CARL BROWN

App Accomplished

Strategies for App Development Success

Foreword by Kyle Richter, CEO, MartianCraft

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

App Accomplished

Strategies for App Development Success

Carl Brown

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

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

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

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2014938789

Copyright © 2015 Carl Brown

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-96178-5
ISBN-10: 0-321-96178-1

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing: July 2014

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Development Editor
Sheri Cain

Managing Editor
Kristy Hart

Senior Project Editor
Betsy Gratner

Copy Editor
Kitty Wilson

Senior Indexer
Cheryl Lenser

Proofreader
Kathy Ruiz

Technical Reviewers
Mark Kolb
Wes Miller
Bob Wesson

Editorial Assistant
Olivia Basegio

Cover Designer
Alan Clements

Compositor
Nonie Ratcliff

This book is dedicated to the app creators, entrepreneurs, CEOs, and managers who have an idea for a mobile app and bother to find the money, take the time, and expend the effort to get the app built. You have fueled an impressive ecosystem the past few years, and I am proud to be a member of it.
I hope this book will help you to get better apps for your money and effort.

This page intentionally left blank

Table of Contents

Foreword .... **xi**

Preface .... **xiii**

1 What Could Possibly Go Wrong? **1**

App Projects Are Not Small and Easy 1

Apps Are Not Easy to Program 5

Poor Skill Set Fit 8

If You Get a Good Developer, You Still Have to Worry 10

The Idea Is Not More Important Than the Execution 12

Unwillingness to Delegate: Micromanaging 15

Bikeshedding 16

Poorly Defined Requirements 16

Out-of-Date Requirements Documentation 18

Constantly Changing Requirements 20

Leaving the Worst for Last 20

Cost Overruns 24

That Last 10% 26

The Whack-a-Mole Problem 27

Poor Communication 29

Abdication of the Management Process 31

Wrapping Up 31

2 The App Development Life Cycle **33**

The Design Phase 34

The Development Phase 39

The Testing Phase 45

Repeating the Cycle as Needed 49

Wrapping Up 52

3 Prototyping and Wireframing Your App 53
Focus on the Core Experience 54
Wireframe the App 58
Build an Interactive Prototype 76
Prototyping Tips and Tricks 91
Wrapping Up 95
4 Determining Your App’s Components 97
Dealing with Devices 97
Native, Web, and Hybrid Apps 104
Dealing with Third-Party Frameworks 111
Dealing with Analytics 119
Dealing with Video and Audio 120
Dealing with Peripherals 121
Dealing with Accessibility 122
Dealing with Custom or Complex Animations 122
Dealing with Conditional Formatting 123
Dealing with Localization 124
Dealing with User Preferences 125
Dealing with Data Storage 125
Dealing with Servers 131
Dealing with Syncing 133
Dealing with Push Notifications 134
Dealing with Background Tasks 134
Wrapping Up 135
5 Finding the Right Tools 137
Selecting Tools for Your Project Size 138
Source Control 138
Bug Tracking 144
Project and Schedule Tracking 148
The Development Environment 154
Continuous Integration 157
Beta Testing Distribution 159
Crash Reporting 160
End-User Feedback 161
Wrapping Up 162

6 Skill Gap Analysis **163**
Programming 163
Testing and Quality Assurance 168
Server Support and Troubleshooting 168
User Experience Design 169
Graphic Design 173
Sound Design and Music 176
Copywriting 178
Marketing 179
About Games 181
Wrapping Up 183

7 Finding a Developer **185**
Template App Sites 185
App Developer Matchmaker Sites 189
Local Versus Remote Developers 191
Creative Agencies 194
App Development Companies 196
Independent Developers 199
Grow Your Own Developer (Maybe Even You) 203
Wrapping Up 204

8 Interviewing and Selecting a Developer **207**
Nondisclosure Agreements 208
Setting Up an Interview 208
Previous Work 210
Gap Analysis 212
Contingency Plans 213
Estimating and Planning 214
Working Relationship 217
Wrapping Up 224

9 Managing to Milestones **227**
Never Agree to "30% Down, and I'll Talk to You in Three Months" 227
Minimizing Risk with Frequent Milestones 228
How I Learned to Stop Grumbling and Love Milestones 229
Milestones Are Not Sprints 230

Organization, Sequencing, and Focus 232
Let Conway's Law Be Your Guide 235
Scheduling Software: Strongly Suggested 237
Remember That Estimates Are Only Estimates 239
Renovation Versus New Construction 243
Estimates and Entomology 245
Plan Reevaluation and Project Feedback Loops 246
Wrapping Up 246

10 Understanding What You're Getting 249

Living Within Your Means 250
The Ticking Clock 251
Justifying Effort for Your Project Size 253
Get the Code, Even if There's Nothing to See in the UI 253
Comments in Source Control 254
Comments in Code 256
Build and Run the App Yourself 258
Third-Party Libraries 260
Source Code Project Organization 261
Automated Test Coverage 262
Detecting Plagiarism 262
Compiler Warnings 264
Duplicated Code 264
Commented Out Code 265
Magic Numbers 265
Huge Combinatorial Complexity 266
Useless, Ambiguous, or Confusing Naming 266
The "UI Thread" or "Main Thread" 267
Wrapping Up 267

11 Pulling the Plug Early 269

So You Missed a Milestone 270
Stop the Presses! Figure Out Where You Are 270
Discussing Failure 271
Milestone Hit but Bugs Abound 272
If Your Developer Is Proactive 274
If Your Developer Isn't Honest 275

If It Might Have Been Your Fault ... 275
Evaluating the Recovery Plan ... 277
How Far Gone Are You? ... 282
Trying to Salvage a Project ... 283
Fair Compensation ... 284
Transitioning to a New Developer ... 284
Wrapping Up ... 285

12 Communicating Using Bugs ... 287

Vocabulary ... 287
Bug Trackers as Communication Tools ... 288
One Bug per Bug Report, Please ... 290
Anatomy of a Bug Report ... 291
Feature Request Versus Bug Fix ... 292
Placeholder Issues ... 294
Bug Trackers as Business Continuity ... 295
Bug Trackers Versus Code Comments ... 295
Writing Useful Bug Reports ... 296
Attaching Files to Bugs ... 298
Data-Specific Bugs ... 299
Reproduction: There's the Rub ... 299
Bug States ... 300
Reopening Bugs Versus Creating New Ones ... 301
Splitting Bugs ... 303
Two Bugs, One Cause ... 303
Saving for Posterity ... 304
Wrapping Up ... 304

13 Testing ... 305

Types of Testing ... 305
Failures of Imagination ... 306
Your Testing Schedule ... 308
Approaching Deadlines ... 311
Your Testing Team ... 311
Getting and Incorporating Feedback ... 319
Wrapping Up ... 327

14 **Submission and Beyond** **329**
Getting Your Marketing Material Together 330
Reviewer Instructions 331
Last-Minute Plea for Sanity 333
Pushing the Button 334
Dealing with Rejection 335
Resubmission 340
Launch 340
Getting Feedback 340
The Next Release 342
The Red Queen’s Race 343
Wrapping Up 343
Index **345**

Foreword

Mobile apps have become big news and even bigger business in recent years. There are shockingly few people who haven't had the "next great app" idea. With frequent news of overnight millionaires and multibillion dollar buyouts, who can blame them for wanting to explore app development? What prevents most of these people from making their mark in the emerging app market is failing to understand the process of creating the app itself. A great idea with poor execution isn't worth anything, but even the most modest idea with fantastic execution can be a valuable product.

Despite all the popularity and buzz surrounding mobile applications, there hasn't been a publication focused on how to successfully plan and build a mobile app—until now. *App Accomplished* fills the same demand that pick-and-shovel salesmen did for nineteenth-century gold miners. Without a proper plan, the right tools, and resources, no app project can be successful, and certainly not on time and on budget.

Carl Brown has been a fixture within the app consulting world for many years. His views are from the trenches and are battle hardened. The current world of mobile development for hire is fraught with peril, from less-than-reputable development companies to outright scams. The release of the iPhone Software Development Kit (SDK) in 2008 was said to usher in a new gold rush era for developers. The landscape quickly shifted from a small community of just a few thousand registered developers to well over a million in just a few short years. With the influx of developers came a lot of fantastic talent as well as a small minority looking to take advantage of those seeking to mine gold. Carl's unabashed and realistically candid look at the current state of mobile development will help even the most novice entrepreneur get a leg up on the competition.

In an industry that is still very much in its infancy, Carl has more than a decade of experience working with some of the most demanding and challenging projects conceived. Development is often described as having to break down a complex task into thousands or even millions of very small problems. Building a mobile app, whether as a developer, designer, or entrepreneur, follows the same pattern. It is all about knowing how to see each step and how to resolve it properly. Carl's experience with software

consulting stretches back more than 20 years, and he has solved countless app development problems. *App Accomplished* provides guidelines, advice, and recommendations to handle the potential pitfalls that you will encounter along your journey. For the first time in history, the knowledge from more than a decade of mobile experience fueled by hundreds of projects is available in an easy-to-follow reference.

From understanding the project and development life cycle to finding the perfect developer for your project, from conception to marketplace, *App Accomplished* will walk you through the entire app development process. Communicating with developers and designers can often be more challenging than building the business strategy of a project. If you cannot properly convey your intentions and feedback to your partners and team, time and money will be wasted before you can bring your true vision to life.

