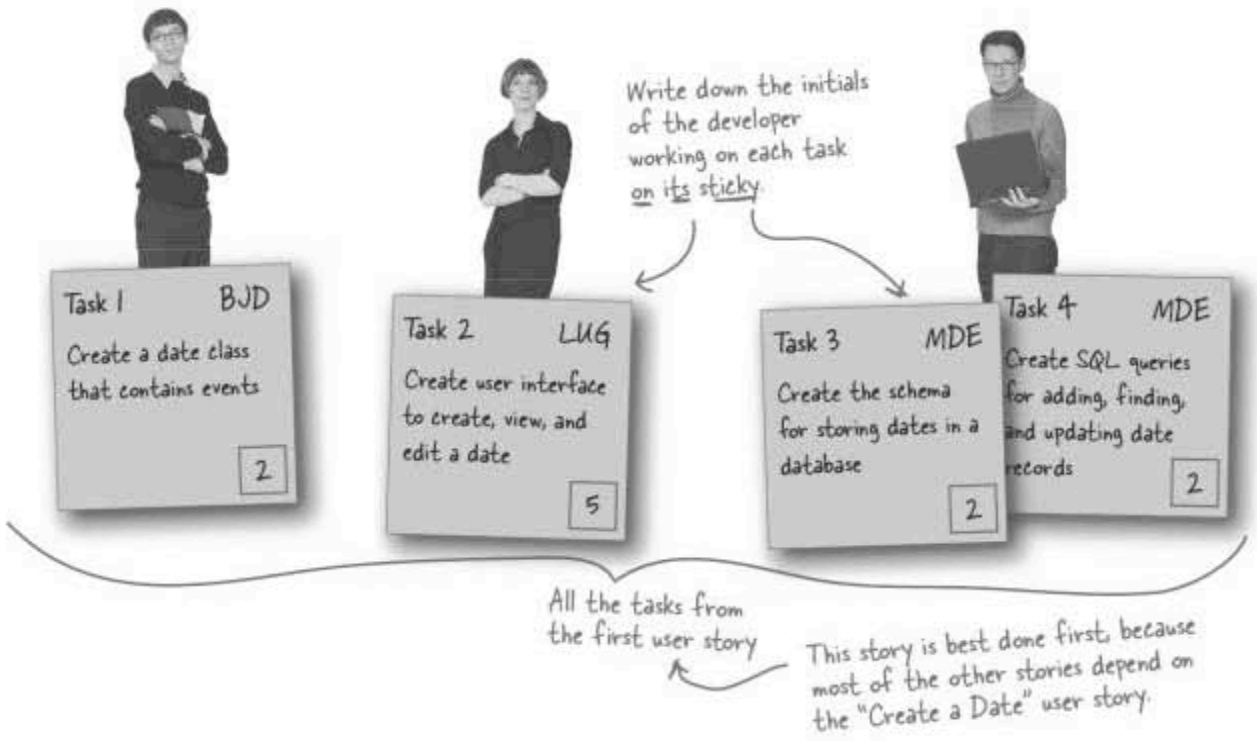
This isn't a virtual board—it should be a real bulletin or whiteboard hanging somewhere, like a common area or maybe the office where you and your team meet each morning.

Yes, you should meet each morning! More on that in just a minute...



Start working on your tasks

It's time to bring that burn-down rate back under control by getting started developing on your first user story. And, with small tasks, you can assign your team work in a sensible, trackable way:



there are no Dumb Questions

Q: How do I figure out who to assign a task to?

A: There are no hard-and-fast rules about who to give a task to, but it's best to just apply some common sense. Figure out who would be most productive or—if you have the time, will learn most from a particular task by looking at their own expertise—and then allocate the task to the best-suited developer, or the one who will gain the most, that's not already busy.

Q: Why allocate tasks just from the first user story. Why not take one task from each user story?

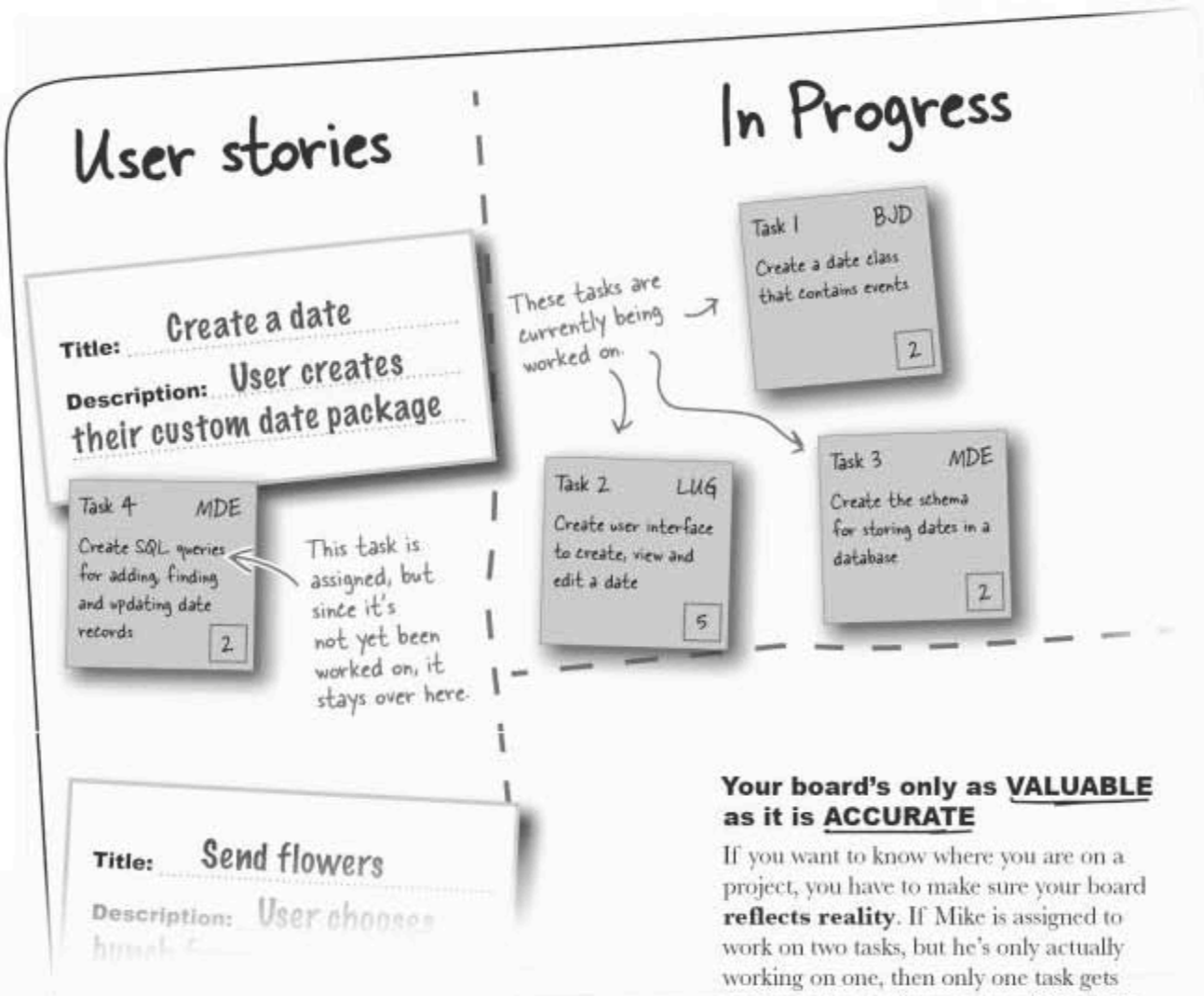
A: One good reason is so that so that you don't wind up with five stories in a half-done state, and instead can wrap up a user story and move on to the next. If you've got one story your other stories depend on, you may want to get all that first story's tasks done at once. However, if your stories are independent of each other, you may work on tasks from multiple stories all at the same time.

Q: I'm still worried about that burn-down rate being way up, is there anything I can do right now to fix that?

A: A burn-down rate that's going up is always a cause for concern, but since you're early on, let's wait a bit and see if we catch up.

A task is only in progress when it's IN PROGRESS

Now that everyone's got some work to do, it's time to move those task stickies off of user story cards, and onto the In Progress area of your big board. But you only put tasks that are **actually being worked on** in the In Progress column—even if you already know who'll be working on tasks yet to be tackled.

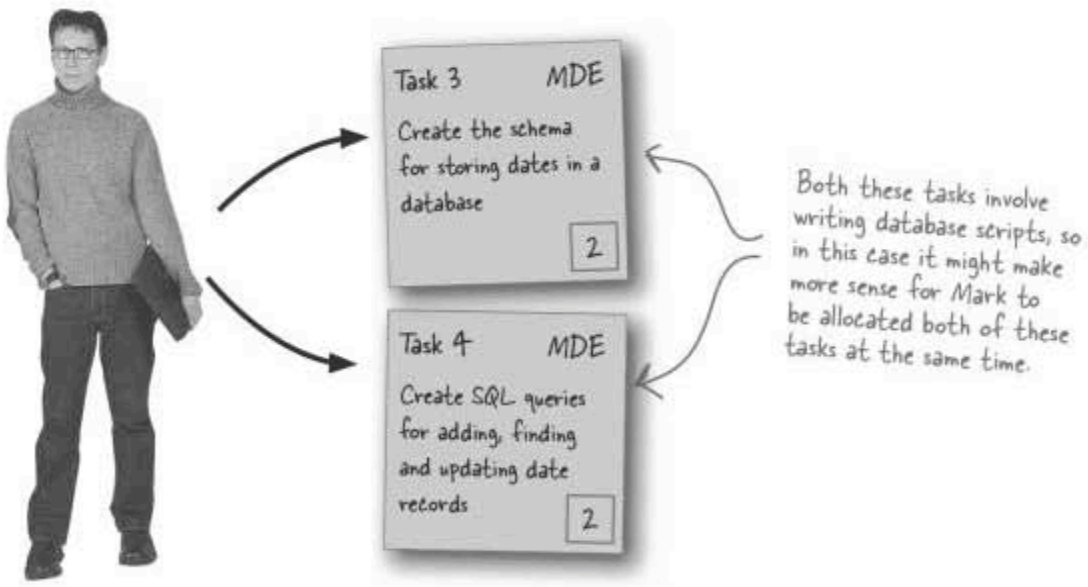


Your board's only as VALUABLE as it is ACCURATE

If you want to know where you are on a project, you have to make sure your board **reflects reality**. If Mike is assigned to work on two tasks, but he's only actually working on one, then only one task gets put up in the In Progress area. Otherwise, it looks like more is being worked on than really is.

What if I'm working on two things at once?

Not all tasks are best executed in isolation. Sometimes two tasks are related, and, because there is so much overlap, it's actually more work to tackle one, and then the other separately. In these cases the most productive thing to do is work on those tasks ***at the same time***...



Sometimes working on both tasks at the same time IS the best option

When you have two tasks that are closely related. then it's not really a problem to work on them both at the same time.

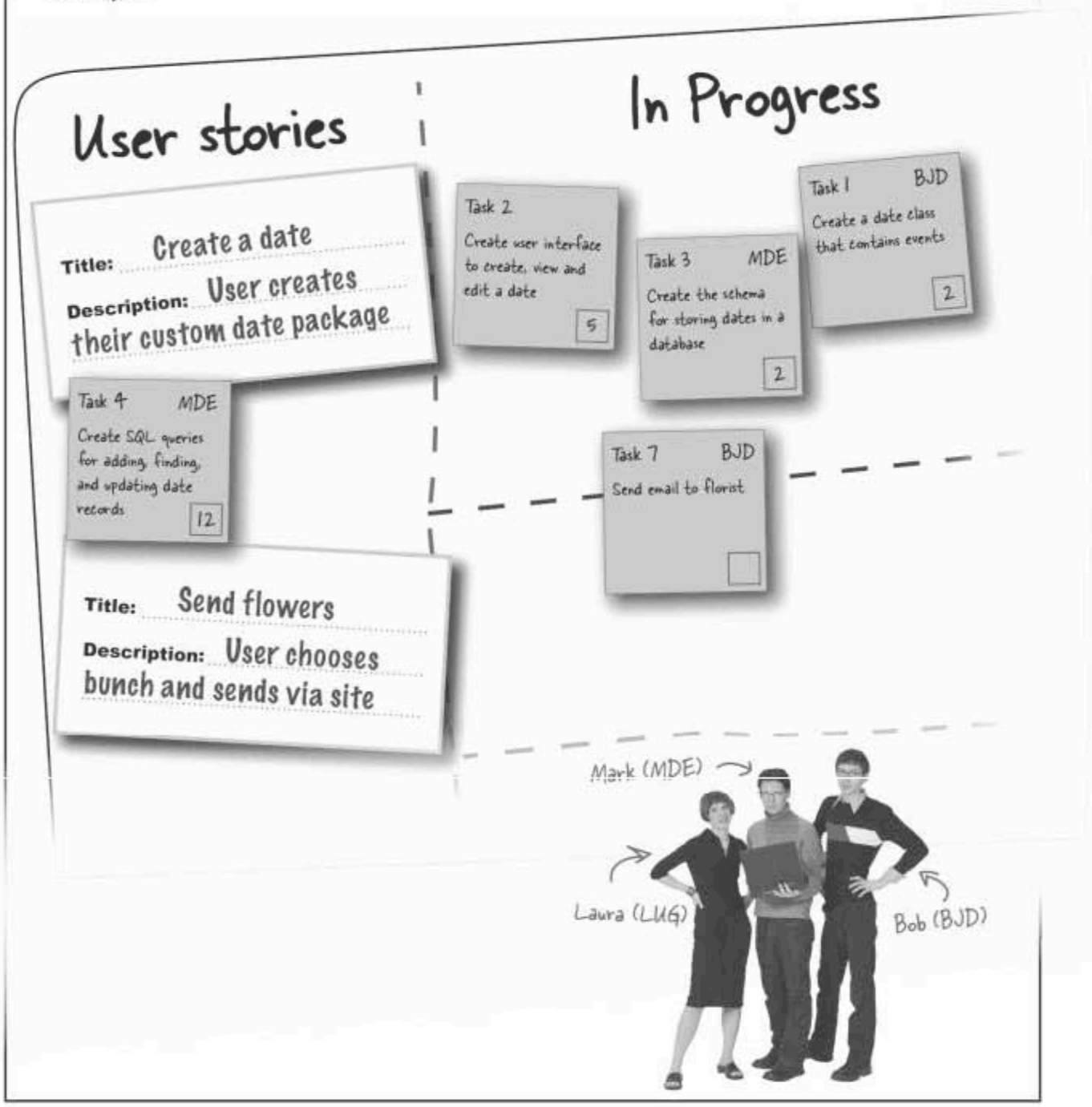
This is especially the case where the work completed in one task could ***inform decisions*** made in the work for another task. Rather than completing one task and starting the next, and then realizing that you need to do some work on the first task again, it is far more efficient to work both tasks at the same time.

