

A Brain-Friendly Guide

# Head First Software Development



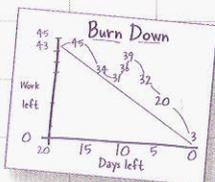
Learn the real user story of how Mary satisfied her customers



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



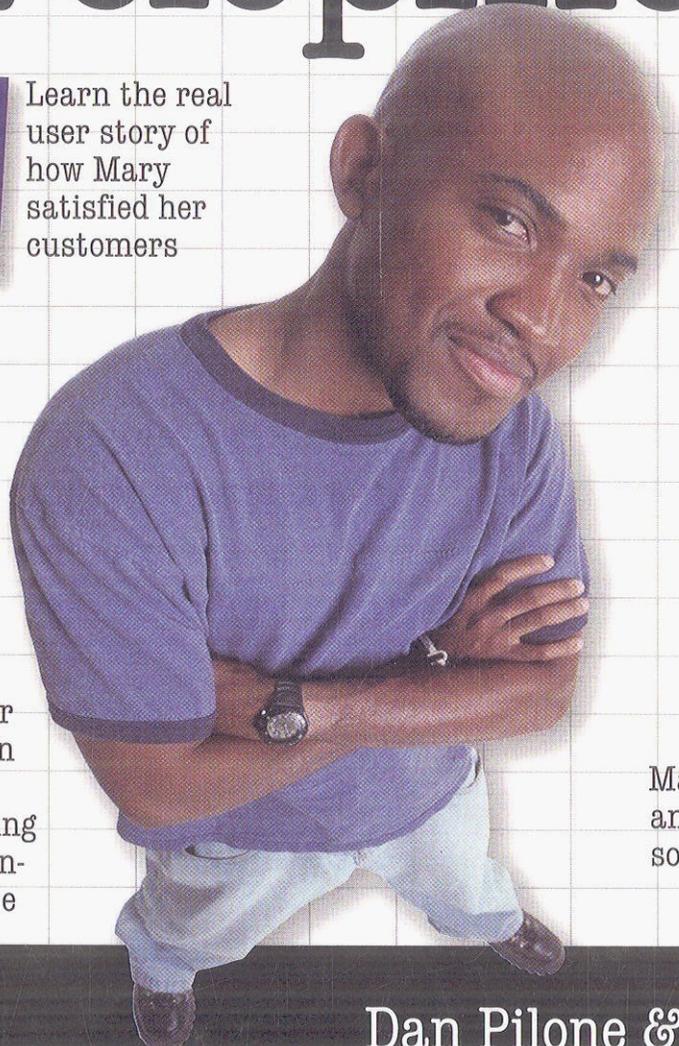
Use test-driven development to avoid unsightly software disasters