As someone who has overseen teams that have shipped more than a thousand mobile apps since the release of the iPhone, I wish that every one of our clients would first read *App Accomplished* before hiring us. The information presented, from the author's warnings of potential pitfalls and beyond, provides the reader with the knowledge needed to understand the development process. Those aspiring mobile entrepreneurs who have a rich understanding of the process will encounter fewer disputes, speed bumps, costly change orders, and overall frustration. Thoroughly understanding the topics discussed on the following pages can easily save a not insignificant portion on the cost of development and design by repeating the history of those who have come before you.

—Kyle Richter

Kyle Richter is Chief Executive Officer of MartianCraft, a leading mobile development studio. He is the coauthor of *iOS Components and Frameworks: Understanding the Advanced Features of the iOS SDK* (Addison-Wesley, 2013) and author of *Beginning iOS Social Games* (Apress, 2013) and *Beginning iOS Game Center and Game Kit: For iPhone, iPad, and iPod touch* (Apress, 2011).

Preface

You have an idea for a mobile app. You want to build it, but you don't have the expertise or the time to build it yourself. So what do you do?

You need to know three things:

1. How to distinguish between a developer who can do a good job with your app and one who can't
2. How to work with your chosen developer to get the best result you can
3. Why 1 and 2 are important

Who This Book Is For

This book is for anyone who has an idea for an app. For the purposes of this book, I define *app* as any mobile smartphone or tablet software application that is available for download from one or more of the app stores, like Apple's App Store, Google Play, Amazon's App Store, or Microsoft's Windows Phone Store. This book talks about the skills and processes needed to make a successful app.

No matter what skills you possess (even if you're a programmer), you're unlikely to have all the skills needed yourself. So I'll talk about how to acquire those skills, either by learning them yourself or by hiring or contracting with another person.

This book is about how to turn your app idea into the best possible app, based on the resources you have.

What This Book Is Not

This book is not a "get rich quick on the App Store" book. I don't believe there is a one-size-fits-all formula for guaranteed success.

This book is not about validating or marketing your idea. I've seen apps that I thought were sure-fire hits go nowhere, and I've seen apps that I laughingly insisted no one would waste money on pull in more than $10,000 in one day.

The same old rules of marketing apply for apps: You have to find a real market, and you have to solve a real problem (or get insanely lucky, but don't count on that). If you want to know how to determine if your app is likely to make any money, start by searching online for "lean startup." Once you've decided that people will pay money for your idea, come back to this book for help building it.

This book is not about how to write code. I briefly discuss code, but this is more about process than code. This is also not a book about graphic design or user experience design.

This book is also not the One True Way™ to create an app. It's a collection of techniques and strategies that I've found helpful, but apps can be (and have been) built successfully without them. I'm not saying that you *must* do things the book's way.

Finally, this book, being a book, is not going to be the most up-to-date resource available on mobile app development. Specific how-to instructions and details placed herein would become out of date before this book went to press. Therefore, I've created a companion website and blog at http://AppAccomplished.com that contains more current information. I'll be able to keep the website updated over time, as new tools and platforms are released and old ones fall by the wayside. This way, I can keep the book's content focused on the "what and why" of mobile development; you can find out about the "how" elsewhere.

Why This Book

More than anything else, this book is about how to get your app built while making the most of your time and money. Hiring an app developer is effectively placing a bet (or making an investment). This book gives you the tools and techniques to increase your odds of success as much as I know how.

There's nothing in my professional life I dread more than looking over the code for an app on which someone has spent tens of thousands of dollars or hundreds and hundreds of hours (or both), and having to tell them that they are much, much further from a salable product than they believe.

In addition to the personal unpleasantness, it's bad for the profession, and for everyone involved. Lots of money is wasted, lots of effort is squandered, and many apps that could have made an impact never see the light of day.

There is no 100% dependable way to make sure you have a successful project. This is very important, so I'll say it again: *There is no 100% dependable way to make sure you have a successful project.* It is, however, possible to dramatically

increase your chance of success. Most failed projects that I've been asked to evaluate have failed in particular ways and had warning signs and red flags all along. This book helps you make the most of your odds, avoid the common traps, and recognize the warning signs as they appear.

How This Book Is Organized

To the extent possible (excluding Chapter 1, which is about why you need this book), this book is organized chronologically as you go through the app development process. However, I structured it so that you don't have to read it in order. You should be able to start reading at whatever place is most relevant to your particular project and find references to other relevant material as you need it.

Here's a brief summary of what you'll find in each chapter:

- **Chapter 1, "What Could Possibly Go Wrong?"**—Explains some of the common misconceptions about mobile app development and gives some reasons many app projects fail.
- **Chapter 2, "The App Development Life Cycle"**—Provides an overview of the high-level steps in the app development process.
- **Chapter 3, "Prototyping and Wireframing Your App"**—Explains how to turn your app idea into something a developer can understand how to develop.
- **Chapter 4, "Determining Your App's Components"**—Provides a list of the different kinds of technologies, libraries, features, and functionalities that are commonly used in mobile apps and explains what each is and why you might want to include it in your app.
- **Chapter 5, "Finding the Right Tools"**—Provides a list of the different services and kinds of development environments and tools that are used to build apps and explains why and when you might use them.
- **Chapter 6, "Skill Gap Analysis"**—Explains how to figure out what skills you need to get your app built and how to figure out which ones you are missing and what you might do about that.
- **Chapter 7, "Finding a Developer"**—Explains the different kinds of potential development resources you might use, where each can be found, and the pros and cons of each.
- **Chapter 8, "Interviewing and Selecting a Developer"**—Discusses the process of determining whether a particular developer might be a good fit for your app project.

- **Chapter 9, "Managing to Milestones"**—Explains how app projects can be organized and managed and gives some recommendations about strategies that have worked for me.
- **Chapter 10, "Understanding What You're Getting"**—Explains how to estimate the quality of the app you are getting and the code that is being written for it.
- **Chapter 11, "Pulling the Plug Early"**—Explains how to determine how far off track your project seems and how to decide whether it's recoverable.
- **Chapter 12, "Communicating Using Bugs"**—Explains how to use a bug or issue tracking tool to communicate with your present and future development teams.
- **Chapter 13, "Testing"**—Explains how to find and work with testers of your app to get the quality you need.
- **Chapter 14, "Submission and Beyond"**—Discusses submitting your app to an app store, what to do if it gets rejected, and how to start planning your follow-up release.

The Case Studies

You'll find sidebars like this throughout this book. These are based on real projects that had real problems and provide examples of the issues being discussed. I've anonymized them by avoiding identifying specific apps, people, and companies. Sometimes, when maintaining privacy dictated it, I also changed some details that didn't directly impact the points under illustration. They all really happened, though.

Acknowledgments

I need to start by thanking my wife, Penny, and my daughter, Tamara, for the many hours I spent writing instead of spending time with them. I love you both, and I couldn't have done this without your help and support.

The guidance of my editor, Trina MacDonald, has been invaluable, especially during the initial stages, when we were trying to figure out what this book should be. I'd like to thank my technical reviewers: Wes Miller for his writer's eye and his encouragement, Bob Wesson for his app creator's perspective, and Mark Kolb for another app developer's second opinion. Thanks also to Olivia Basegio, for herding the necessary cats (yours truly included), and Sheri Cain, for making me sound like a professional author.

Thanks also to John and Nicole Wilker of 360|iDev, who encouraged me to become a conference speaker, and Mike Lee, who pushed me to step up my presentation game—without any of whom I'm unlikely to have been asked to write a book in the first place. Likewise, I am grateful to Brandon Alexander, Bill Dudney, Kevin Kim, and Kirby Turner, who shared with me some of their experiences writing their own books, along with Joe Keeley, Graham Lee, Erik Price, David Fox, and (once again) Wes Miller, for reviewing and providing valuable feedback on the early outlines of this book as it was taking shape.

About the Author

Carl Brown (@CarlBrwn) started writing software for client companies while working at EDS in 1993. He became enamored of developing for mobile connected devices in 2005, starting with the Palm VII and moving on to Windows CE. Since 2008, he's been focused primarily on the iOS app market (with some Android thrown in). He's worked on dozens of apps, starting with the Calorie Tracker for LIVESTRONG.com. He's also been brought in to rescue a number of troubled or failing app projects, with varied success. He's a frequent speaker at the annual 360|iDev iOS developer conference and a speaker and organizer with CocoaCoder.org, the largest Mac and iOS developer meet-up group in Austin, Texas, where he lives with his wife and daughter.

Editor's Note: We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can e-mail or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or e-mail address. I will carefully review your comments and share them with the author and editors who worked on the book.

E-mail: trina.macdonald@pearson.com

Mail: Trina MacDonald
Senior Acquisitions Editor
Addison-Wesley/Pearson Education, Inc.
75 Arlington St., Ste. 300
Boston, MA 02116

This page intentionally left blank

1

What Could Possibly Go Wrong?

In my consulting practice, I am often asked to take over, or at least examine, app development projects that are in trouble. Usually, months of effort and tens of thousands of dollars have been expended, and I often find that the quality of the existing development is so bad that it would be less work to just start over. I dread few things more than breaking that bad news to entrepreneurs, business owners, and managers. Those conversations are unpleasant and heartbreaking.

I fervently hope the information in this book can reduce the frequency of those conversations—not just for me but for all competent contract app developers.

One aspect of most of those conversations is surprise that the situation could possibly be so bad. Part of it is pure denial: People never want to find out they've wasted their effort. But I think there's more to it than that. I think that there is a common misconception that apps are easy to make and that any developer ought to be able to make them. Unfortunately, that is simply not the case.