Rules of Thumb

- Try to double-up tasks that are related to each other, or at least focus on roughly the same area of your software. The less thought involved in moving from one task to another, the faster that switch will be.
- Try not to double-up on tasks that have large estimates. It's not only difficult to stay focused on a long task, but you will be more confident estimating the work involved the shorter the task is.

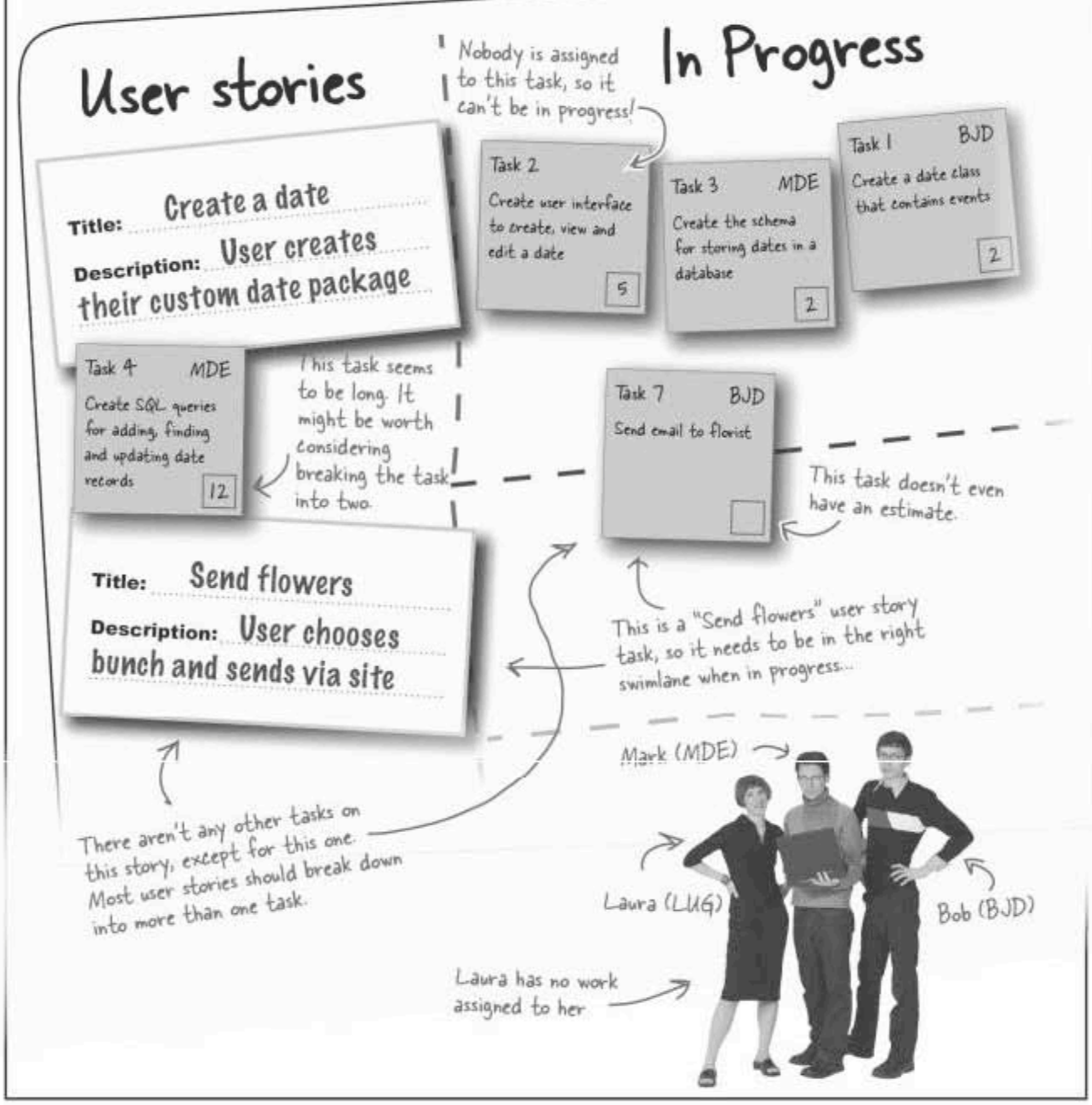
Exercise

Someone's been tampering with the board and things are a real mess. Take a look at the project below and annotate all of the problems you can spot.



Exercise Solution

Your job was to to take a look at the project below and annotate all of the problems you could spot...



A meeting so quick you don't even have time to sit down.

Your first standup meeting...

You've now got some tasks in progress, and so to keep everyone in the loop, while not taking up too much of their time, you conduct a quick standup meeting every day.

Mark: So, we've all had our tasks for one day now. How are we doing?

Bob: Well, I haven't hit any big problems yet, so nothing new really to report.

Mark: That's great. I've had a bit of success and finished up on the scripts to create tables in the database...

Laura: Things are still in progress on my user interface task.

Mark: OK, that all sounds good, I'll update the board and move my task into Completed. We can update the burn rate, too; maybe we're making up for some of that time we lost earlier. Any other successes or issues to report?

Bob: Well, I guess I should probably mention that I'm finding creating the right Date class a little tricky...

Mark: That's fine. I'm really glad you brought it up, though. That's a two-day task and we need it done tomorrow, so I'll get you some help on that as soon as possible. OK, it's been about seven minutes, I think we're done here...

Your daily standup meetings should:

- **Track your progress.** Get everyone's input about how things are going.
- **Update your burn-down rate.** It's a new day so you need to update your burndown rate to see how things are going.
- **Update tasks.** If a task is completed then it's time to move it over into the Completed area and check those days off of your burn-down rate.
- **Talk about what happened yesterday and what's going to happen today.** Bring up any successes that happened since yesterday's standup meeting and make sure everyone knows what they're doing today.
- **Bring up any issues.** The standup meeting is not a place to be shy, so encourage everyone to bring up any problems they've encountered so that you all as a team can start to fix those problems.
- **Last between 5 and 15 minutes.** Keep things brief and focused on the short-term tasks at hand.

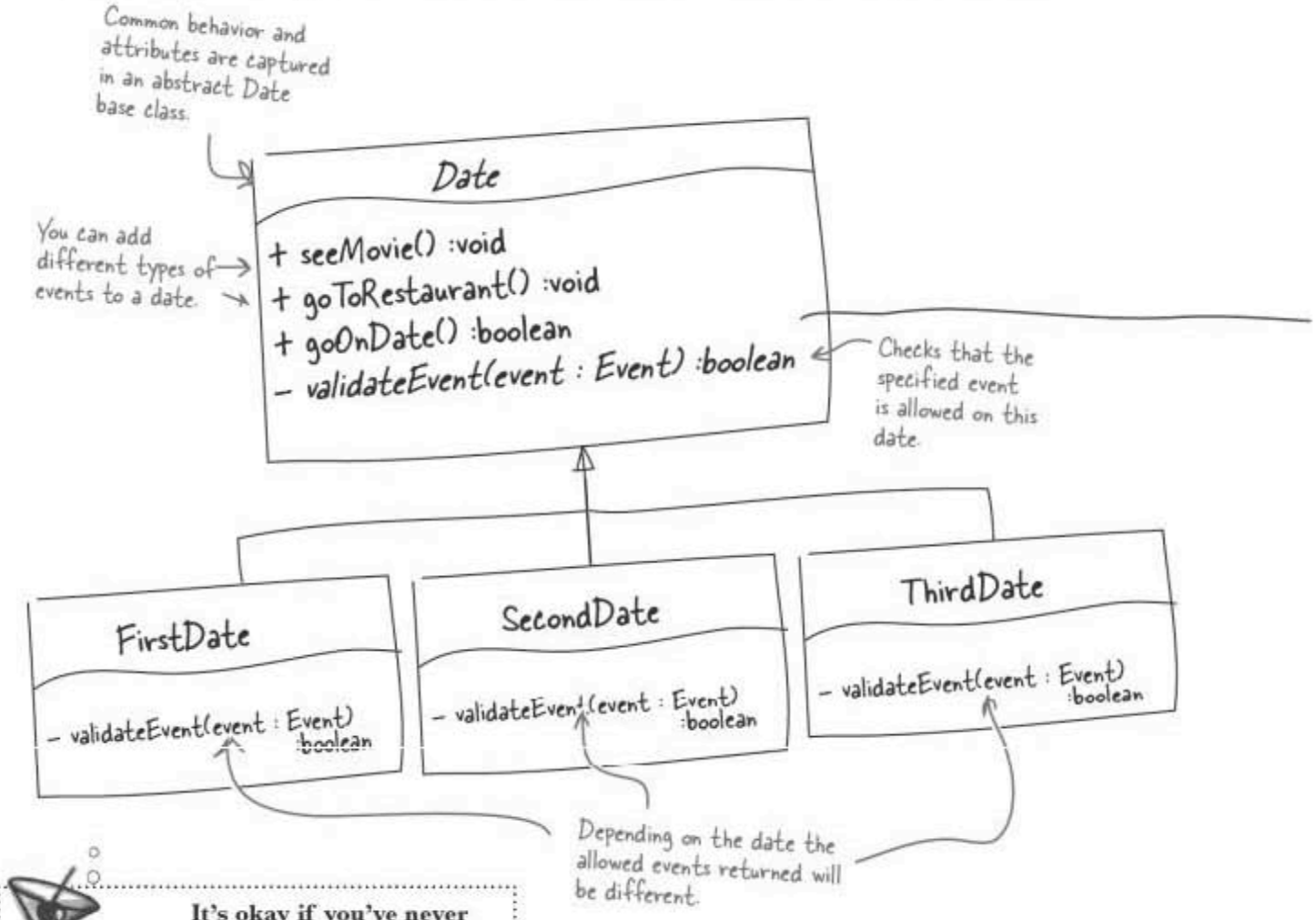
A daily standup meeting should keep everyone motivated, keep your board up-to-date, and highlight any problems early.

Task 1: Create the Date class

Bob's been busy creating the classes that bring the "Create a Date" user story to life, but he needs a hand. Here's a UML class diagram that describes the design he's come up with so far.

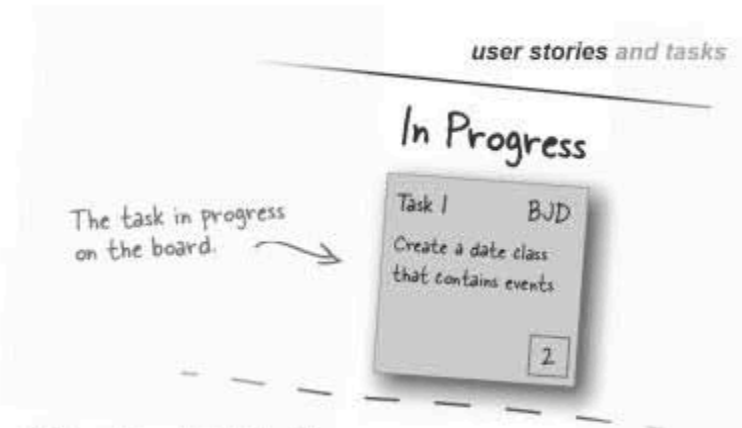
A UML class diagram shows the classes in your software and how they relate to each other.