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

# Table of Contents (Summary)

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

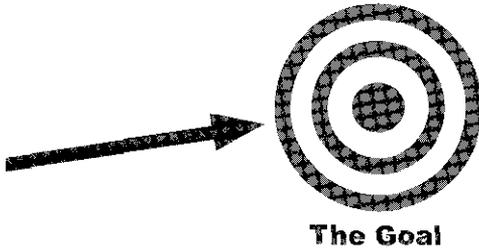
# great software development

## Pleasing your customer

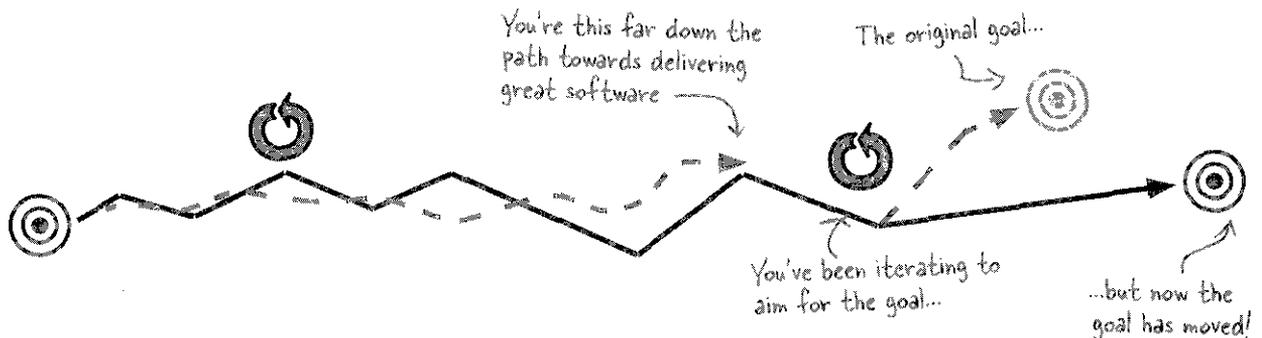
# 1

### 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.



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



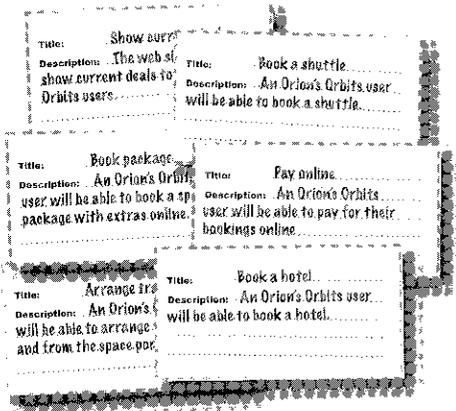
# gathering requirements

## Knowing what the customer wants

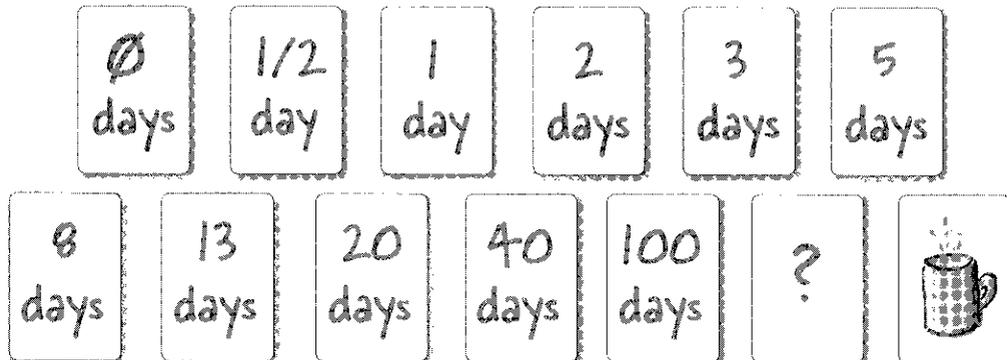
# 2

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	



# project planning

## Planning for success

# 3

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 <del>planned</del> customers	98
The Big Board on your wall	100
How to ruin your team's lives	103

Here's what a programmer **SAYS**...



Sure, no problem, I can crank through that in 2 days.

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



I'll grab a Monster on the way home, program 'til 3 AM, take a Halo break, then work through the morning. Sleep a few hours, get the guys over to hack with me, and finish at midnight. As long as nothing goes wrong... and Mom doesn't need me to pick up dinner.