App Projects Are Not Small and Easy

Compared to enterprise software running on a desktop or in a web browser, it's true that apps are smaller and cheaper, but that's really not saying a whole lot. A fairly typical iPhone app project in 2014 can take two or three developers three or four months and can easily run between a hundred

thousand and a quarter of a million lines of code. While that's not a lot compared to some kinds of development, it's not trivial.

Through the years, many studies have reported that a significant percentage of software projects fail (although the percentage can vary wildly). App projects are definitely software projects, and they have many of the same issues and risks as any other software project. App projects can and do fail—some of them spectacularly. I haven't seen any studies specifically on the percentage of app development projects that fail, but I'd expect it to be similar.

I would say that anecdotally more than half and maybe as many as two-thirds of app projects I have knowledge of would be considered failures under the criteria outlined in the following sections. Several examples of such project failures can be found in sidebars in this chapter (and throughout the rest of this book). This is not a scientific survey, and I don't know how representative my experience might be. I have talked to many would-be app creators about what it would take to rescue their failed projects (some I could help, many I could not), and this might skew my experience. However, I know other app developers who have also spent a significant amount of time attempting to rescue failed projects and their anecdotal estimates are comparable.

Defining Project Failure

Let's talk about what I mean by the word *failure* in the context of app development. Different people and different studies use different definitions, which is likely part of the reason that different studies reach such different conclusions. When I say a project *failed*, I mean that one of these four things happened:

- The app failed to ship (that is, didn't become available to users).
- The app failed to work (that is, didn't work as intended for a noticeable percentage of intended users).
- The project cost significantly more money than planned (more than 10% or 20% over budgeted funding).
- The project took significantly more time than planned (more than 10% or 20% over budgeted time).

Let's talk through these situations.

First, the app has to ship. I think we can all agree that if the app was never seen by any user in its target audience, then the project failed. You have to

ship in order to succeed. End of story. This is the least controversial of my criteria.

Second, the app has to work. This doesn't mean that the app has to be bug-free, as virtually none of them are. But the end user has to be able to use it in order for it to be called a success. This can be a subjective criterion; there can be legitimate arguments on both sides about whether an app is "useable" or "functional." For purposes of determining via this criterion whether an app has failed, though, it's usually either clear or irrelevant. (This criterion is irrelevant if the last two criteria fail, and if the app's functionality is in question, they usually do.)

Finally, the app can't have gone significantly over budget or schedule. These final two are the most controversial of my criteria, but I stand by them. I live in a world where I have customers, and those customers are depending on me to produce an app for them. Those customers care how much it's going to cost and when it's going to be done, and they need to plan on that.

Getting Clarity on Functionality

The second criterion, "the app has to work," is necessarily subjective. It depends very much on the nature of the app and the project, and it's just not possible for a book to give an unambiguously measurable way to determine functionality for any possible project. But that doesn't mean you can't.

Every app creator should (although most don't) insist on writing into the contract (or an accompanying statement of work) criteria for determining whether an app is considered fully functional. Agreeing on that up front (and amending it as you go, if needed) can save a lot of disagreement later.

Project Success Is Mostly About Estimates

Notice that the last two criteria—the project cost significantly more money or took significantly more time than planned—make a distinction between the outcome for the app itself and the project that produced it. It's possible (and not uncommon) for an app to be launched successfully but the project that produced it to have been a failure. Many contract app developers who get paid by the hour would likely count such a project as a success. If the app makes it to market and gets good reviews and the client company paid the bill, then most contractors are happy, even if it took twice as long and cost three times as much as planned.

To me, this feels dishonest. If a contracting company tells you that they will build your app for $50,000, but in the end you end up paying $75,000 or $100,000 or more, I consider that a bait-and-switch tactic. And it's unfortunate because it creates an incentive for contracting companies to generate unrealistically low estimates, knowing that they'll just charge the customer more in the end.

Project Success Is Also About Scope

An estimate, however, is only valid for a given *scope* (a given set of features). If new functionality is added to a project, then the budget and schedule have to be revised.

There is sometimes a disagreement about whether a given feature or bug or issue is within scope. Assuming that everyone is acting in good faith, such disagreements result from failure in the communication process between the client and contractor. Scope should be documented unambiguously.

Communication failures arise consistently, though, and often contractors who work on a time-and-materials basis choose not to clearly define the scope. Assuming that they can talk their clients into continuing to pay, this is to their advantage.

Another tactic contractors use is ignoring the scope as agreed and beginning work on new features as if they are in scope. This leads to unpleasant conversations after the work has been done. It should be incumbent on a contract developer to make sure any changes to the scope are mutually agreed upon and documented before work on the new functionality begins.

The Specter of Unprofitability

There is one important factor about an app project's success that these criteria do not take into account: *profitability*. I can't consider profitability in my criteria because I don't generally have access to app revenue information. All I (or any contract app developer) can do is build the best app I can within the time and budget I promised.

But profitability (or at least return on investment [ROI]) should be important to app creators. And if I had access to profitability information and could consider ROI goals as a failure criterion, then I would expect an even higher percentage of app projects to be failures. As it stands, app projects have to make it all the way through the funnel depicted in Figure 1.1 to be ultimately successful. I don't know what percentage of apps fail at each stage, but I'm confident that a large percentage of them don't make it all the way to profitability.

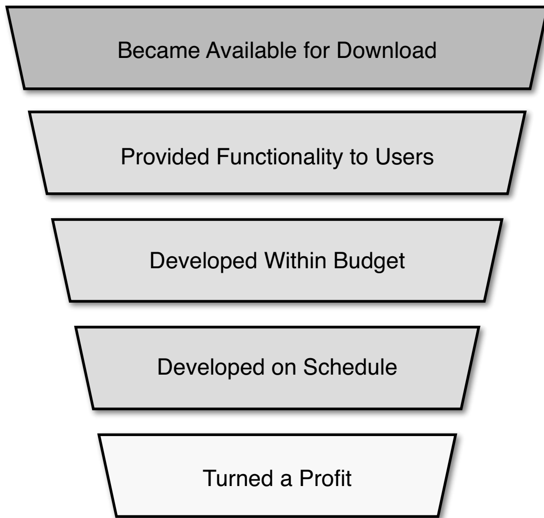

Figure 1.1
App projects must work through this funnel to be ultimately successful to their creators.

Apps Are Not Easy to Program

App development requires skill and experience. Because of the relative lack of raw computing power, one of the difficulties with mobile development is that the code in an app is fairly interdependent. It's unfortunately way too easy for this piece of code in this part of the app over here to have an effect on that piece of code in that part of the app over there. So the developer has to be extra careful.

Our Expectations Were Set by the Web…

We software developers as an industry got somewhat spoiled during the late 1990s and early 2000s because much of the exciting work being done in technology used web technologies and web servers. And the good thing about web servers at the dawn of the century is that really powerful servers could be purchased for a modest amount of money. With the rise of the 64-bit PC CPU, servers could hold more RAM than most web transactions could ever use, and it became not just feasible but expected for servers to hold entire databases in memory cache.

Add to that the fact that web protocols themselves were becoming widely adopted, and it became easier than ever to build a web application that

could be reached by millions or tens of millions of users with a relatively small team of programmers working out of someone's garage. Web protocols also allowed the creation of much richer user experiences than had previously been possible for widely distributed applications because the layout rendering, font handling, and image display were offloaded to the machine running the web browser.

The good thing about this kind of web development is that each web request is mostly independent of every other web request, and the servers have enough power that each request can have all the resources it wants. Under enough traffic, that ceases to be true, but then at that point, it's relatively simple (assuming that things are engineered correctly) to put more web servers in parallel and use the networking infrastructure to share the incoming traffic load between them. Eventually if a service became popular enough, it would start running into tricky scaling problems that required a lot of performance tuning, but a lot of companies avoided this entirely.

…And Then Invalidated in the Mobile World

But mobile development has different constraints and requires different techniques. There's never as much memory in the device as the app developer wants because memory banks draw too much power, and battery life is the overriding concern of the device vendor. All the graphics have to be drawn on the device's screen as well, so you don't have the luxury of offloading it to another machine as you can with the web. And apps don't even have exclusive use of the device; there's often mail being fetched or music being played in the background, and the app could be interrupted at any instant if a phone call or text message comes in. This means that apps are constantly resource constrained. It's a delicate balancing act that doesn't happen so often in the web world.

So the situation in mobile app development today is much closer to the client/server programming of the 1980s and 1990s than the web programming of the early 2000s. The primary differences from a programming standpoint are that mobile platforms have far, far better development environments (developer tools, frameworks, libraries, and components) than existed back in the heyday of client/server, and mobile apps generally only have to implement the client half of the client/server equation. Figure 1.2 shows the major differences between web, enterprise, and mobile development.