The Date class is split into three classes, one class for each type of date...

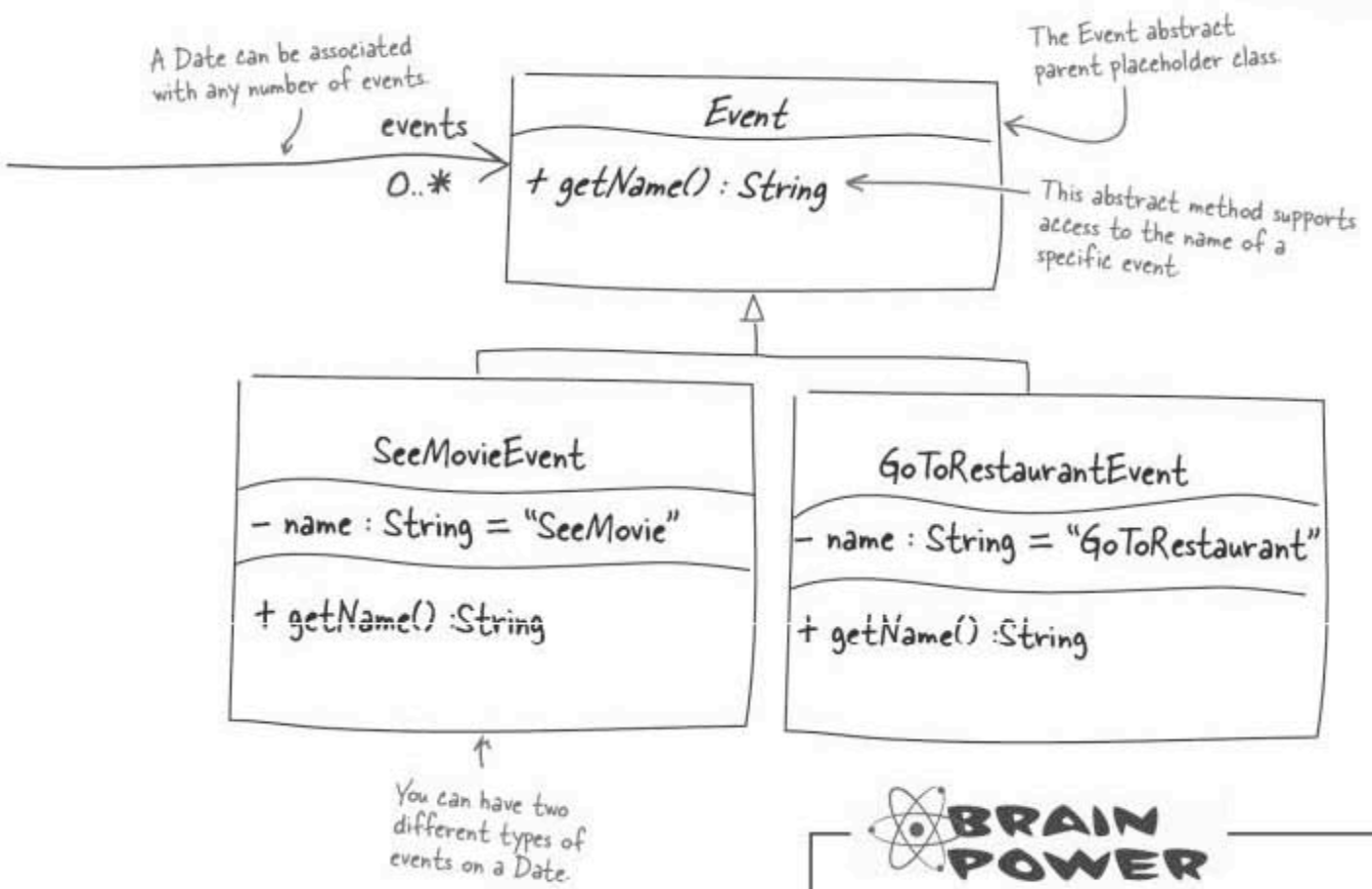


It's okay if you've never seen UML before!

Don't worry if you don't know your UML class diagrams from your sequences; there's a short overview in Appendix i to help you get comfortable with UML notation as quickly as possible.



Each Date can then have a number of Events added to it...



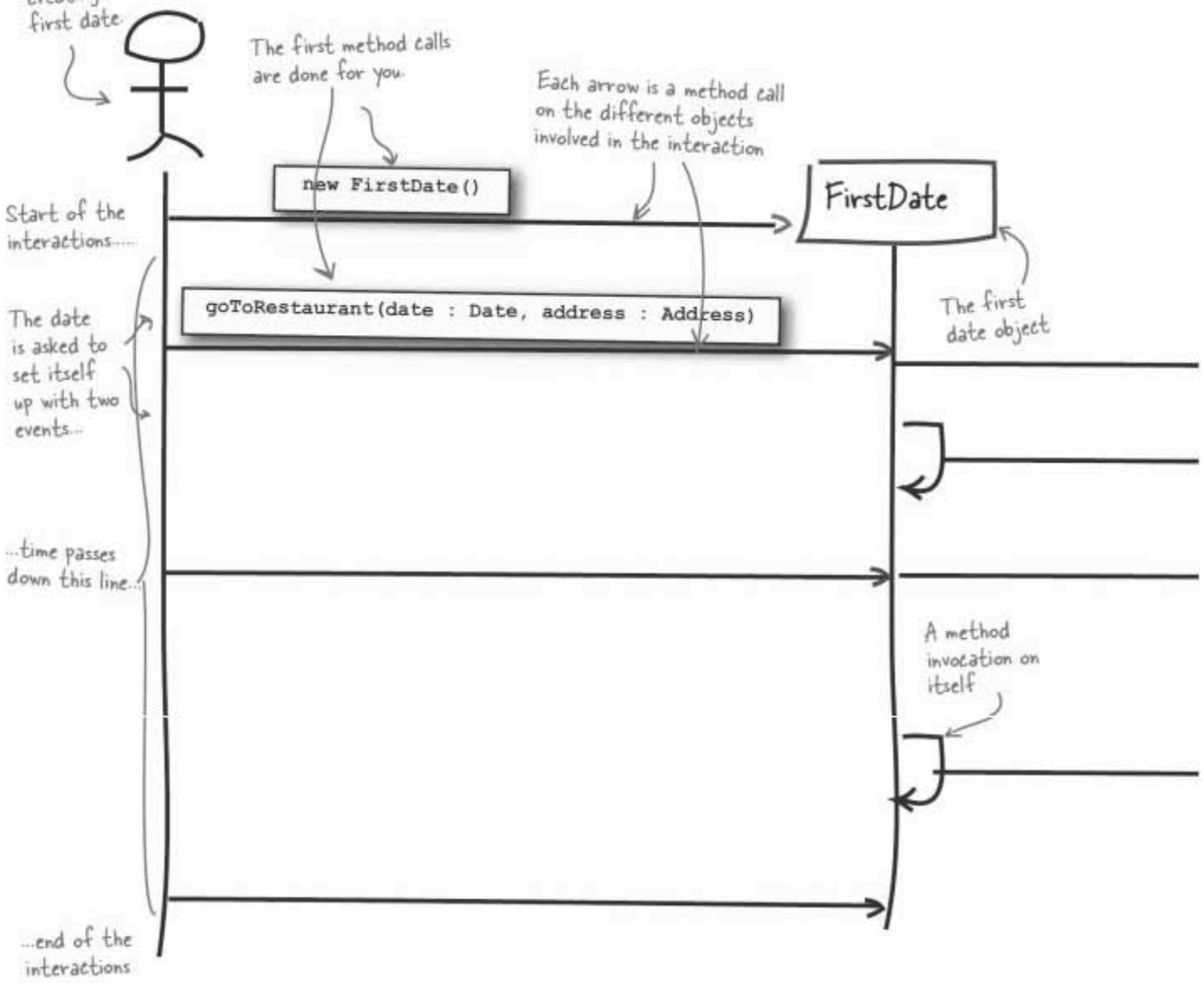
BRAIN POWER

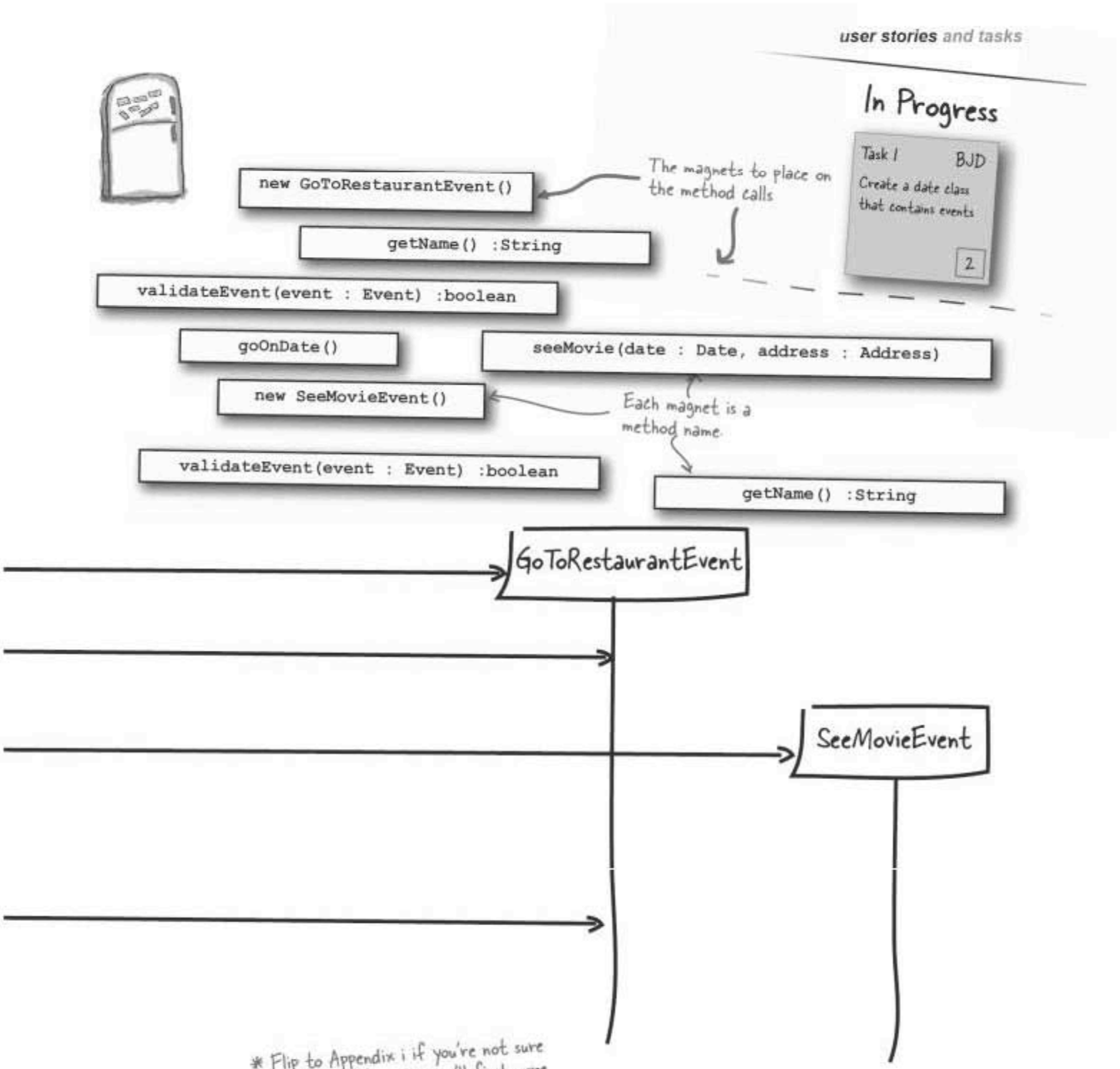
What do you think of this design?