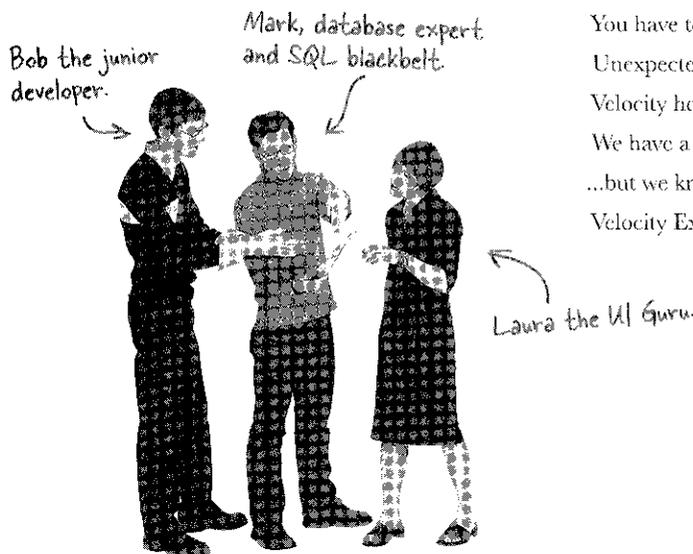
## Getting to the real work

# 4

**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...

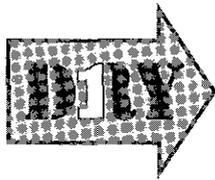


# good-enough design

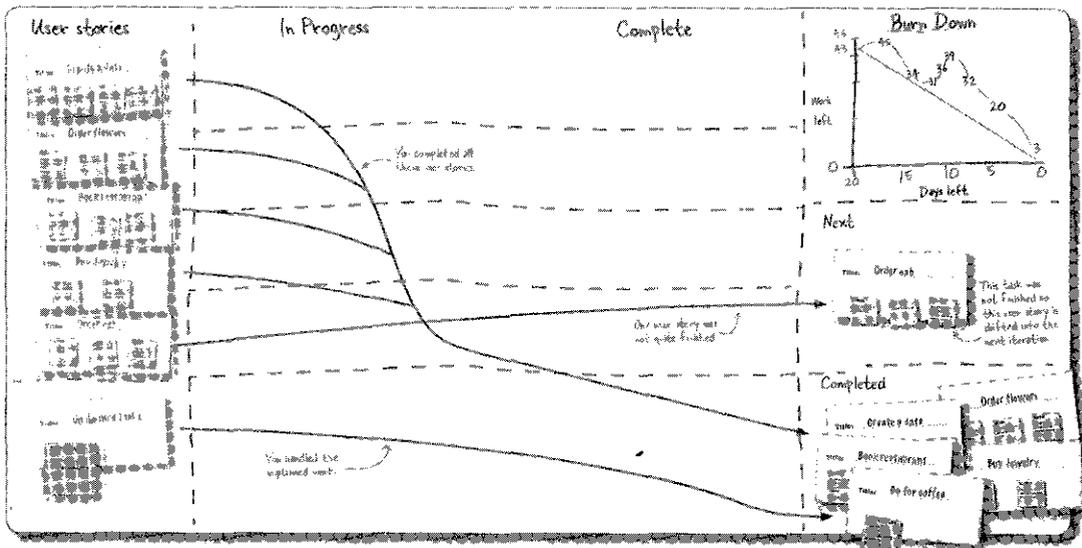
# 5

## 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 responsibilities in your design	156
Going from multiple responsibilities 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

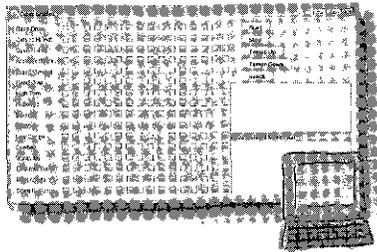


## Defensive development

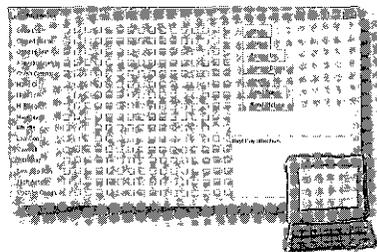
# 6

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

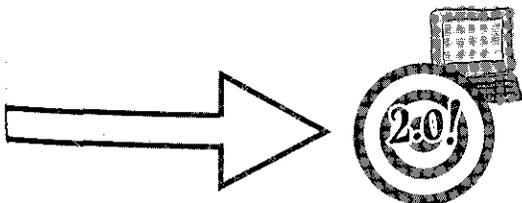
Writing great software isn't easy... especially when you've got to make sure your code works, and **make sure it keeps working**. All it takes is one typo, one bad decision from a co-worker, one crashed hard drive, and suddenly all your work goes down the drain. But with **version control**, you can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