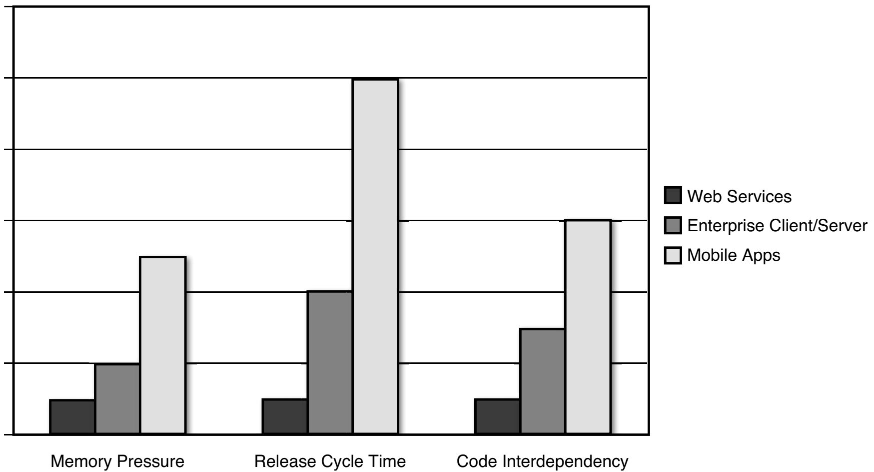

Figure 1.2
Visualization of the three largest differences between mobile app development and other common types. (Shorter bars are better in this case.)

Mobile Apps Are Outside Your Control

The other thing that web servers made us lazy about is bugs (and I say that as a recovering ASP.Net and Ruby on Rails programmer). The nice thing about writing server code is that when something goes wrong, you can figure out what happened (hopefully) from the server logs, and you can make a change and push it to the servers and fix the bug before it affects too many of your users (again, hopefully). There were times, especially immediately after particularly poorly tested releases, that teams I've worked on have done a dozen or so different releases of web server code in a day, each one fixing one or more bugs or performance problems. And the great thing there is that as soon as the server is patched, every subsequent web request will get the fixed behavior.

In mobile app development, by contrast, you can't iterate as quickly. Once an app is installed on a user's phone, only that user can decide to upgrade to the next version of the app. Depending on the bug, the user might choose to just delete your app instead of update it (and maybe even leave a bad review). But even worse, in mobile app development, you don't ever have direct access to the machine your code is running on, so if there's a problem, you can't just look at your server logs and find it. Tracking down a bug that's happening only on a subset of smartphones and tablets and doesn't happen for the developer can be a real nightmare.

And last but not least, there's often the dreaded "Waiting for Review" lag. Although some app stores, like Google Play, will let you upload new app versions as soon as you've fixed a bug, other stores, like those from Apple and Amazon, require your update to be reviewed before it can be released, and that causes even more time to elapse between when a bug is brought to your attention and when a fix is available to your users.

Poor Skill Set Fit

Some parts of mobile development are more difficult than others, and some are relatively easy. Unfortunately, it's not immediately obvious which parts are which, and it varies from platform to platform. It's actually common for a developer who is familiar with one programming language or library to believe that familiarity is more relevant when moving to a new programming language or platform than it actually is.

Example: Threading on the iPhone

Most development platforms and operating systems these days have something called *threads*, which you can think of as containers inside an app into which commands are placed so that they can be executed. By having multiple threads, an app can do more than one thing at the same time (or nearly the same time). For example, one thread can be redrawing the app's screen as it scrolls, while another thread is downloading a video that will be displayed at a later time.

If a programmer knows how threads work, and that programmer learns that iOS (the operating system that runs on the iPhone and iPad) has threads, that programmer can easily assume that he or she knows how to make the iPhone do multiple things simultaneously. But he or she would be wrong. Although the iPhone does have threads, it also has queues, and Apple recommends that programmers should almost always use queues, and Apple's libraries then manage the threads automatically. Programmers who don't realize this often use the wrong mechanism and create their own threads, which can cause the app to malfunction under load.

The bad news is that when there isn't a whole lot going on (for example, when you're using small amounts of test data), spawning your own threads works fine. If the programmer assumes that it will work the same way with a lot of data, then he or she will likely fail to test at higher loads. Such an app may develop unexpected, undesired, and likely unacceptable behavior right at the end of the development cycle.

Your App Isn't Just Any App

The takeaway is that different apps need different skills, and just because developers did a good job with someone else's app doesn't necessarily mean they can do a good job with yours, unless the apps are really similar. Unfortunately, you can't always rely on developers to volunteer what skills your app needs that they lack (especially since it's in their best interest for you to hire them, and for you to do that, they want you to think they know what they are doing).

But don't worry. In Chapter 4, "Determining Your App's Components," you'll learn how to figure out what skills your app needs, and in Chapters 7, "Finding a Developer," and 8, "Interviewing and Selecting a Developer," you'll learn how to select the developer who is the best fit you can find.

The Case of the Videos Hung by a Thread

I was once asked to talk to app creators who were having problems with their app. It was presented to me as yet another one of the many mythical "it's 90% of the way to being done" projects (which has never actually been the case…see the section later in this chapter called "That Last 10%"). The app creators had fired the original developer and were looking for someone to finish the project before an impending deadline.

The app creators had actually done a better job at selecting a developer than I am used to seeing. The app being built was an audio/visual (A/V) app that let users create their own multimedia content. The developer who had been chosen had built several (as I remember it) disc jockey and A/V mixing apps that seem to have done well in Apple's App Store. The developer had also been recommended by an acquaintance of the app creators. So the developer seemed like a logical fit from the perspectives of both skill set and work ethic.

One critical feature was missing, though. The app required that the user would be able to upload completed A/V files to a server, and that feature was a complete disaster. First off, the app, as written, required the user to leave the phone on the *Uploading* screen until the upload was complete. There was no status indicator, so the user didn't know how long the upload would take, and the app appeared to be *hung* (also referred to as being *locked-up*, or unresponsive). In addition, if the user went to any other screen in the app, touched the Home button, launched another app, or received a phone call or text message, the upload would stop. As a result, the upload almost never worked.

The developer (I'm told) insisted to the app creators that what they wanted wasn't possible, despite the fact that there were a number of apps already in the app store that did the same thing. After some dispute to which I wasn't

a party, the app creators and the developer parted ways, and then the app creators came to me.

By looking at the code, I discovered that the previous developer seemed to have no concept of threads or background tasks (discussed further in Chapter 4). It appeared that the developer had come to mobile development from Flash development, which is a technology where background tasks aren't an option. My guess is that the successful apps this developer had written previously managed to work without needing to use more than one thread, but large network data transfers absolutely require multiple things to be happening simultaneously.

Unfortunately, it wasn't feasible for the app to be completed before the deadline, so I wasn't able to solve the app creators' immediate problem.

As discussed further in Chapter 8, there are two questions that the app creators could have asked that might have prevented this problem:

- Have you ever written code that performed a large file upload? This question would have illuminated the potential risk and allowed everyone to manage it differently.
- What resources for honing mobile development skills have you recently used, and where would you turn if something unfamiliar came up during the project? Apple and many third parties have created tutorials, videos, and documents that explain how large file uploads should be done on a background thread. Asking this question would have uncovered the fact that this developer wasn't familiar with any of those.

If You Get a Good Developer, You Still Have to Worry

Although it is certainly true that inexperienced programmers have far lower success rates than established firms, despite what the established firms might want you to believe, there is plenty of failure to go around. Once cost and schedule overruns are considered, even established firms have an uncomfortably high percentage of failures.

The good news is that with well-established firms, in order for a project to fail, something has to have gone wrong. The bad news is that these days, things go wrong surprisingly often.

Inexperienced Members of an Experienced Firm

In the current app development business climate, demand is outstripping supply by a large margin. This causes a number of problems: Hiring is nearly constant, turnover is high, and responsibilities are in flux pretty much across

the board. It's not unusual for most or all of a project team at a reputable development firm to have been hired in the past few months.

It's also common for existing employees to be promoted from developer to lead developer, often so there is someone to supervise newly hired junior developers placed under them. The skills needed to be a successful member of a project team are but a subset of the skills needed to lead a project team, and mistakes made when learning development supervision and project management skills can mean the difference between the success and failure of a project.

Subcontractors

Another consequence of demand outstripping supply is that firms often have more business than they can handle. Instead of turning away business, many subcontract some of their business out to other developers, taking some percentage of the revenue off the top. Since the firm that signed the deal wants to make some money, it stands to reason that the subcontractor will be working for less money than the original firm's prices (that is, the subcontractor is almost always a cheaper developer). The communication between the original firm and the subcontractor is also likely to be inferior to the communication that the original firm would have with its own employees.

Juggling Resources

When any project at any company goes wrong, the company working on it usually tries to fix it. The good news is that if your project is being developed by a reputable firm and something goes wrong, the company will likely do its best to marshal its resources and attempt to get your project back on track. The bad news is that if your project is going fine and some other project at the firm you are paying goes off the rails, the resources on your project may be pulled off (fully or partially) to try to shore up that other project. This can have a negative impact on your project and can seem to come out of nowhere, even when a project seems to be going well.

Contractor Company Overhead

Another question with large firms, reputable or otherwise, is what the firm considers *billable*. Some firms consider time spent on internal activities, meetings, emails, and conferences calls billable to you, even when those activities don't involve your project (or involve it only peripherally). This can cause your budget to get used up by activities that aren't related to your project.

Another overhead item is hours billed to project management (often at a high rate). If your firm is billing you for project management time, you need to make sure that you understand what that time is spent on and decide whether that amount of money is worth it to you.

It's Important to Know What You've Signed Up For

The solution to all these issues (and many more) is to clearly document up front exactly what you're paying for. You need to know what team is actually going to be working on your project, what their experience is, what their relationship to the firm you are paying is, and under what, if any, circumstances those staffing arrangements might change. You need to know what the firm considers billable work and what that work will be. If you're paying higher rates to get an established, experienced development firm, then you are paying to have experienced staff working on your project, and you need to make sure that happens.