Task 1: Creating dates

A diagram that brings objects to life, showing how they work together to make an interaction happen

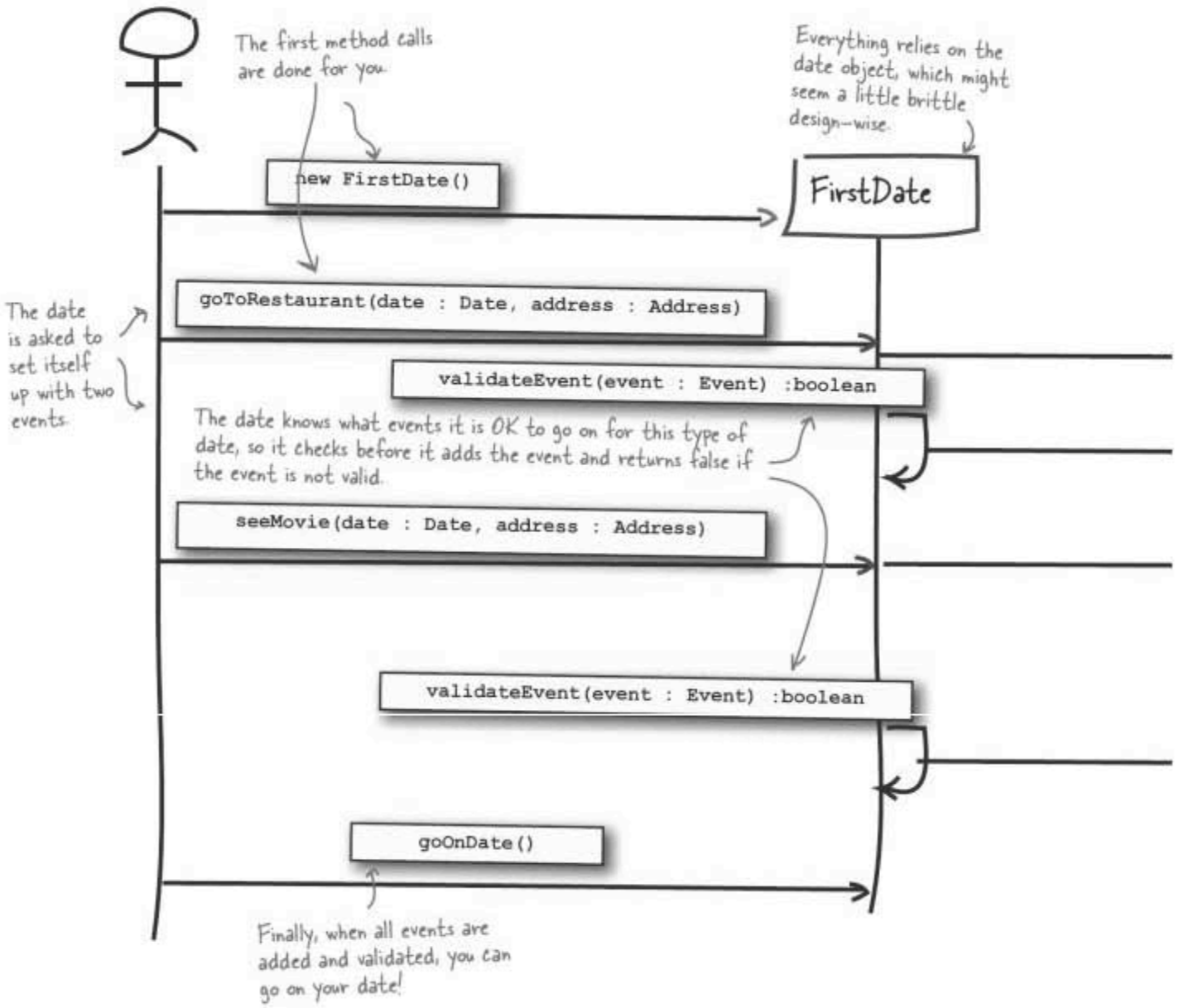
Let's test out the Date and Event classes by bringing them to life on a sequence diagram. Finish the sequence diagram by adding the right method names to each interaction between objects so that you are creating and validating that a first date that has two events, going to a restaurant and seeing a movie.

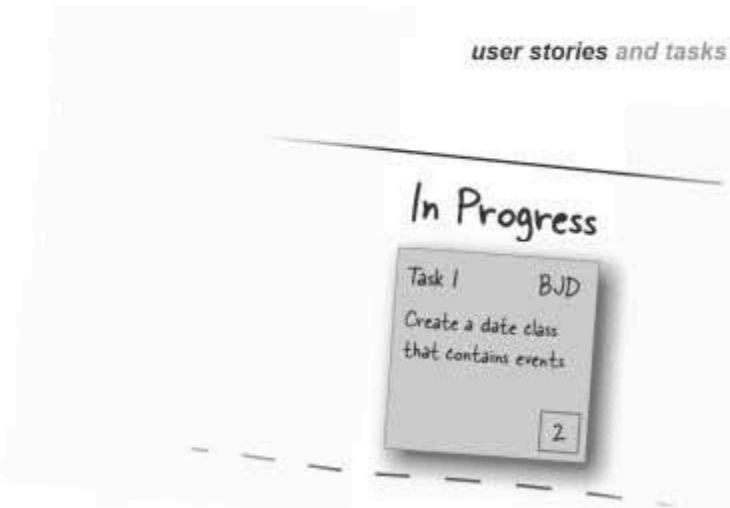


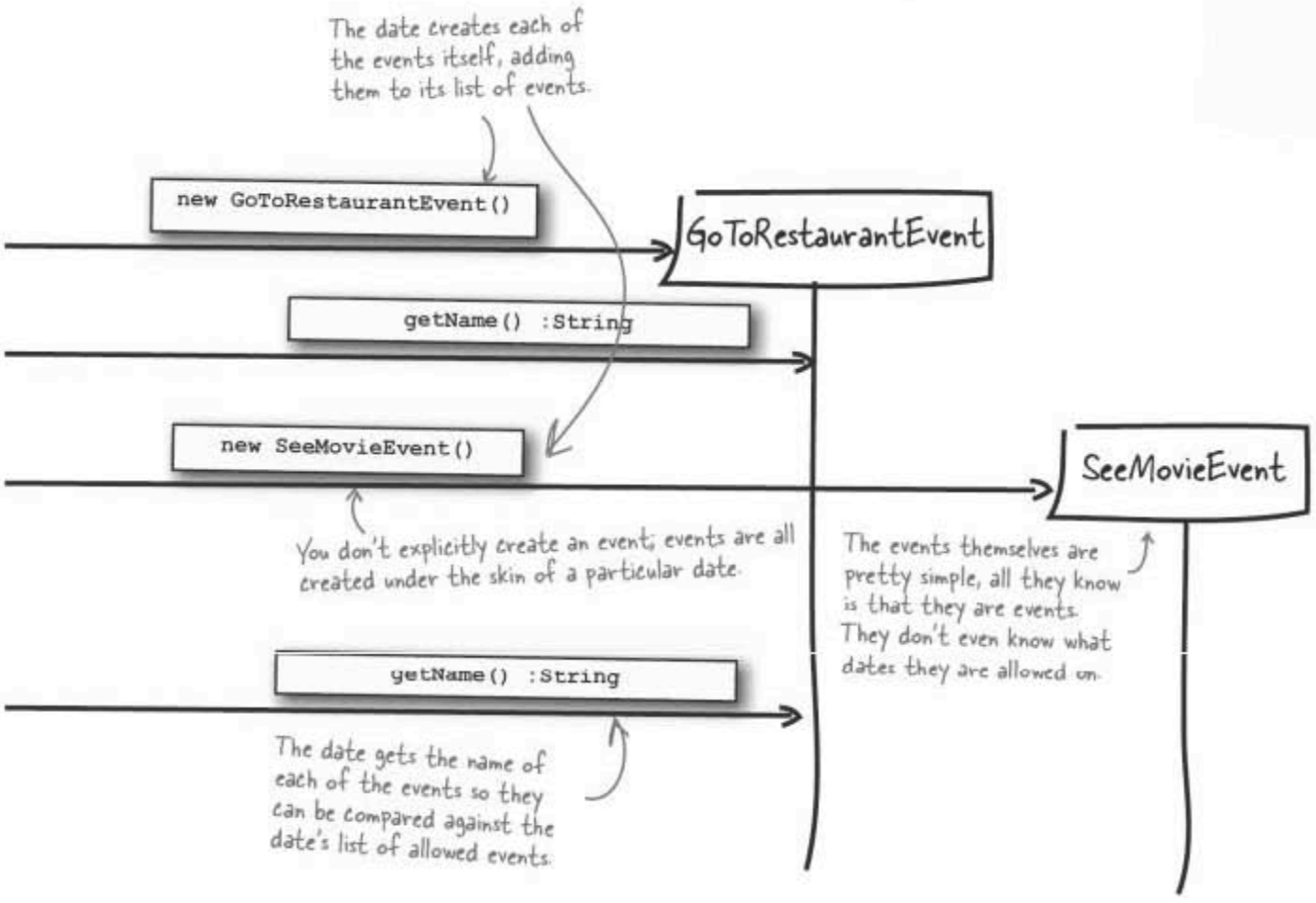


* Flip to Appendix i if you're not sure what this stuff means; you'll find more on UML class diagrams and sequence diagrams there.

Your job was to test out the Date and Event classes by bringing them to life on a sequence diagram. You should have finished the sequence diagram so that you plan and go on a first date with two events, going to a restaurant and seeing a movie.




In Progress
Task 1
BJD
Create a date class that contains events
2


The date creates each of the events itself, adding them to its list of events.
new GoToRestaurantEvent()
GoToRestaurantEvent
getName() :String
new SeeMovieEvent()
SeeMovieEvent
You don't explicitly create an event; events are all created under the skin of a particular date.
The events themselves are pretty simple, all they know is that they are events. They don't even know what dates they are allowed on.
getName() :String
The date gets the name of each of the events so they can be compared against the date's list of allowed events.

Standup meeting: Day 5, end of Week 1...

Bob: Well, I finally got the date class finished with a little help, ran late by a day though...

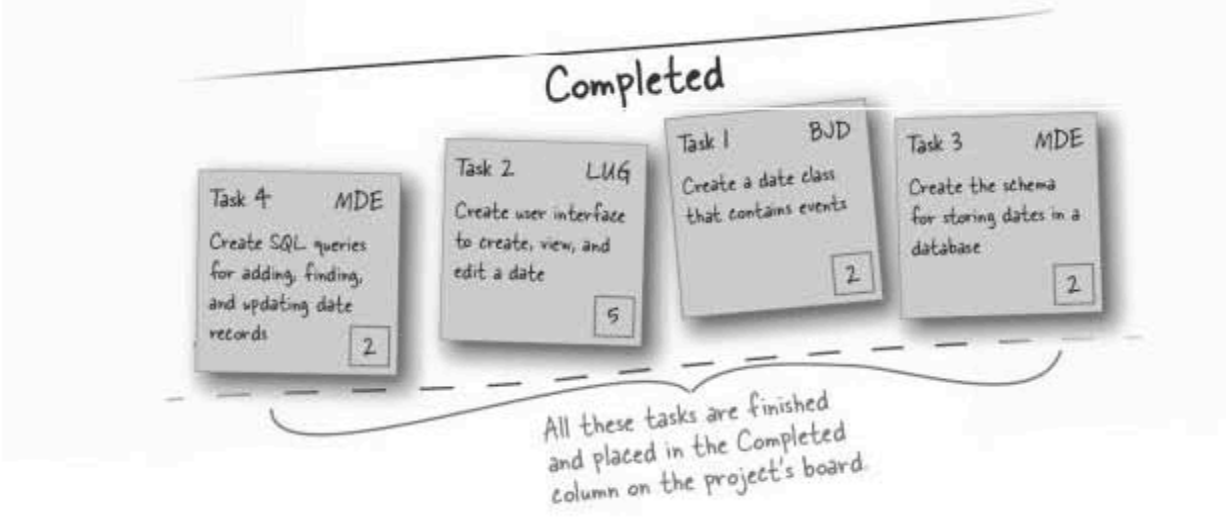
Laura: That's OK, this time around. We can hopefully make some of that time up later.

Mark: All work on the database is now done; I'm all set for the next set of tasks.

Laura: Great, and I got my work done on the user interface pieces, so we've actually got something running.

Bob: Always a good week when you head out of the office with something working...

Laura: Absolutely. OK, it's time to update the board and our burn-down rate to get things set up for next week.



there are no Dumb Questions

Q: Do I REALLY have to get everyone to stand up during a standup meeting?

A: No, not really. A standup meeting is called "standup" because it is meant to be a fast meeting that lasts a **maximum** of 15 minutes; you should ideally be aiming for 5 minutes.

We've all been stuck in endless meetings where nothing gets done, so the idea with a standup meeting is to keep things so short you don't even have time to find chairs. This keeps the focus and the momentum on only two agenda items:

- Are there any issues?
- Have we finished anything?

With these issues addressed, you can update your project board and get on with the actual development work.

Q: An issue has come up in my standup meeting that is going to take some discussion to resolve. Is it OK to lengthen the standup meeting to an hour to solve these bigger problems?

A: Always try to keep a standup meeting to less than 15 minutes. If an issue turns out to be something that requires further discussion, then schedule another meeting ***specifically for that issue***. The standup meeting has highlighted the issue, and so it's done its job.

Q: Do standup meetings have to be daily?

A: It certainly helps to make your standup meetings daily. With the pace of modern software development, issues arise on almost a daily basis, so a quick 15 minutes with your team is essential to keeping your finger on the pulse of the project.

Q: Is it best to do a standup meeting in the morning or the afternoon?

A: Ideally, standup meetings should be first thing in the morning. The meeting sets everyone up for the day's tasks and gives you time to hit issues straight away.

Still, there may be situations when you can't all meet in the morning, especially if you have remote employees. In those cases, standup meetings should be conducted when the majority of your team begin their working day. This isn't ideal for everyone, but at least most people get the full benefit of early feedback from the meeting.