# 6 1/2

## building your code

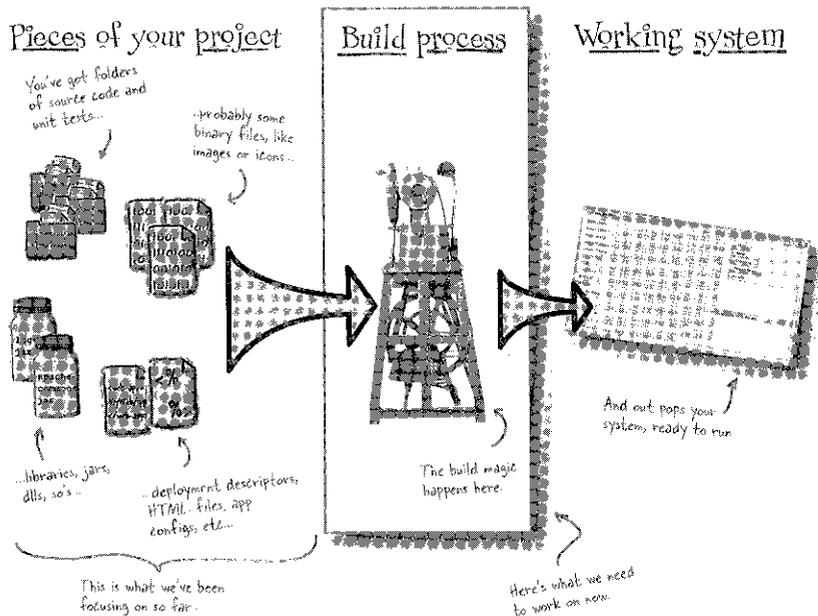
### Insert tab a into slot b...

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



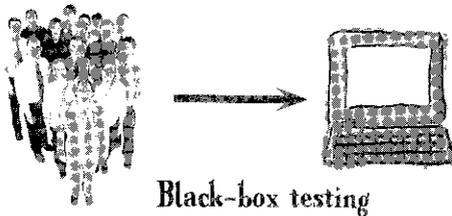
# 7

## Things fall apart

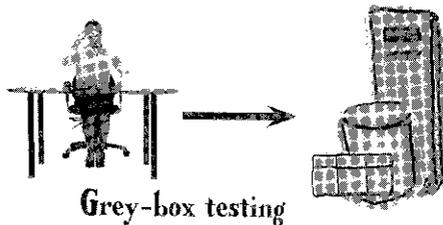
### Sometimes even the best developer breaks the build.

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



Black-box testing



Grey-box testing



White-box testing

# 8

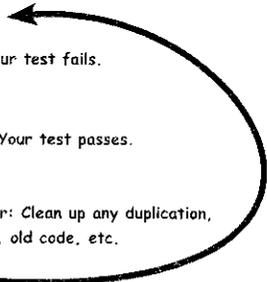
## Holding your code accountable

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 for multiple 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



- 1 Red: Your test fails.
- 2 Green: Your test passes.
- 3 Refactor: Clean up any duplication, ugliness, old code, etc.