The Idea Is Not More Important Than the Execution

A common fallacy is that once you get an idea for a great app, you're most of the way there. In fact, that's not the case at all. As with most other inventions, a truly exceptional idea is still only the first step in the process of making an app.

In the process of making an app out of that idea, literally thousands of additional decisions have to be made. Some of those decisions involve colors, graphics, data storage, workflow, architecture, layout, transitions, animations, and monetization, and each of these requires even more related decisions. Some of the decisions you need to make have a wide-ranging impact on the success of the app, and some have little or none. It's not always possible to tell how much of an app's success or failure is attributable to any particular decision. In general, I've found that app creators often underestimate the impact of each of the little decisions.

Each decision must be executed upon and turned into code, and that code becomes part of the final app. There are lots of opportunities for poor execution in this phase that can render an otherwise fantastic idea unworkable.

There Is No Idea So Good That It Can't Be Poorly Executed

There's a tendency I've seen among app creators to become so enamored of their idea that they don't want to worry about anything else. They often say (or think) they are looking for a developer to "implement their idea," by which

they mean to make all the decisions and execute those decisions, often in return for half (or less) of the app's eventual revenue. I've never seen or heard of this strategy working out (which doesn't mean it can't, but it's definitely a low-percentage bet).

The root cause of this situation is usually that the app creator doesn't participate in the decision-making process except at a very high level. In such a case, the developers end up implementing their own vision instead of the app creator's, slavishly copying some other app, or just guessing what the app creator wants.

In the virtually certain event that the app creator is unsatisfied with the result, an adversarial relationship usually forms between the app creator and the developer. The app creators point out things that they consider to be obvious deficiencies (but generally can't articulate or communicate alternatives in sufficient detail to be implemented). The developer, frustrated, builds something else that is still unlikely to appease the app creators. And the cycle often repeats.

Programmers often refer to this as "rock fetching," and it can be hugely frustrating. Imagine the following scenario:

> Your boss: "Bring me a rock."
>
> You, after bringing back a rock: "Here is a rock for you."
>
> Your boss: "I don't like that one. Bring me a different rock."
>
> You, after getting a different one: "How about this one?"
>
> Your boss: "No, I don't want that one either."
>
> You: "What's wrong with it?"
>
> Your boss: "I just don't like it."
>
> You: "What kind of rock are you looking for?"
>
> Your boss: "A better one than that one."
>
> You: "Better how?"
>
> Your boss: "You know, better. Rockier. More rock-like."
>
> You, confused and hesitating: "Uhhhhh."
>
> Your boss: "What are you waiting for? Bring me a rock."

Most people find rock fetching irritating. It certainly doesn't facilitate a person's best work. Given enough of this, most people will eventually quit. But when app creators refuse to involve themselves with detailed questions about how an app should look or work and either don't know or can't communicate what they really want, they create a rock-fetching scenario.

A word of warning: Some developers love this kind of app creator and are happy to continue billing by the hour to write code that will be rejected until the app creator runs out of money. Personally, I find this distasteful, but some legitimate developers make the argument that they aren't responsible for their customers' whims or lack of requirements. And they have a point. Caveat emptor.

The Case of "Facebook for *Insert Demographic Here*" for Four Times the Price

I was once approached by an app creator who had been unable to get his app launched, despite having spent many months and many tens of thousands of dollars having it developed. From his initial description, I could tell that the project was in a late-stage failure mode that is always a hard state to recover from. I arranged to meet him for lunch and sat down with him to talk about what had been happening with his project.

At lunch, he spent roughly twice as much time telling me about how great his app idea was as he spent on how the project had gone. This was difficult for me because (1) I was there to figure out what was going on with his project, not how much money he thought his app would make and (2) the idea wasn't good. (Not that it matters for this story, but his idea was "I'm going to make a new social network just like Facebook, except targeted at this one particular market segment of the population." I hear that idea every few months with a different segment of the population, and none of them have gotten any traction.)

He asked his current developer to send him the latest copy of the source code they had built for him so far, and he gave it to me to look at. Based on the wireframes, the requirements, and the source control, I thought he had spent about three times as long and about four times as much money as I would have estimated in order to have built it by myself.

As I looked through the code, I realized what had been happening. The developers had built many, many different versions of the app, one after the other. They would build a version and present it to the app creator. He would tell them to do it differently, and perhaps make some vague suggestions. They would then pile his new suggestions on top of the existing code and show it to him again. Over time, they had written tons and tons of code, much of which was from several iterations ago and no longer called but had never been cleaned up. It was incredibly difficult and time-consuming just to understand what was supposed to do what and what should have been deleted. It was a horrid mess.

I wrote up my set of recommendations for getting his code cleaned up and for creating a well-defined set of features and documenting them before doing any more development. I gave him my recommendations document

and never heard back (which wasn't surprising, as I didn't have time to take on more development tasks at the time, so having me do his development wasn't an option). As near as I can tell, that app never made it to any app store.

Unwillingness to Delegate: Micromanaging

If refusing to get involved in discussions about the details of their app is on one end of the app creator engagement spectrum, the opposite extreme is micromanaging, and it's no less destructive.

Some app creators insist on being far too involved in the minutia of their projects. They question the necessity of every code change and want justifications for every hour spent. They're insistent on getting exactly what they think they want and/or spending as little money as possible, but they end up forcing work to be done over (and therefore wasted) and making people wait on them. Causing developers to wait for someone to make a decision is a waste of time and money, and causing them to have to do something over is a waste of time, money, and goodwill.

It's important for app creators to have control over what happens on their projects. I'm not trying to contradict that at all. The question is one of frequency. The vast majority of the time developers spend on a project should be time spent doing development. That's what developers are being paid to do, and it's where their expertise is. That sounds obvious, but it's quite often not what happens.

There's a way for app creators to maintain control of their projects, and it's to make (and document) decisions the developer needs to have made before the developer needs them, so the developer doesn't have to wait (or guess wrong and do it over). A good developer should be able to generate a list of questions that are going to need to be answered far enough ahead of time to give the app creator time to think and decide. If you as an app creator find that your developer is having to wait on you, or that you are frequently asking your developer to redo things a different way, it's time to have a conversation about doing a better job about defining requirements.

We'll discuss this more in Chapter 2, "The App Development Life Cycle." For now, understand that it's not necessary for all the requirements for an entire app to be completely documented before any development starts, but it is important for developers to know what's required of the piece of functionality they are currently working on when they start it. Not doing so leads to wasted time and money and can put the whole project at risk.

Bikeshedding

Some projects feature endless repetitive design meetings with arguments over font sizes, RGB color values, and pixel-level control placement. The developers are paralyzed and spend a lot of (billable) time waiting for decisions to be made by the client, and progress is correspondingly slowed.

I'm not implying that control placement or color choice aren't important—they are—but they should take up a relatively small portion of the overall project budget and schedule (at least for the vast majority of apps).

C. Northcote Parkinson coined the term *bikeshedding* in 1957 for groups spending far more time arguing about things that don't matter than things that do. The canonical example is a group of townspeople tasked with commenting on plans for a nuclear reactor spending their time arguing about the color of the bike shed at the reactor. Obviously, the color of the bike shed makes no difference to the efficiency or safety of the reactor, but it's something that everyone in the group can feel qualified to have an opinion about. Everyone wants his or her opinion to be heard about *something* and to leave a mark on the project, so any trivial item can become a source of arguments. For the important things (like cooling redundancy and radiation shielding), the nuclear experts' opinions are usually left unquestioned because no townspeople in the group feel qualified to argue those points and don't want to be responsible if they turn out to have been wrong.

Most app creators don't have much (if any) experience with app programming, and so they don't feel qualified to weigh in on issues of coding style and data models. They do, however, often feel that they are qualified to give opinions about colors and fonts and graphic design. So in trying to feel in control, they cause large amounts of time to be spent on noncritical items.

Poorly Defined Requirements

With software, the old adage is true: "The devil is in the details." Software can have lots of details, and each one has to be decided upon and communicated. As discussed in the proceeding sections, sometimes the details aren't considered important, and sometimes they become the source of arguments, but sometimes the problem is one of communication. On many projects, the app creator has an understanding of what he or she wants the app to do but does a poor job of transferring that understanding to the developer who is doing the implementation.

Sometimes this is because the app creator is trying to avoid what he or she considers unnecessary paperwork. When this is the case, requirements

documents, if they exist at all, often take the form of lists of bullet points—short phrases that can mean very different things to different people. Often the developers don't object to this because they don't like paperwork, either. Then the developers implement what they think is meant by a given line item, the app creator thinks that work has just started on it, and the cycle repeats for a while. It usually comes to a head when the developers check some number of items off the list, and the app creator objects because those items aren't done yet. Arguments and recriminations may follow from there, and the likelihood of a successful project is greatly diminished.

Other times, the opposite is true. On projects for very large companies, sometimes the requirements document is a large binder full of contradictory legalese that is the output of many rounds of the request for proposal (RFP) process. Verbiage from an early section implies that a feature should be implemented one way, and many pages later in the binder, an answer to an RFP question seems to say that a different implementation is required. Jargon can be a problem here, too, as large organizations often create their own vocabulary. Here, the problem is not a lack of detail but a lack of clarity.

In either case, what ends up happening is that both sides work on the project as if they understand each other—after all, there is documentation that purportedly explains what should be built. The project can go a long time like this, with both sides thinking that everything is going well. And then one day, one side says something that the other side disagrees with, and then the gloves come off. Voices are raised and accusations fly as the two sides begin to realize how far apart they are. Success is not a likely outcome.