On rare occasions, you can split the standup meeting in two. You might do this if part of your team works in a completely different time zone. If you go with this approach, keeping your board updated is even more critical, as this is the place where everyone's status from the standup meeting is captured for all to see.

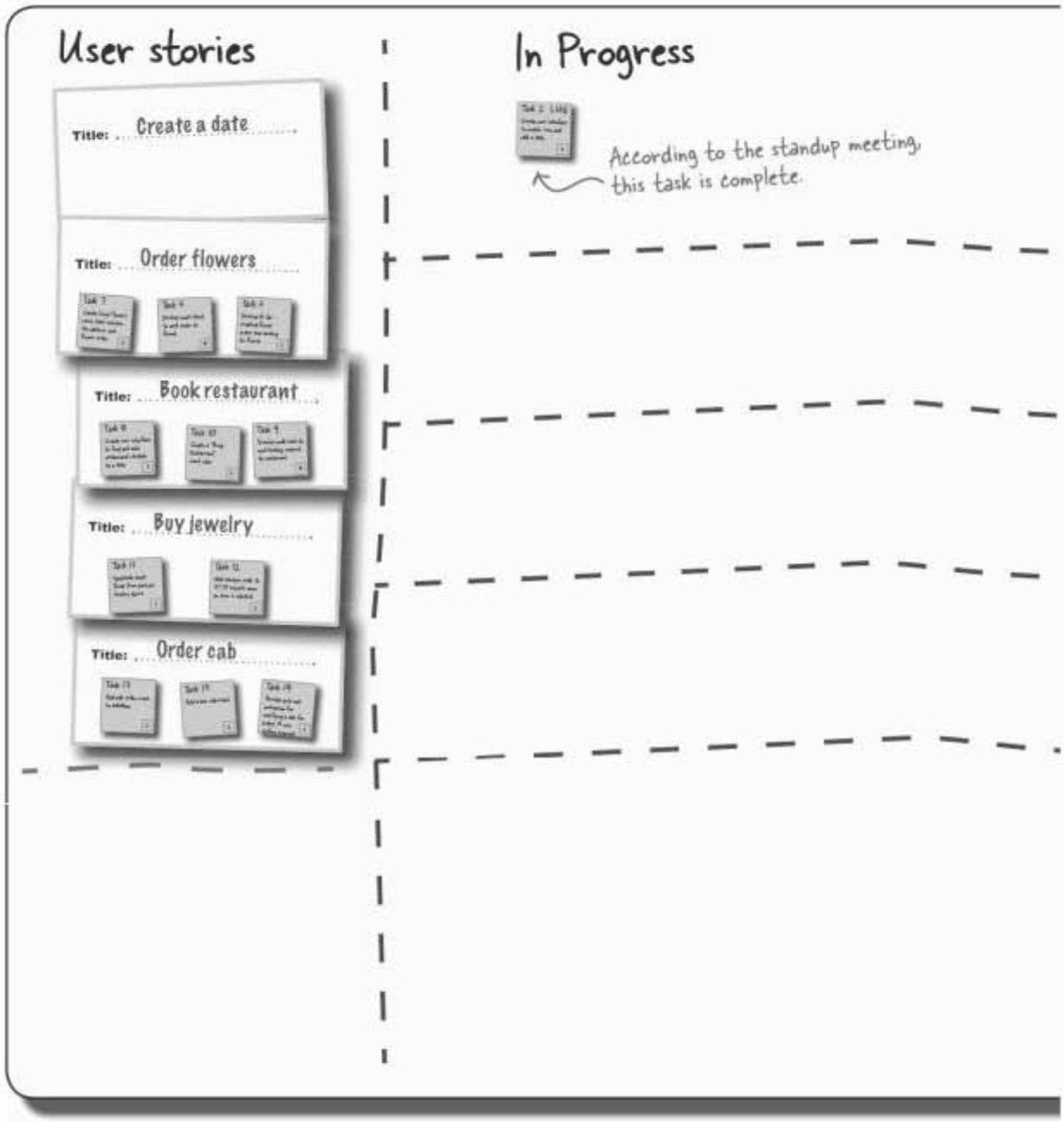
Standup meetings keep your peers, employees, and managers up to date, and keep your finger on the pulse of how your development work is going.

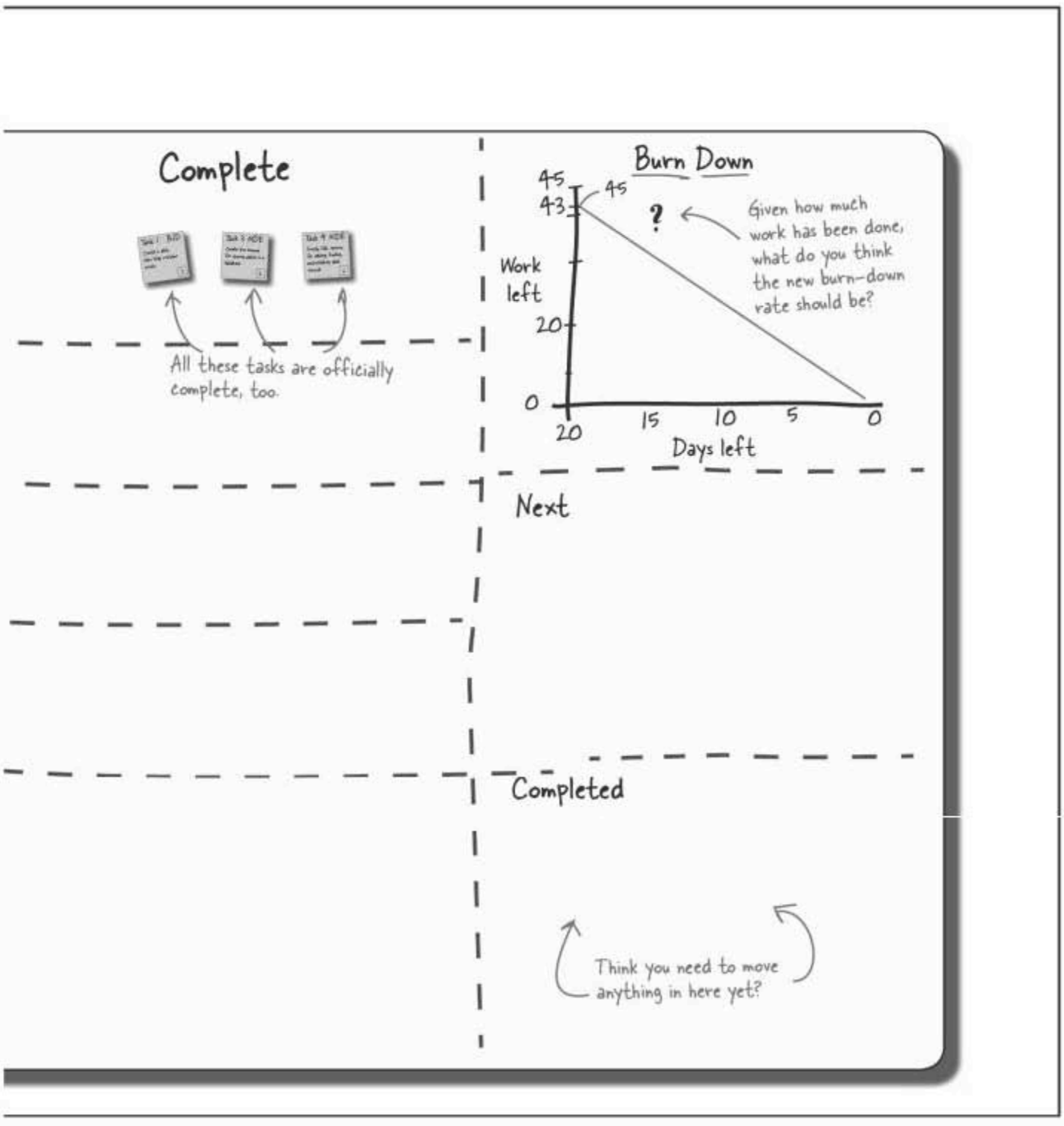
BULLET POINTS

- Organize **daily standup meetings** to make sure you catch issues early.
- Keep standup meetings **less than 15 minutes**.
- A standup meeting is all about **progress**, **problematic issues**, and **updating your board**.
- Try to schedule your standup meetings for the **morning** so that everyone knows where they are at the **beginning of the working day**.

Long Exercise

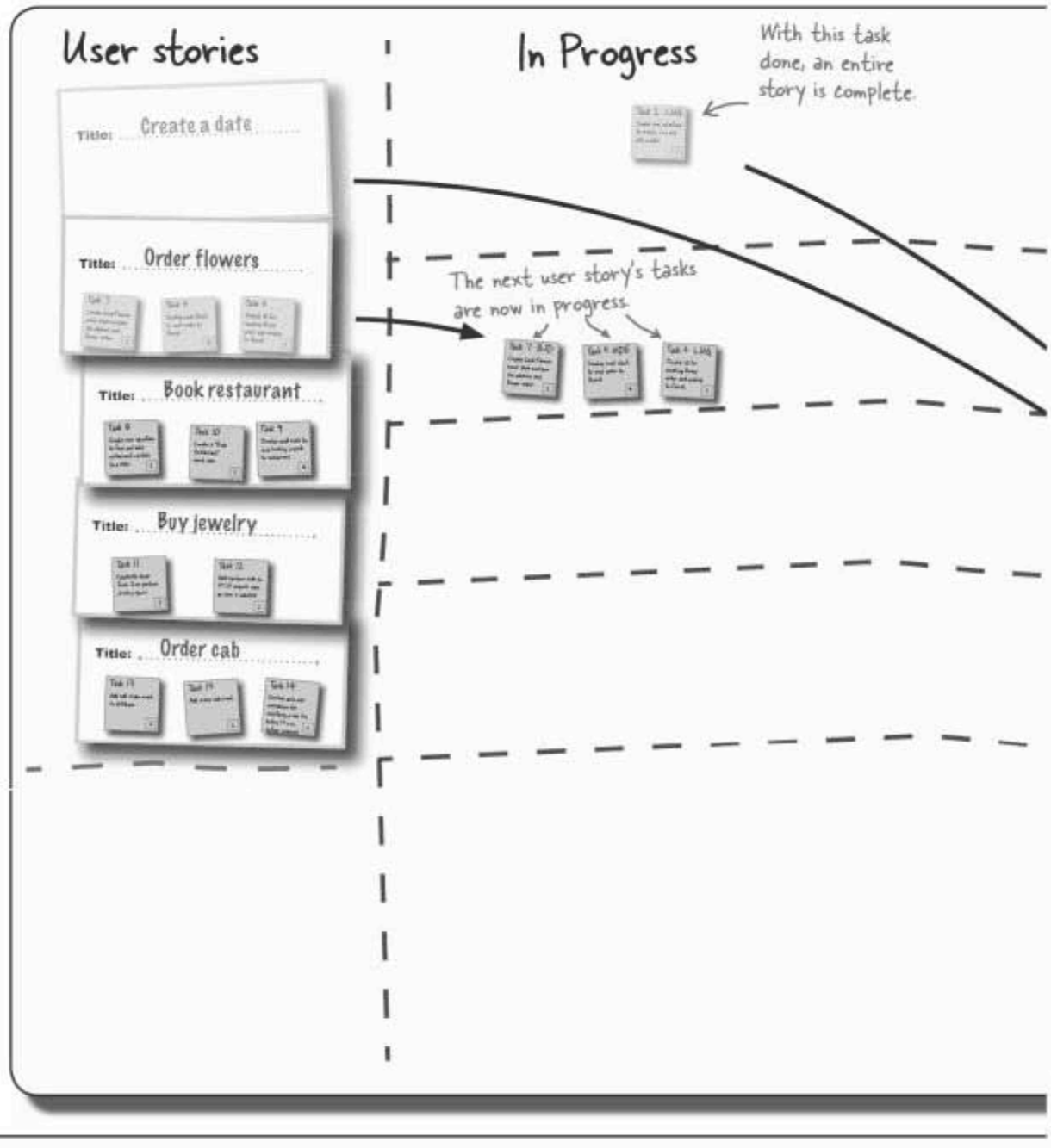
It's the end of Week 1, and you and the team have just finished your standup meeting. It's time to update the project board. Take a look at the board below and write down what you think needs to be changed and updated on the board to get it ready for Week 2.