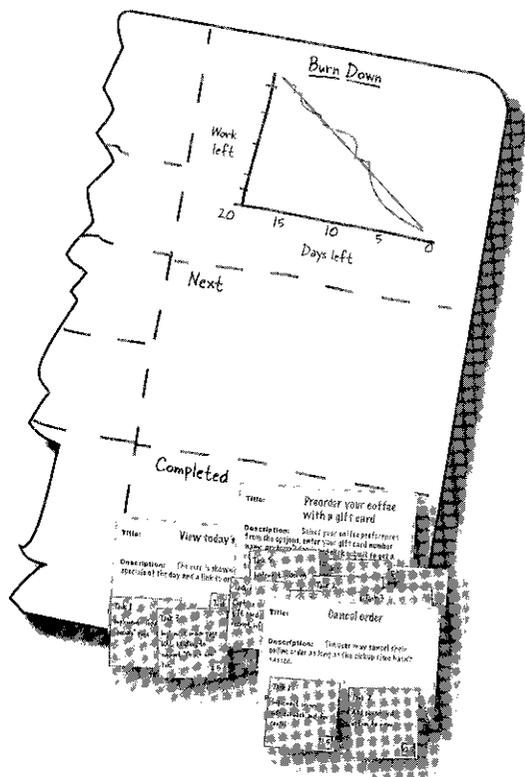
## ending an iteration

### It's all coming together...

# 9

**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 <b>MUST</b> be done...	324
...but <b>WHO</b> does system testing?	325
System testing depends on a complete system to test	326
Good system testing requires <b>TWO</b> 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 <b>COULD</b> do...	338
Time for the iteration review	342
Some iteration review questions	343
A <b>GENERAL</b> priority list for getting <b>EXTRA</b> things done...	344
Tools for your Software Development Toolbox	346



the next iteration

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

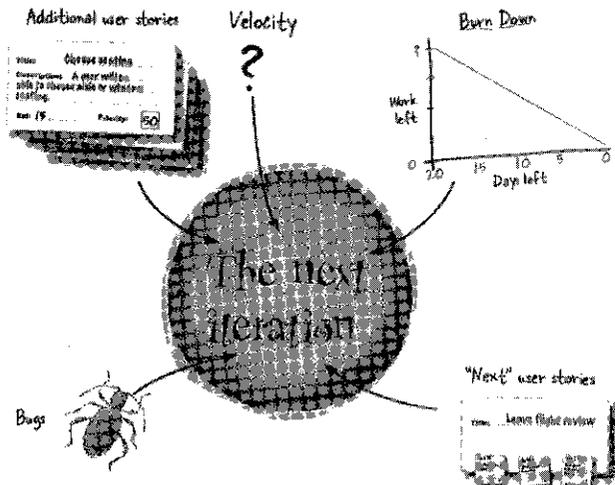
# 10

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 expectations 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 S'TILL about the customer	360
Someone else's software is S'TILL 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

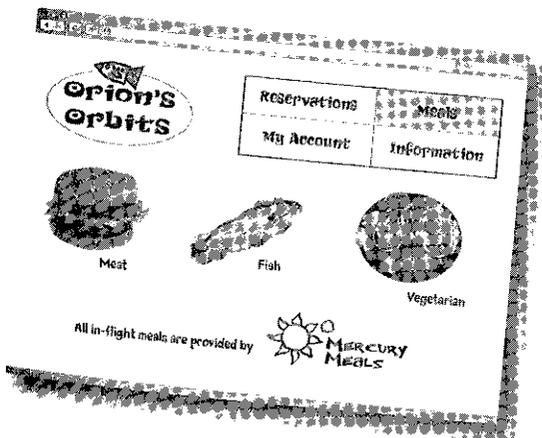


## 11

**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

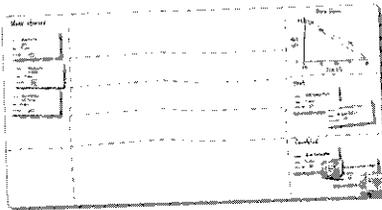


the real world

## Having a process in life

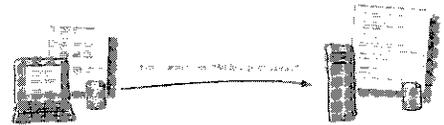
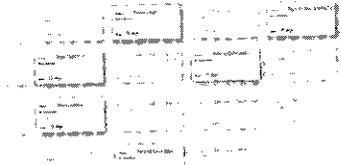
# 12

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**.