The Case of the Required Preferences

I was once asked to look at an incomplete app project that had already passed its initial estimated completion date. It had a requirements document that was a giant list of one-liners like "The user must be able to specify preferences."

The developer interpreted this particular line to mean that the app was required to have a settings screen (which it did). On the other hand, the app creator expected an elaborate menu system that synced with the server so that if the user had an iPhone and an Android tablet and changed the preferences on one that those preferences would be reflected on the other. The app creator justified this by pointing out another line in the document that said that the iOS and Android versions should be "functionally interchangeable" (or words to that effect).

The developer had expected this requirement to take at most a day, but what the app creator wanted would take multiple developers a week or more, including adding additional capabilities to the server platform. And because

this was just one of many insufficiently specified requirements, it was no surprise (at least to me) that the project had run way over on both schedule and budget.

The other problem was that the discrepancy between the developer's and app creator's expectations didn't come to light until well past the point when the project was initially estimated to have been completed, when both sides were already upset. If requirements are vague (as they sometimes need to be at the beginning of a project), the time to get clarification is early in the process, not once the relationship is already falling apart.

In the case of this app, though, neither side wanted to spend the effort to flesh out the details in the requirements document up front, so they ended up with a mess at the end.

Out-of-Date Requirements Documentation

Yet another way projects go wrong is when the requirements documentation exists but is not kept up to date as things change. This often happens in projects that involve multiple organizations and many simultaneous conversations. Decisions get made, and the two people on the phone know what was discussed, but nothing gets written down, and the rest of the folks on the project are left in the dark. Alternatively, sometimes the requirements document does get updated, but the change doesn't get announced, and other parties execute the instructions from an old copy of the document.

Requirements documents often start out as multipage documents that are basically tables of contents. Someone (usually a single person) starts the project with every intention of documenting all the requirements and begins by writing a three-page list of stuff that needs to be documented. Then as the project goes on, less and less gets written on the requirements document each day, until finally it ends up a derelict Word file on some shared drive somewhere and serves only to confuse the unwary.

Another common setup is that someone sets up a project wiki server with the expectation that people will use it to document the project, but the wiki never ends up being integrated into the project's workflow. The content on it grows stale over time, and the wiki ends up being worse than useless because no one knows whether it can be trusted.

The way to solve this problem is to force the documentation to be part of the workflow for the project—either documentation that the quality assurance (QA) team uses to do testing (so the team opens bug reports when the documentation doesn't match the reality) or in the form of a

suite of automated acceptance tests that are run on a frequent (or at least periodic) basis. The automated acceptance test idea works especially well for documenting the requirements between the server development team and the mobile development team.

Without some way to bring to light discrepancies between the documented requirements and reality, documentation usually becomes out of date to some degree during a project, and this can cause headaches for everyone involved.

The Case of the API Document That Became an End in Itself

I was once a subcontractor on a mobile project that had a server component that was being written simultaneously. The project had someone with the title "architect" whose job was to oversee the technical teams. (I was the senior developer on the iOS team, and there were Android, server, and graphic design teams as well.)

One of the required deliverables to the client was a document that explained the interactions between the mobile client and the server (I presume so that if the client wanted a Windows Phone or Blackberry version of the app to be created at a later date, they would have enough information to do so). The project architect worked with a contract technical writer to build this document.

The API document was running behind schedule, and the architect was pushing to get it completed. The problem was that the focus of the document was on the document itself, and not on the service it was supposed to be documenting.

From time to time, one of the developers would run into an issue that required a change to the contract between the server and the client. For example, the iOS developer would see that the graphics designer had added an element like "date user joined" on a wireframe, but the server wasn't providing that information, or the server developer would realize that two different API calls could be combined to reduce latency since they were always called together. At that point, the developer would figure out what needed to change, and the server developer would make the change and send a note to the architect. But that information would never make it into the document. So when a different developer started working on that API call (for example, implementing it for Android this time instead of iOS), that developer would be working from a document that was now out of date.

Easily 15% (any maybe 20%) of the time spent working on the networking between the mobile devices and the server ended up being wasted because the tool that was supposed to make it easier for the developers to communicate (the document) became focused not on furthering the project but on checking a box on a checklist of deliverables.

Constantly Changing Requirements

Another thing that can cause a project to fail is indecision. Many projects start without a clear idea of what needs to be accomplished and have the design spend months changing over and over while never getting any closer to a ship date. Hundreds of thousands of dollars get spent, and there's no evidence that the design changed for the better during the process.

Sometimes the trigger for this is user testing; sometimes it's anecdotal, with someone showing the app to a friend or two and the friend(s) not immediately understanding how the app is supposed to be used. Sometimes someone says it just doesn't feel like the design is progressing in the right direction. Sometimes the desire is to find something "better" than some competitor's app, but with no clear idea of what "better" would look like. However it starts, it costs a lot of money and doesn't accomplish much (if anything).

What makes this especially hard is that there are times you can tell from user testing that an app design really isn't working. You don't want to ship a design that people can't figure out, so it's important to get a better design. The problem is the amount of money that change will cost.

When I see this happen, it's usually for apps that went from concept directly to coding and skipped the prototype phase. The point of a prototype is to get a design in front of people to figure out whether it works in the cheapest way possible. If you figure out that users aren't understanding (or liking) your app, you can try a different design (or several different designs) without needing to have a single line of code written. If there's confusion in the middle of a project, you could even hopefully move all your development effort to a part of the app that you know will stay the same (like data storage or networking or something) while you show potential users several different prototypes in an effort to get clarification. (For more information, see Chapter 3, "Prototyping and Wireframing Your App.")

But if you are well into building your app when you run into problems and you try to quickly come up with a new design that you think will be better and have your developers start coding on it before you know whether it's going to actually improve the user experience, you're starting a chain of events that can lead to months of churn with nothing useful to show.

Leaving the Worst for Last

A constant theme I hear when talking to app creators about their failed projects is that things seemed to be going very well right up until the point where they thought they were about to be done.

It's rare that a project fails a week or two after starting (and if that happens, it's usually due to an external event). This is because at the beginning of a project (or at least the beginning of the development phase of a project), there's often nothing to see, so expectations are low. It's only when the project has been going on long enough that there should be something to see that expectations are raised to the point that failure seems an option.

A common pattern is that the developer periodically meets with the app creator and says that progress is being made. Something might be shown at the meeting, but there seems to be plenty of time in the schedule to get the rest of the work done. As the time remaining in the project dwindles, the app creator might begin to be uncomfortable but is assured that everything is still on schedule, and there's nothing to worry about. It's only when the app creator looks at the list of outstanding items and sees that the schedule is almost gone that the platitudes of the developer start to ring hollow, and the app creator can no longer believe that everything is on track. This is when the failure becomes obvious.

There are two ways this can happen. First, the developer may be dishonest and just lie to the app creator, trying to get as much money out of the deal as possible. Second, the developer may be incompetent and genuinely just as surprised as the app creator when things seem to fall apart at the end. The good news is that the same remedy works for both of these cases.

To avoid failure, it's important to identify the pieces of a project that have the most risk and push them as early into the project as you can. That way, if there is going to be a problem, you can find out about it before you've spent most of your budget and you still have money to pay for a different developer.

For what it's worth, the two things that I find to be the most risky on the majority of projects are integration and performance. Not coincidentally, these are often the last two things that are done in a project before testing starts in earnest. This is not the best risk management strategy.

Figure 1.3 shows a typical project development schedule. Note how right after the project begins, many tasks are started in parallel by several different developers, and only toward the end is their work integrated (connected together) and tested for functionality and performance.

Figure 1.4 shows a different schedule for the same project. Note how much earlier the first integration and performance test occurs. This is a much safer plan because, if problems show up in the test, there is far more time to deal with them.

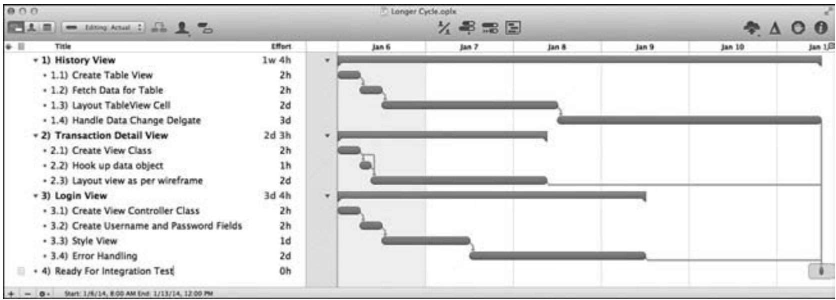

Figure 1.3
A simple Gantt chart for part of a fictitious project. Note how all three views are created and styled before the integration test starts at item #4.

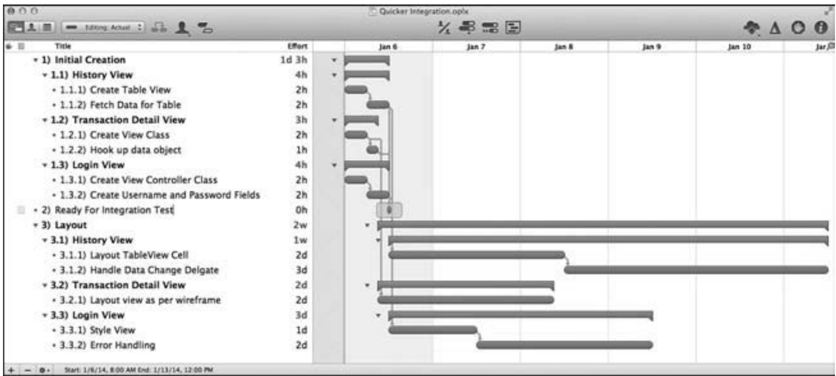

Figure 1.4
An alternate Gantt chart for the same part of the same fictitious project as in Figure 1.3. Note that the integration test has been moved much sooner in the project, to item #2.