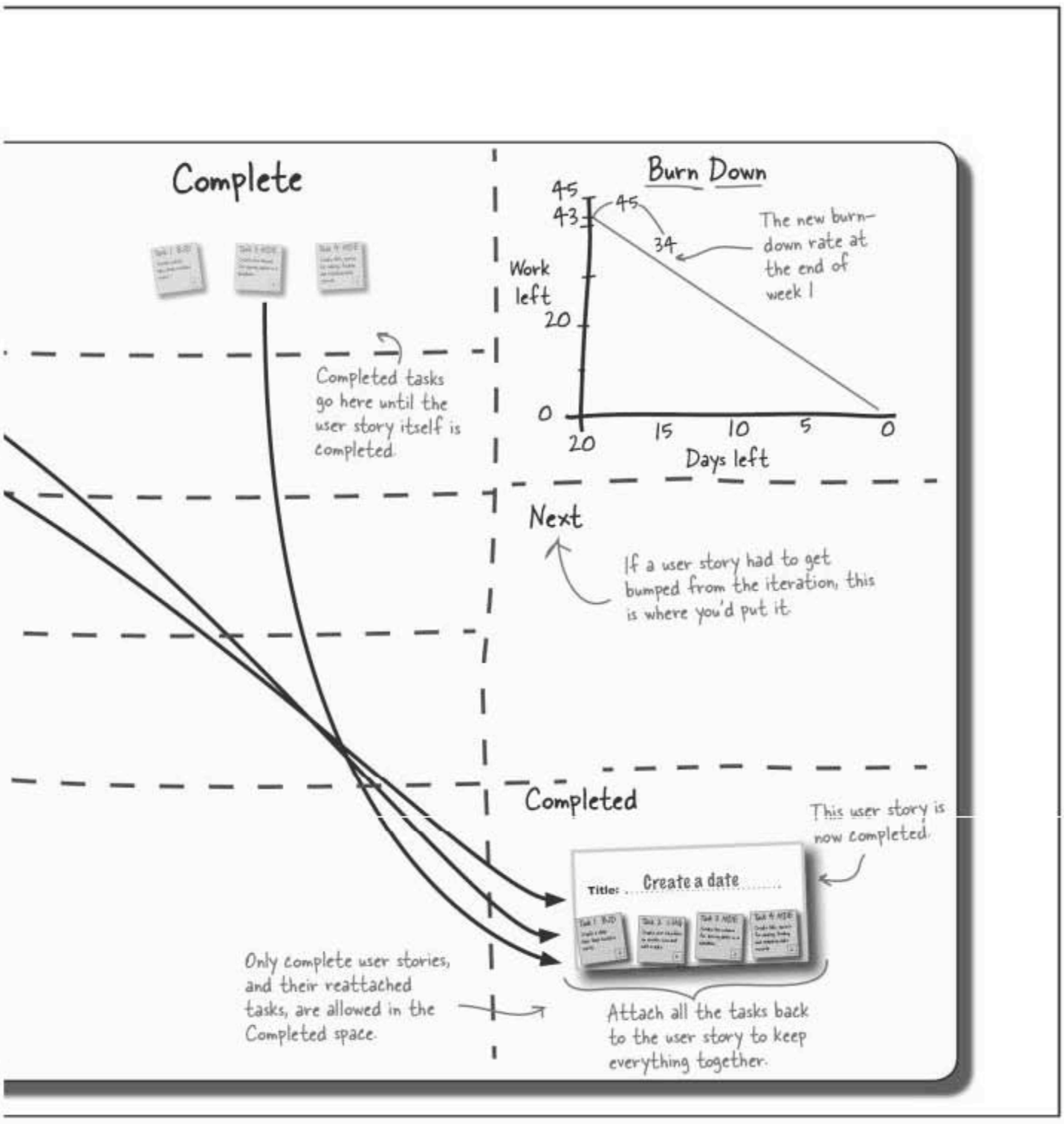



Complete
All these tasks are officially complete, too.
Burn Down
45
43
45
?
Given how much work has been done, what do you think the new burn-down rate should be?
Work left
20
0
20
15
10
5
0
Days left
Next
Completed
Think you need to move anything in here yet?

Long Exercise Solution

You were asked to update the board and write down what you think needs to be changed to get it ready for Week 2.




Complete
Completed tasks go here until the user story itself is completed.
Burn Down
45
43
45
34
The new burn-down rate at the end of week 1
Work left
20
0
20
15
10
5
0
Days left
Next
If a user story had to get bumped from the iteration, this is where you'd put it.
Completed
This user story is now completed.
Title: Create a date
Only complete user stories, and their reattached tasks, are allowed in the Completed space.
Attach all the tasks back to the user story to keep everything together.

Standup meeting: Day 2, Week 2...

One of the In Progres tasks from the board

Laura's acting as the team lead, at least on this iteration.

Hey guys, I've been busy working on my task and I noticed a way of saving us some time and effort by extending our design a little...

Laura: How are you going to do that?

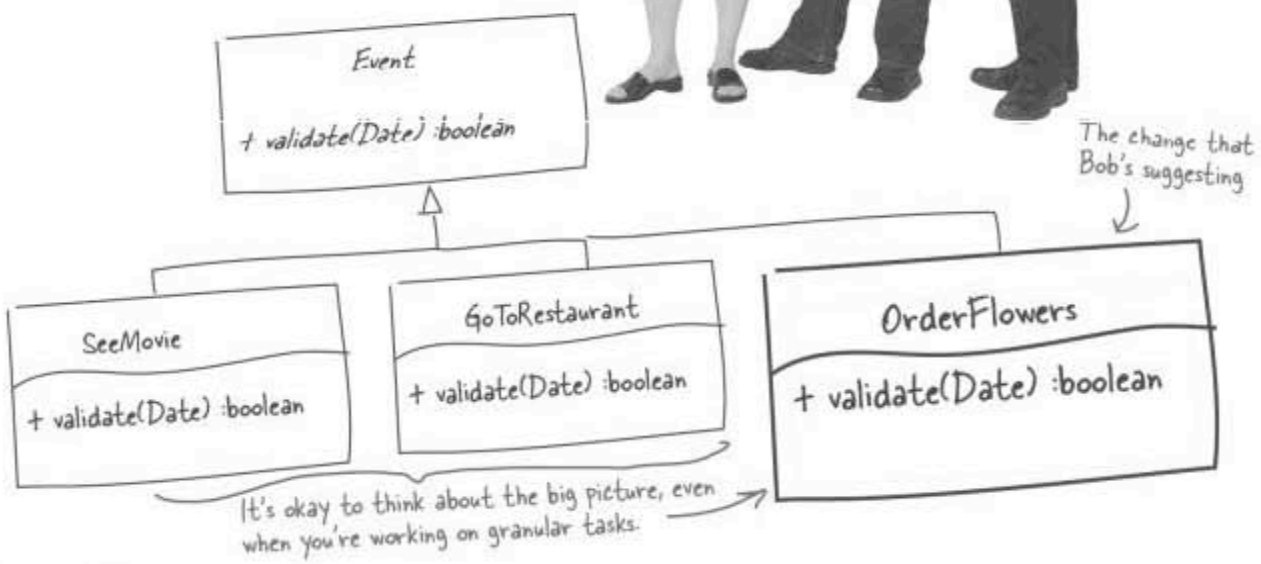
Bob: Well, if you treat someone ordering flowers as just another type of event, then we can add it straight into our current class tree, and that should save us some time in the long run.

Laura: That's sounds good. What do you think, Mark?

Mark: I don't see any problems right now...

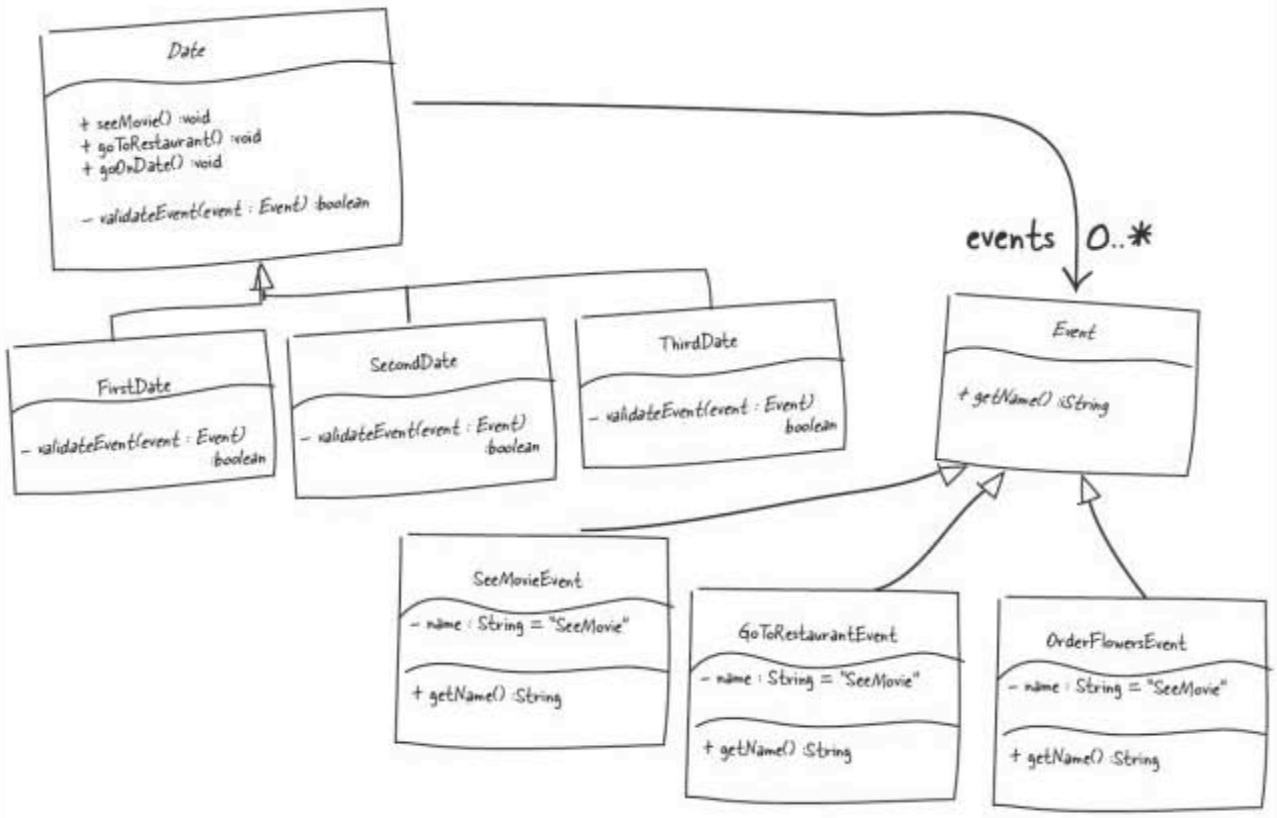
Bob: Apart from it might take an extra day right now to make the changes, but in the long run this should save us some time.

Laura: Mmm. We're still a little behind, but we can probably lose a day on the burn-down rate now if it saves us time later on in the iteration. OK, I'm sold, let's go for it...



Exercise

What refactoring do you think Bob is talking about? Take the class hierarchy below and circle all the things that you think will need to change to accommodate a new OrderFlowers event.

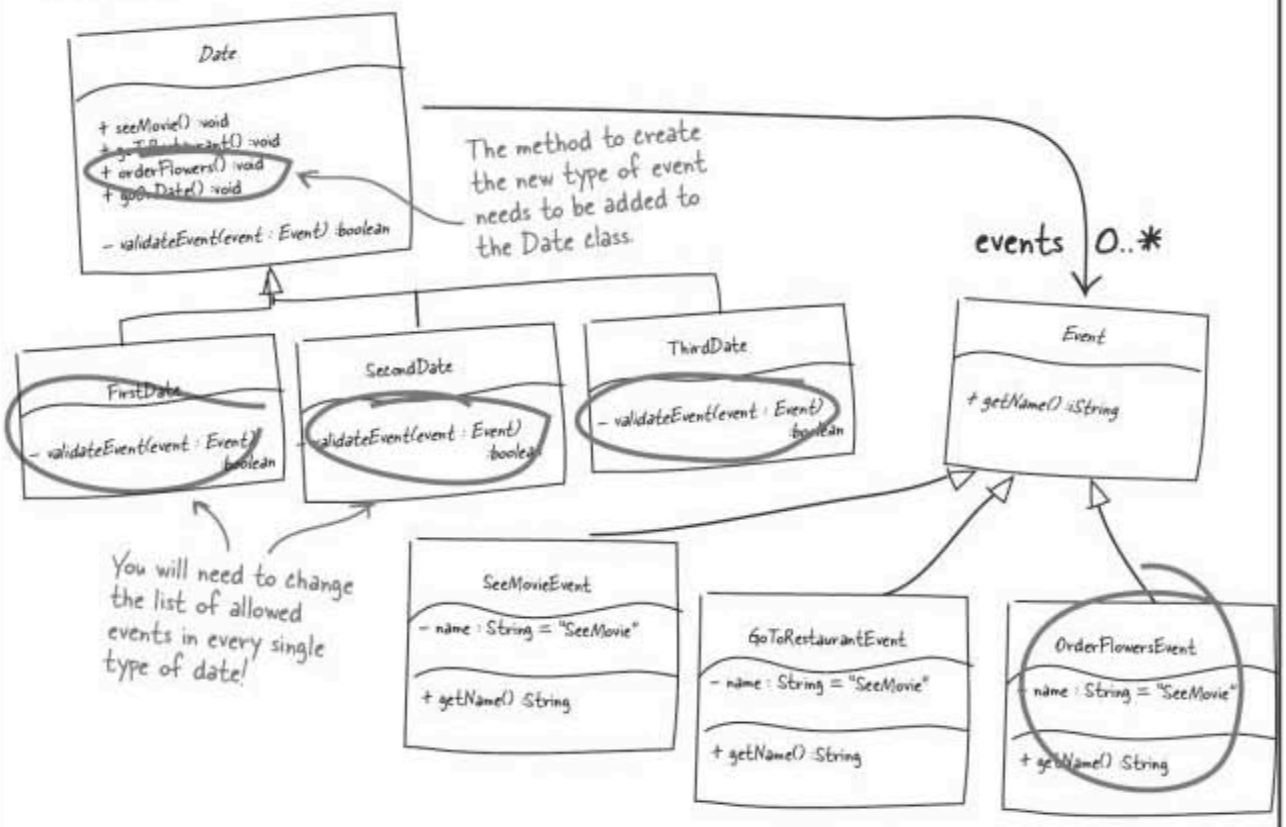


How many classes did you have to touch to make Bob's changes?