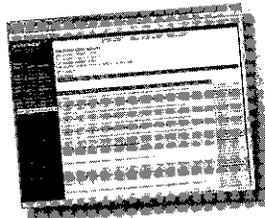


**Story and Burn Down board**

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



**Configuration Management (CM)**

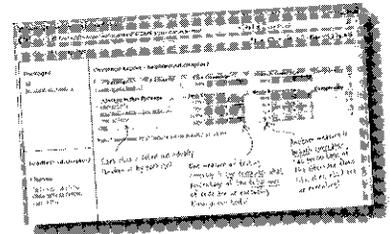


**Continuous Integration (CI)**

### User Stories



- Red - Your Test Fails**  
If a test fails, you are in a red state. This means you have a problem. You need to investigate the failure and fix it. This is the most common state for a test.
- Green - Your Test Passes**  
If a test passes, you are in a green state. This means you have a working test. This is the most common state for a test.
- Reference - Code is not deployable**  
If the code is not deployable, you are in a reference state. This means you have a problem with the code that prevents it from being deployed. This is the most common state for a test.



**Test Coverage**

### Test Driven Development (TDD)

# appendix 1: leftovers

## The top 5 things (we didn't cover)

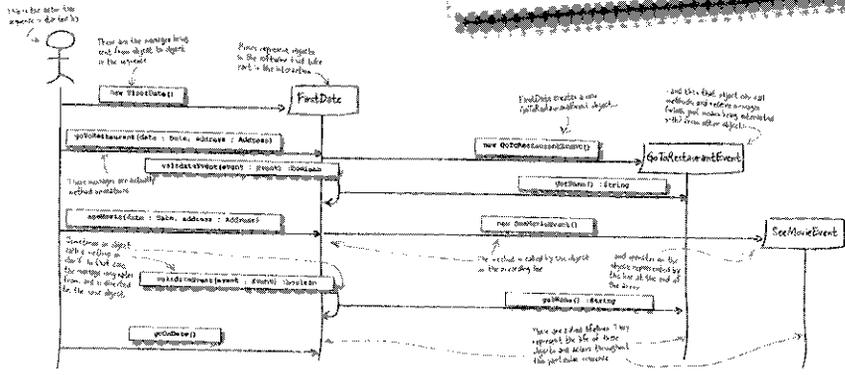
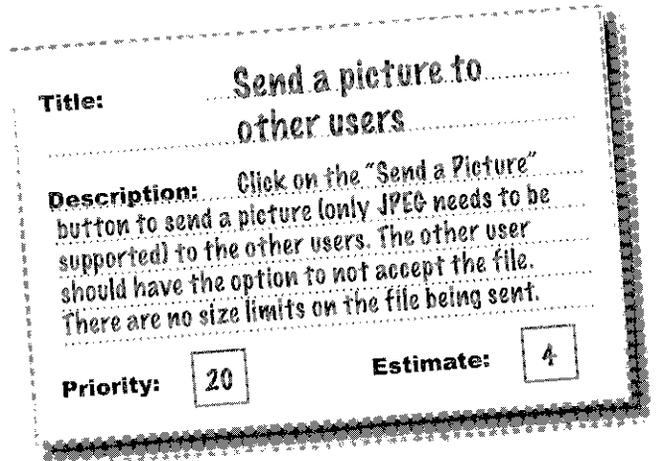
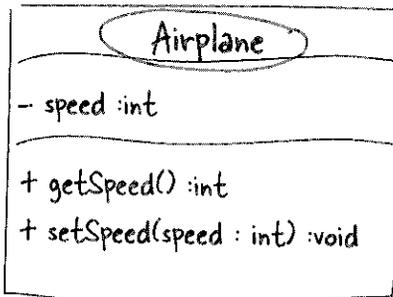


Ever feel like something's missing? We know what you mean...

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