The downside to using this technique is often calendar time. Typically you can complete an app in the smallest amount of time if the project is chopped up into many small pieces of functionality, and each piece has a different programmer, and all their work is hooked together at the end. The problem is that this is a high-risk strategy: If something goes wrong, you don't know it until the very end, and then you might not have money left to fix it. Many development shops love this model, though, because they can extract the maximum number of dollars in the minimum amount of time, and they don't have to show an app that actually works until most of the money is already paid.

But if you want to reduce your risk, time is on your side. By having your developer build a piece at a time and not moving on until it's integrated and working, you reduce the amount of your budget that you spend before you have evidence that things are proceeding the way you need them to.

The Case of Throwing Out the UI with the Bathwater

I once was brought into the middle of an iPhone project that had a fairly busy user interface. It wasn't badly designed, but it crammed a whole bunch of information into a very small portion of an already small screen.

Some time before I was brought onto the project, a decision was made that the interface would use an Apple technology called Auto Layout, which Apple had recently released to developers. Auto Layout is a cool technology that allows you to program a user interface once, and then it adjusts to different screen sizes and orientations. It's a nice time-saver for developers (once they get the hang of it) because they no longer have to write a bunch of code that figures out what the X and Y coordinates of each interface element need to be for the current screen size and what else has already been added to it.

The downside of Auto Layout is that it requires the device to solve fairly complicated equations every time the screen needs to be redrawn to figure out where everything needs to be placed. The good news is that computer processors are very good at solving these kinds of equations. The bad news is that sometimes, the processor in a mobile phone has better things to do.

Two other mistakes were made on this project that are relevant for this discussion. First, the contract didn't specify which models of iPhones the app needed to support. Second, the project schedule called for performance testing to be done at the end of the project.

Toward the end of the project, the app creator declared that the app was too slow on older iPhone models. The app had never been tested on iPhone models that old because the development team assumed that they weren't going to be supported. Effort was put into making the app faster, but it couldn't be made fast enough. The older, slower processors in the much older iPhones just didn't have enough power to solve the equations to lay out the screen while doing everything else that they needed to do.

And so, at the end of the project, long after the budget and schedule were exhausted, the Auto Layout code had to be removed from the user interface, rendering all the work than had been done on Auto Layout wasted and requiring lots of additional code to replace what Auto Layout had been doing. Earlier performance testing would have saved a lot of wasted effort.

Cost Overruns

This book spends quite a bit of time discussing risks. The cost of a project turning out to be way more than expected is probably the most likely failure mode in the contract app development space. This is sad because app development billing rates start higher than those of many (if not most) other forms of contract development right now. Costs are high under even the best of circumstances, and they become downright stratospheric if not kept under control.

Much of this book focuses on getting you your money's worth, but the following sections discuss a few of the most common ways that costs get out of control.

Unnecessary Staffing

One way that projects run way over budget is by having too many developers. The more developers a project has, the more communication effort is required. Each additional developer is not only less productive than the previous one but actually slows down the existing developers. This effect was documented in 1975 in *The Mythical Man-Month* by Fred Brooks (which I mention several more times in this book).

Especially when your development company has a number of developers on salary who are idle, the company may pressure you to have more developers on your project than you need. Make sure that the company can articulate what every developer will be doing and why each is necessary before you approve additional headcount.

Unnecessary or Wasteful Work

Not all work gets you closer to the project's goal. Some work turns out not to have been worth it. Under what circumstances should you have to pay for that?

First, understand that just because a particular piece of work doesn't make it into the final product doesn't mean it was wasteful. Sometimes there's more than one way to implement something, and it makes sense to spend time experimenting with multiple possibilities to figure out which is the best fit for your app. It's better to do that than to guess with no data and risk ending up with an unworkable solution. These experiments are often referred to as *spikes*, and we discuss them more in Chapter 2. Mockups and prototypes are also useful work that won't ever ship, and we discuss them at length in Chapter 3.

However, some kinds of work are wasted. What happens when a developer makes a mistake? What happens when fixing the bugs in a feature turns out to take twice as long as building that feature initially? What happens when the developer you got was less experienced than the developer you needed and ended up making a mess? Under many contracts, that cost is passed directly to the app creator, so there's not a lot of incentive for the developer to get it right the first time.

Unproductive Billable Time

We touched on this earlier: Developers should be developing. Try to keep meetings and administrative tasks to a minimum and make sure you aren't being billed for time when the developer is not working on your project. (For example, time spent eating lunch should not normally be billed to you.) Note that explaining and documenting the code that is being written should be considered part of development, not administrative time.

Unexpected Bugs

Sometimes projects run into bugs that take a while to fix. Sometimes those bugs are in code that the developer wrote, and sometimes they're in third-party libraries or the framework for the mobile platform itself. Bugs always happen, but most of the time they don't take a project off the rails. Sometimes they do. However, project-endangering bugs should not happen in areas of the project that are similar to what your developer has done before. Talk to your developers about what risks they see in your project before you start. (See the "Gap Analysis" section of Chapter 8 for more details.)

Unforeseen Circumstances

Sometimes bad things happen. Sometimes developers become ill or quit or have to leave a project to care for a sick family member. Developers are human, and most humans have chaotic and unpredictable periods at some points in their lives. Discuss what might happen in such a case with your developer before work starts. (See the "Contingency Plans" section of Chapter 8 for more details.)

Poor or Changing Requirements

As discussed earlier in this chapter, requirements need to be clearly decided, communicated, and understood. If they're not, they might as well be randomly generated.

Poor Initial Estimation

One of the things that can throw project costs far afield is unforeseen work being "discovered" or "found" during the development process. If this sort of thing occurs and wasn't a result of bad requirements, then it was most likely missed during the estimation process. Ask your developer before work starts about who will pay for poor estimates and how much. (I suggest that you at least should never have to pay full price for work that wasn't estimated.)

The other common result of poor estimation is everything just taking longer than expected. This could be a result of insufficient information during the estimation process (which happens a lot on projects that involve taking over a code base that someone else wrote and the code base turns out to be worse than you thought). It also happens when the developers who end up on your project are slower and less experienced than the developers that the person doing the estimation had in mind (in which case you probably shouldn't be paying the same hourly rate as the estimator had in mind either).

That Last 10%

Many people have come to me with projects that they believe are "90% complete." They never have been. It's hard to know exactly how far along a project actually is, but when someone tells you they're 80% or 90% of the way done with a project, they're almost always wrong. Why does this happen so often?

Well, first off, most people don't actually bother to go through the effort of really measuring project status but want to act as if they know what they're doing. So they tend to make up numbers, and 80% and 90% are good round, made-up numbers. Really, it's just a common and convenient lie told by developers and project managers the world over. When people tell you they're 80% or 90% done, ask them how they arrived at that percentage.

Despite the frequent fabrications, there is an underlying truth here: Most mobile app developers leave testing until the end of the project. Testing is how you find out how well a project is really going. If a developer has written code for half the features on the project, the project may be as much as half done. But if that set of features turns out to have tons of bugs, then there are tons of bug fixes that must be written before the project is half done. And before testing, there's no way to know which.

What happens is that developers write all the code to implement all the features before turning over the code to QA for testing, and they assume that testing is just a formality and won't turn up very much. They assume,

for example, that the testing at the end will be 10% or so of the project time. When testing turns up a whole bunch of stuff, they stick to their guns and think (or at least say) that they're still 90% done. This is where we get the joke that "the first 90% of the project takes 90% of the time, and the last 10% of the project takes the other 90% of the time."

Avoiding this situation requires testing early, testing often, and testing the right things.

First, understand that early testing doesn't reduce risk; it just reduces uncertainty, so it's still important not to leave the riskiest parts of the project until the end. (See the "Leaving the Worst for Last" section earlier in this chapter.) Sequencing of features is still critical, even with lots of testing.

Second, each feature or component needs to be tested to make sure it's functioning correctly. Note that it needs to be tested to make sure it behaves well when expected things happen and that it doesn't behave badly when obvious but unexpected things happen. Most developers do at least an adequate job of this.

Finally, testing needs to verify that when each new feature is added, all the previous features still work. This is called *regression testing*, and lots of developers don't do a good job of it.

For more about testing, see Chapter 12, "Communicating Using Bugs," and Chapter 13, "Testing."

The Whack-a-Mole Problem

There's a specific situation that causes cost overruns and stretches out the last 10% of a project. It sometimes happens earlier in the project, but most often it shows up at the end. I call it the *whack-a-mole problem*, after the carnival game where a player with a hammer tries to hit mechanical creatures as they pop up out of holes.

The whack-a-mole problem occurs when it seems that the fix for any given bug causes a new bug to pop up. This is often an indication of a poor architecture (or poor developers). Unfortunately, if you've reached whack-a-mole, it's usually too late to change out the architecture (or the development team) without a lot of effort. There are many ways to end up in this situation, but let me take you through a common scenario.

Mobile apps are largely driven by *events*—things like the user pressing a button or new information being received from the network. Somewhere in the source code, a set of instructions gets called when a particular button gets

pressed. When that set of instructions becomes too complicated, you get the whack-a-mole problem.