Are you happy with this design? Why or why not?

Exercise Solution

You were asked to take the class hierarchy below and circle all the places that you think will need to change to accommodate a new OrderFlowersEvent...



How many classes did you have to touch to make Bob's changes? Five classes were changed or added to add just this one new type of event. First the "OrderFlowersEvent" class needed to be added, and then the method to order flowers on a date needed to be added to the Date class. Finally I had to update each of the different types of date to allow, or reject, the new type of event depending on whether it's allowed on that date or not.

Are you happy with this design? Why or why not?

Five classes being changed seems like a *LOT* when all I'm adding is *ONE* new event. What happens when I have to add, say, a dozen new types of event; is it always going to involve this much work?

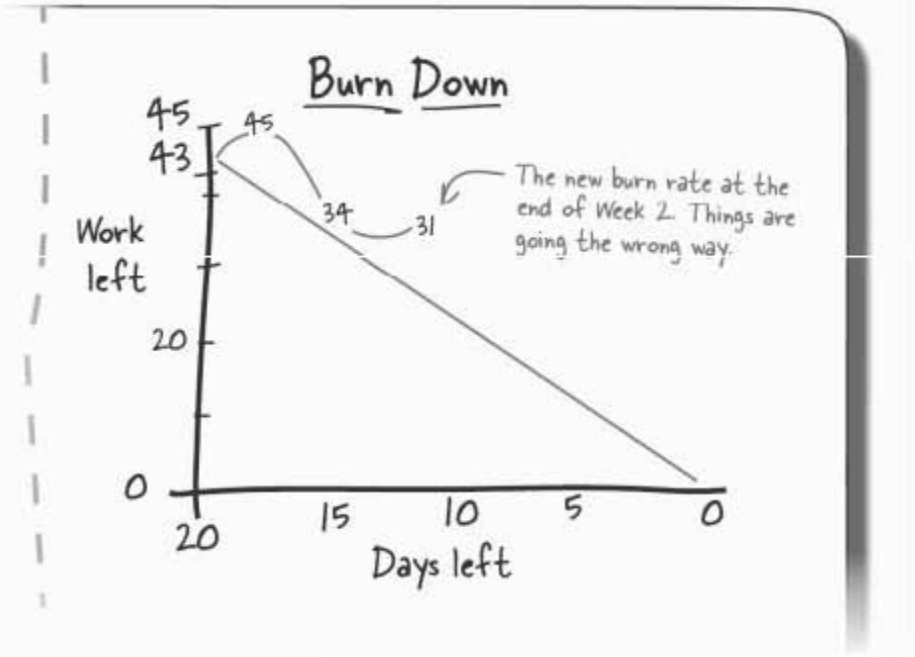
Laura: Hey, isn't "Buy jewelry" coming down the line? That works as just another event, too, right?

Bob: Yeah, but we'll need to add some time to make those changes to all the classes again.

Mark: Can't we come up with a more flexible design, so we can avoid this pain and effort each time we add a new event?

Bob: That's exactly what I was thinking.

Laura: But that will take even more time, right? I guess we're invested, though, huh? This will save us time later, I hope...



We interrupt this chapter...

You're already getting behind on your burn-down rate and then the inevitable happens: the customer calls with a last-minute request...

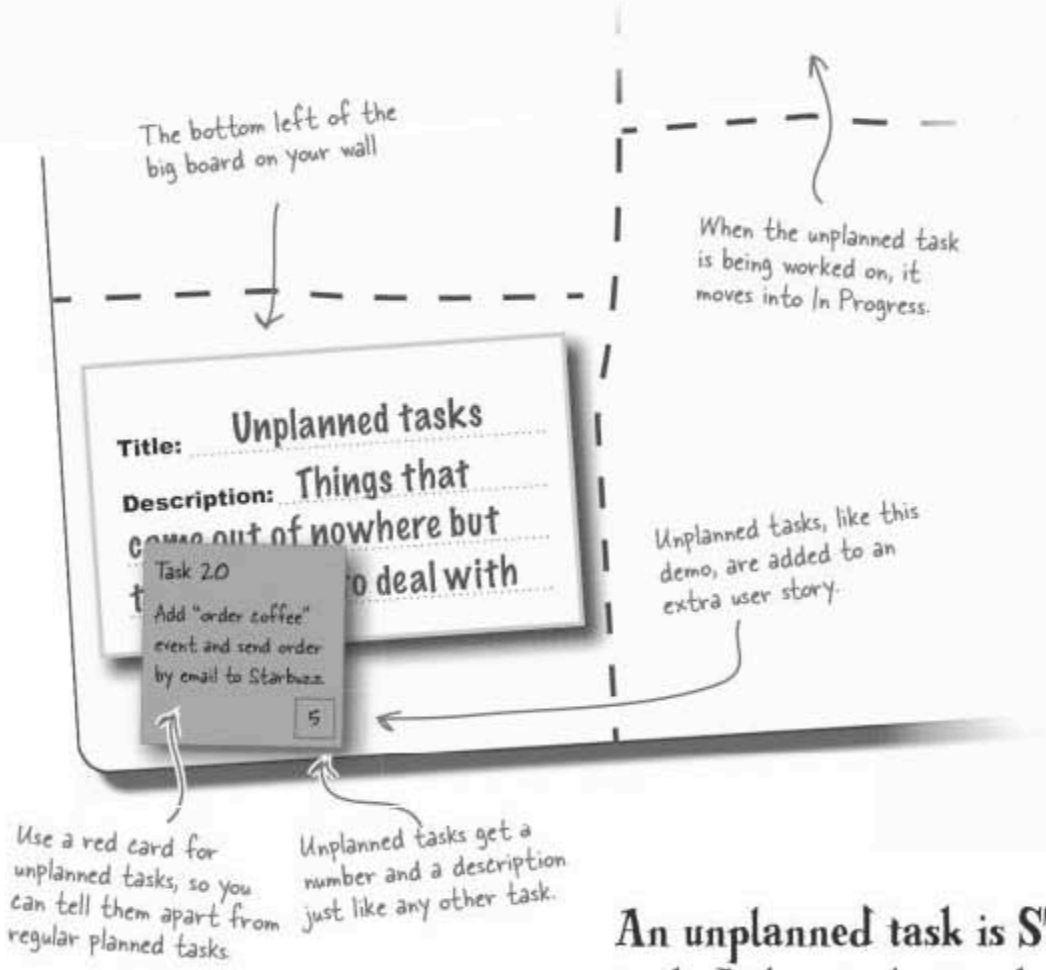
Hey! The CEO of Starbuzz just called, and he wants to see a demo of ordering coffee as part of a date. Can you show me that tomorrow?

Your customer, iSwoon's CEO

You have to track unplanned tasks

So far, your board has kept track of everything going on in your project. But what happens when somthing unplanned comes up? You have to track it, just like anything else. It affects your burn-down rate, the work you're doing on user stories, and more...

Let's take a look at a part of the board we haven't used yet:



An unplanned task is STILL a task. It has to be tracked, put in progress, completed, and included in the burn-down rate just like EVERY OTHER TASK you have.

Wait a sec! You're saying we have to do the demo? What if it blows our deadlines?

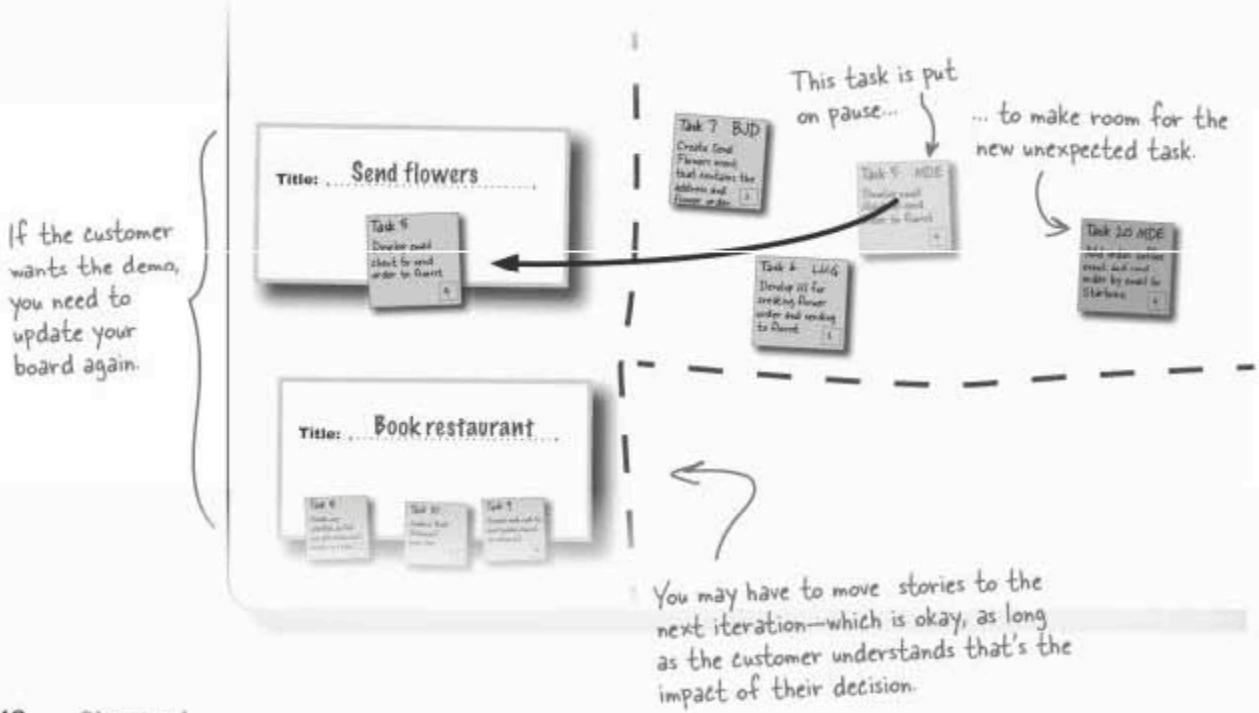
Yes, you've heard it before, but talking to the customer is the answer to most scheduling- and deadline-related problems.

Talk to the customer

You've been hit by the unexpected, but that's part of software development. You can't do everything, but you also can't make the choice about what takes priority. Remember, **the customer sets priorities**, not you.

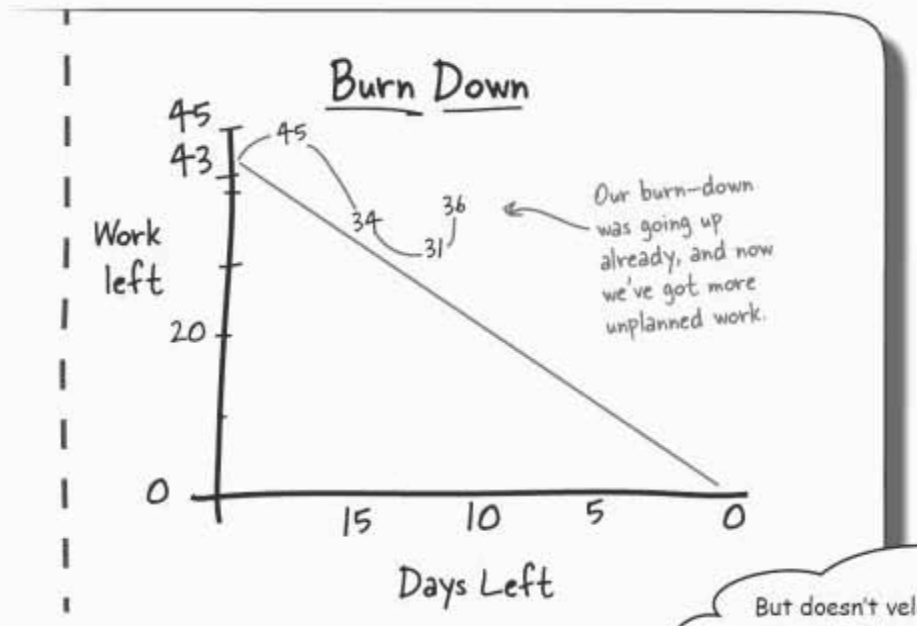
You need to deal with new tasks like customer demos, and the best way to do this is to ask the customer what takes priority. Give the customer a chance to make a considered decision by estimating the amount of work that the new task requires and explaining how that will affect the current schedule. Ultimately, the customer rules, so as long as they have all the information needed to make a choice, then you need to be prepared to go with their decision by reshuffling your existing tasks and user stories to make room for the surprise work.

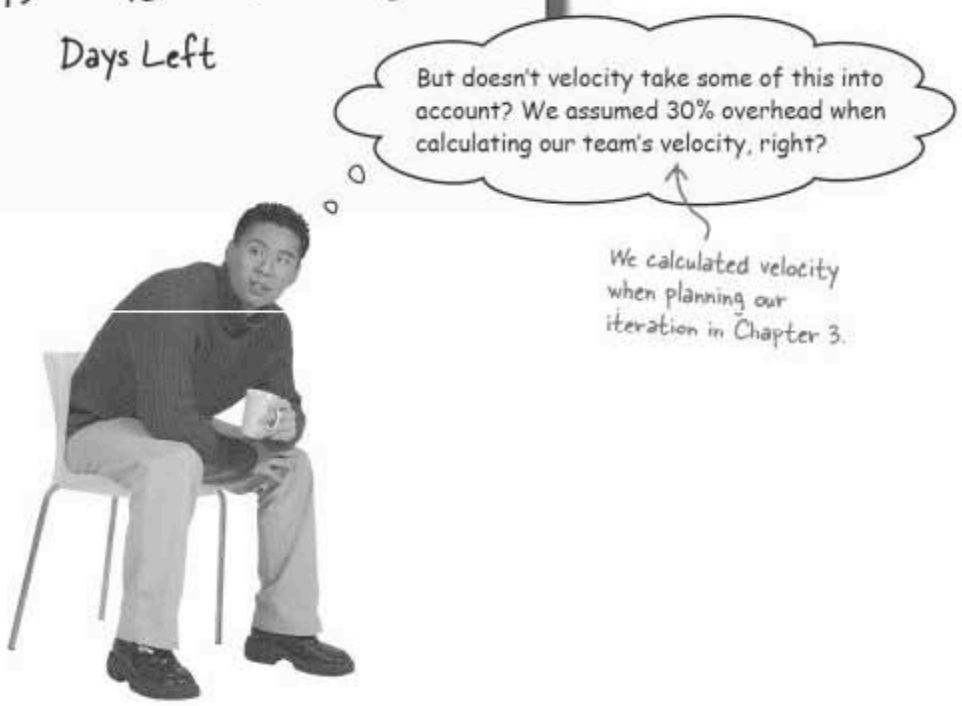
Ultimately you need to keep your customer in the picture as to what is in and what is out. Adding new unplanned work is not the end of the world, but your customer needs to understand that the work has an impact, and then they can choose what that impact is.



Unexpected tasks raise your burn-down rate

Unexpected task mean extra work. If the unexpected tasks can't be pushed into another iteration, then they need to be factored into your board. All of this means that your burn-down rate is affected, and not in a good way...





Velocity helps, but...

You've got more work thanks to some unexpected requirements from your customer, but didn't you factor this in when you calculated your team's velocity? Unfortunately, velocity is there to help you gauge how fast your team performs, but it's not there to handle unplanned tasks.

We originally calculated velocity as...

Remember this equation from Chapter 3?

$$3 \times 20 \times 0.7 = 42$$

- 3: The number of people in your team
- 0.7: Your team's first pass velocity, which is actually a guess at this point
- 42: The amount of work in days that your team can handle in one iteration

So we have this much "float"...

$$3 \times 20 - 42 = 18$$

These are the possible days we could have, if everyone worked at 100% velocity...

... but it may not be enough!

Float—the "extra" days in your schedule—disappear quickly.

An employee's car breaks down, someone has to go to the dentist, your daily standup meetings...those "extra" days disappear quickly. And remember, ***float is in work time, not actual time***. So if your company gives an extra Friday off for great work, that's *three* days of float lost because you are losing *three* developers for the whole day.

So when unplanned tasks come up, you may be able to absorb some of the extra time, but velocity won't take care of all of it.

there are no Dumb Questions

Q: You said to add unplanned tasks as red sticky notes. Do I have to use colored sticky notes? And why red?

A: We picked red because regular tasks are usually on regular yellow sticky notes, and because red stands out as a warning color. The idea is to quickly see what's part of your planned stories (the normal stickies), and what's unplanned (red). And red is a good "alert" color, since most unplanned tasks are high-priority (like that customer demo that came out of nowhere).

It's also important to know at the end of an iteration what you worked on.The red tasks make it easy to see what you dealt with that wasn't planned, so when you're recalculating velocity and seeing how good your estimates were, you know what was planned and what wasn't.

Q: So later on we're going to recalculate velocity?

A: Absolutely. Your team's velocity will be recalculated at the beginning of every single iteration. That way, you can get a realistic estimate of *your* team's productivity. 0.7 is just a good conservative place to start when you don't have any previous iterations to work from.

Q: So velocity is all about how me and my team performed in the last iteration?

A: Bingo. Velocity is a measure of how fast ***you*** and ***your team*** are working. The only way you can *reliably* come up with a figure for that is by looking at how well you performed in previous iterations.

Q: I really don't think 0.7 captures my team's velocity. Would it be OK to pick a faster or slower figure to start out with? Say 0.65, or 0.8?

A: You can pick a different starting velocity, but you have to stand by what you pick. If you know your team already at the beginning of a project, then it's perfectly alright to pick a velocity that matches your team's performance on other projects, although you should still factor in a slightly slower velocity at the beginning of any project. It always takes a little extra time to get your heads around what needs to be developed on a new project.

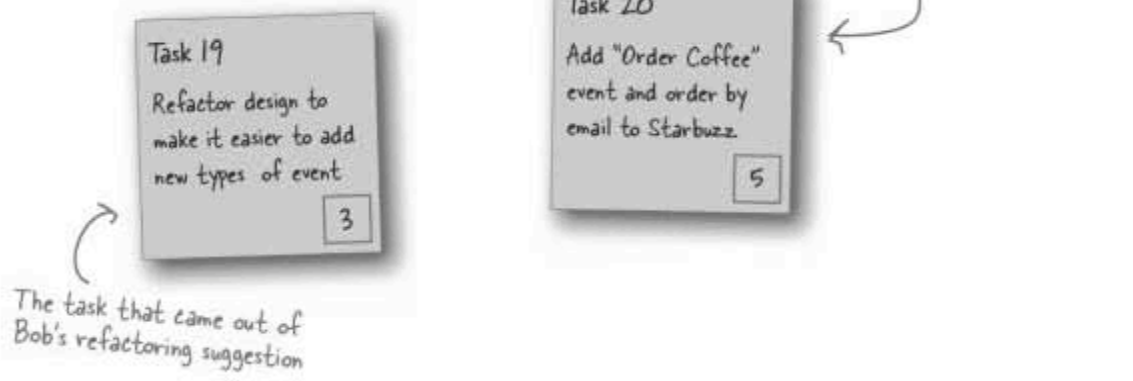
Remember, velocity is about how fast you and your team can comfortably work, for real. So you're aiming for a velocity that you believe in, and it's better to be slightly on the conservative side at the beginning of a new project, and then to refine that figure with hard data before each subsequent iteration.

Velocity is NOT a substitute for good estimation; it's a way of factoring in the real-world performance of you and your team.

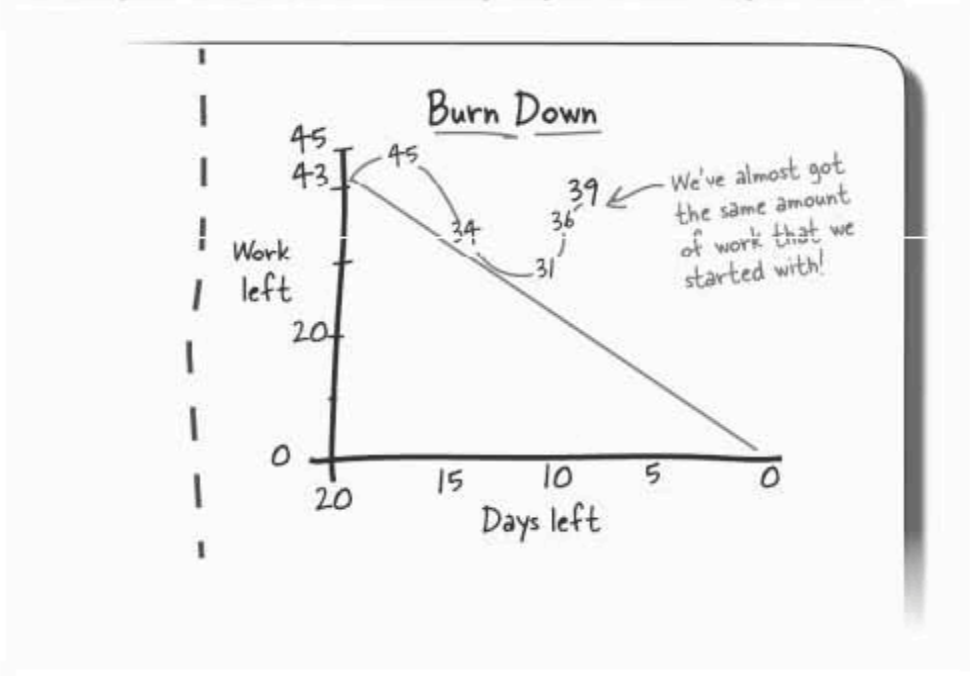
We have a lot to do...

You're in a tough spot. Doing some refactoring work is going to cost you time now, but the hope is that it will save you time in the long run. In addition you have the new demo that you need to prepare for the iSwoon CEO....

You've got more work to do...



...and your burn-down rate is going in the wrong direction.



...but we know EXACTLY where we stand

The customer knows where you are

At every step you've kept the customer involved so they know exactly what work they've added, and you can show them exactly what the changes will impact.

YOU know where you are

You and your development team are also on exactly the same page thanks to your board and the burn-down rate. This means that although things look a bit bleak, at least no one is burying their heads in the sand. The challenges are right there on your wall.

You know there are challenges, NOW.

Because you're monitoring your project using your board you know right now that there are challenges ahead if you're going to keep things on track. Compare this with the Big Bang "See you later, I'll deliver something in 3 months" approach from Chapter 1.

With the Big Bang approach, you didn't know you were in trouble until day 30, or even day 90! With your board and your burn-down rate you know immediately what you're facing, and that gives you the edge to make the calls to keep your development heading towards success.

Sometimes you'll hear this referred to as the waterfall approach

Successful software development is about knowing where you are. With an understanding of your progress and challenges, you can keep your customer in the loop, and deliver software when it's needed.

All is far from lost! We'll tackle all these problems in Chapter 5, when we dig deeper into good class and application design, and handle the customer demo.

Velocity Exposed

This week's interview:
Keeping pace with Velocity

Head First: Welcome, Velocity, glad you could make time in your busy day to come talk with us.

Velocity: My pleasure, it's nice to be here.

Head First: So some would say that you have the potential to save a project that's in crisis, due perhaps to surprise changes or any of the other pieces of extra work that can hit a plan. What would you say to those people?

Velocity: Well, I'm really no superhero to be honest. I'm more of a safety net and confidence kinda guy.

Head First: What do you mean by "confidence"?

Velocity: I'm most useful when you're trying to come up with realistic plans, but not for dealing with the unexpected.

Head First: So you're really only useful at the beginning of a project?

Velocity: Well, I'm useful then, but at that point I'm usually just set to my default value of 0.7. My role gets much more interesting as you move from Iiteration 1 to Iteration 2 and onwards.

Head First: And what do you offer for each iteration, confidence?

Velocity: Absolutely. As you move from one iteration to the next you can recalculate me to make sure that you can successfully complete the work you need to.

Head First: So you're more like a retrospective player?

Velocity: Exactly! I tell you how fast you were performing in the last iteration. You can then take that value and come up with a chunk of work in the next iteration that you can be much more confident that you can accomplish.

Head First: But when the unexpected comes along...

Velocity: Well, I can't really help too much with that, except that if you can increase your team's velocity, you might be able to fit in some more work. But that's a risky approach...

Head First: Risky because you really represent how fast your team works?

Velocity: That's exactly my point! I represent how fast your team works. If I say that you and your team, that's 3 developers total, can get 40 days of work done in an iteration, that's 20 work days long, that doesn't mean that there's 20 days there that you could possibly use if you just worked harder. Your team is always working as hard as they can, and I'm a measure of that. The danger is when people start using me as a pool of possible extra days of work...

Head First: So, if you could sum yourself up in one sentence, what would it be?

Velocity: I'm the guy that tells you how fast your team worked in the last iteration. I'm a measure of how you perform in reality, based on how you performed in the past, and I'm here to help you plan your iterations realistically.

Head First: Well, that's actually two sentences, but we'll let you get away with that. Thanks for making the time to come here today, Velocity.

Velocity: It's been a pleasure, nice to get some of these things off of my chest.