Imagine a banking app with a button that says Account History on the main screen. The user taps the button, and the account history screen appears. Everything works fine.

Then someone asks, “What should that button do if the user isn’t logged in?” So the programmer is tasked with writing code so that if the user is logged in, that button shows the account history screen but otherwise shows the login screen. Then the login screen gets programmed to return to one place if it was shown from the login button and another place if it was called by the Account History button.

Then later in the project, someone says, “We shouldn’t open the login screen if the network isn’t available since the user won’t be able to log in; that would be confusing.” Now the programmer goes through the code and finds all the places that the login screen is called and puts a check at each one to see if the network is available, and if it isn’t, it shows the screen that says the network isn’t connected and asks the user to connect to Wi-Fi.

At this point, the code that is executed when the button is pressed depends on two states: whether the user is logged in and whether the network is available. The programmer isn’t thinking about all the different possibilities, only the case where the user isn’t logged in and the case where the network isn’t available. The programmer makes the change to show the screen that asks the user to connect to Wi-Fi and checks in the change.

If the programmer wasn’t careful, when the user taps the Account History button when the network isn’t available, the app will ask the user to connect to Wi-Fi, even if the user was already logged in, which wasn’t part of the requirement.

As time passes, more and more states can get thrown into the mix. There could be a special screen that needs to be shown on login when the users have overdrawn their accounts. There could be a special screen that’s shown on the user’s birthday. There could be an alert that needs to be shown when fraud has been detected on the account. Maybe an interstitial ad needs to be shown, and so on and so on. Each time a new set of behaviors is added to that button, all the previous behaviors are at risk of becoming broken, and the amount of time it takes to do a thorough test gets longer and longer because each combination of states needs to be tested. Multiply that by the number of buttons in the app, and you see how big the whack-a-mole problem can be.

Some application architectures and programming techniques do a good job of handling this complexity (assuming that the programmer is experienced with such techniques). The trick is that they have to be put in place at the right time. If they're put in place too early, they are more trouble than they are worth, but if the programmer waits too long, it becomes tedious and time-consuming to move all the existing behavior to the new architecture.

Poor Communication

Poor communication dooms any software project, but mobile projects, with their relatively smaller budgets, shorter time lines, and often remote teams are particularly susceptible. The solution, though, is not more meetings. I've been on projects that had conference calls every day, and the communication was still horrible. In fact, lots of meetings often make it worse because everyone thinks that if nothing came up on the conference call, everything must be going fine, when that's not necessarily the case at all.

It's in vogue these days for every project to have a daily stand-up meeting. I'm not going to tell you not to have one; there's nothing inherently wrong with those meetings. What I *am* going to tell you is that a daily stand-up doesn't give you information about *how well* the work is being done or whether the whole project is on track. A daily stand-up just tells you what everyone is working on that day, and what, if anything, they're waiting on to be able to complete their current tasks.

And then there are project status meetings where everyone says that his or her part of the project is going just fine. Again, these aren't particularly useful.

Making the Most of Meetings

As with requirements, good communication is all about details. And to prompt your developers to give you good details, you have to ask the right questions. The questions usually asked at meetings are either two narrow ("What are *you* doing *today*?") or too vague ("Is *your part* of the project *going okay*?").

One recommendation is that you should never have a project meeting without a written agenda, and the agenda should be circulated to everyone who will attend the meeting in advance to give people time to think about what they need to say.

The agenda should ask the questions that you as the app creator need answered, like "What's the biggest risk to this project as you see it right now?" or "What's the most likely thing that you think could go wrong on this

project, and what can be done about it?" or even "What decisions need to be made in order to complete the tasks you are currently working on, and what alternatives are there for each decision?" These kinds of questions will prompt discussion of the details that you need to know.

Then, once the meeting is over, make sure that someone writes up a summary of what happened in the meeting and distributes it to the group. These meeting notes become very useful later in the project when confusion arises.

Examine the Project's Artifacts and Ask Questions Contemporaneously

Another way to facilitate communication is to look at what's happening on a project and ask questions about it. I try to review changes to the source control repository and bug trackers at least every day or two during a project to keep up to date with what's going on. (See Chapter 10, "Understanding What You're Getting," for how to do this.) If I have any questions after reading the commit messages on the code check-ins or the updates to the bugs, I send someone email and ask for clarification. If the developers know that they're going to be contacted if they don't write clear commit messages or bug updates, they eventually put more thought into what they write, and the quality of that information goes way up. (There are many discussions of source control and bug trackers throughout this book, so don't worry if these terms are unfamiliar right now.)

It's important to ask these questions soon after the developer does the work. If you wait a week or two (or maybe even a few days), the developer might not remember as well what he or she was doing, and the quality of the explanation will suffer.

Insist on Getting a Plan

Periodically (at least weekly), you need to insist on getting a plan that shows what needs to be done between now and the end of the project, with specific details about what the next few steps are. Much of the report will be the same from week to week, but as you get more and more of these reports, you should see how the things that have been done have lined up with what the plan had previously said was going to happen. As with everything else, make sure you understand what you are looking at and ask detailed questions until you do.

If at all possible, I recommend getting these plans in the form of Gantt charts (which are discussed in Chapter 5, "Finding the Right Tools," and Chapter 9, "Managing to Milestones").

Abdication of the Management Process

Many of the issues discussed in this chapter boil down to one root cause: The app creator didn't pay enough attention to managing the project or left the management of the project up to someone else (who may or may not have done it at all). Whether you are doing the programming yourself or can't write a line of code, if you want your app idea to come to fruition, you have to keep an honest and vigilant perspective on where you are in the project and make good decisions based on that knowledge.

Many app development companies provide a professional project manager to do this for you (and charge a premium for the privilege), but project managers don't have the same incentives you do, and their loyalty is not to you or your project but to their employer. Often, those project managers don't want to ask hard questions. They don't want to admit that failure is an option. They are generally incentivized primarily to manage your expectations and get you to pay your invoices.

You, as an app creator, are responsible for understanding where a project is now and where it is going, what risks are involved and what can be done about them, and whether your developer is doing a good job or needs to be replaced. If you don't feel comfortable about that responsibility, welcome to the club. This is difficult stuff to do, and anyone who says differently is selling something.

The simple fact remains that there's no one else you can trust to do it. You are the one selecting the developer, and you are the one controlling the purse strings. It's your vision and your goal. But despite the difficulty, it is possible. It can be done, and this book is here to help you do it.

Wrapping Up

This chapter discusses the different ways that app projects fail and provides some tips about what can be done about it. Here are some key points to take away:

- Despite the relatively small size of app development projects, they are still software development projects, and many of them fail.
- App projects can fail by taking too long or costing too much, even if the app produced at the end is acceptable.
- A number of factors make mobile development more difficult than web development, but inexperienced programmers don't always recognize that.

- Even experienced development companies with good reputations can have failing projects.
- Good requirements are key to getting a good estimate, a good project, and a good app. And they don't just have to be defined well but also have to be communicated well.
- To the extent possible, uncertainty and risk need to be pushed early into the project schedule to avoid catastrophic failure at the end.
- Testing should never be left until the end of a project.
- You can prevent project failure. It's difficult, but this book is here to help.

Index

A

abandoned third-party components, 115-116
accessibility features, 122
accessing
  source code, 253-254
  stored data, 127-130
accounts
  app stores, 151
  providing to app reviewers, 332
accuracy of estimates, 239
  bug fixes, 245-246
  familiarity, 239-240
  granularity, 243
  isolation, 241-242
  padding, 242-243
  renovation versus construction, 243-245
  tasks included, 243
  uncertainty, 240-241
ad agencies, 194-196
adapting screens for tablets, 67
advertising services, 154
agenda for meetings, 29
analytics, 119, 320
  services, 119-120, 153-154
  what to collect, 120
animations
  custom/complex, 122-123
  in design phase, 38
  in user experience design, 171
API (application programming interface), 43
Apollo I fire, 306
app builder tools, 187, 189
app creators
  source control usage, 142-143
  stealing app ideas, 179-180
app development
  companies, 196-199
  interactive prototypes. *See* interactive prototypes
  life cycle, 33-34
    design phase, 34-39
    development phase, 39-45
    repeating, 49-51
    testing phase, 45-49
  project failure. *See* project failure
  prototyping. *See* prototyping
  wireframing. *See* wireframing
app programming. *See* app development; mobile development
app projects. *See* projects
app reviews. *See* reviews
app stores
  accounts, 151
  discoverability, 110-111
  number of apps in Apple iOS App Store, 337
  rejections. *See* rejections
  resubmission, 340
  review requirements, 329
  reviewer instructions, writing, 331-332
  submission process, 334-335
Apple, 50
  iOS App Store, number of apps in, 337
  WWDC, 336-337
application as a service, 152-153
application programming interface (API), 43
apps
  avoiding last-minute changes, 333-334
  building from source code, 258-260
  components. *See* components

feedback after app release, 340
crash reports, 340
infrastructure for, 342
reviews, 341-342
HTML5 apps, defined, 104
hybrid apps
comparison with HTML5 and native apps, 105-110
corner cases, 118-119
defined, 105
game engines, 117-118
third-party frameworks, 111-117
icon design, 175
launching, 340
marketing materials, creating, 330-331
mobile web apps, defined, 104
native apps, defined, 104
new releases, 342-343
publishing to testers, 310-311
releasing after review, 335
submitting. *See* submitting apps
websites as, 110-111
archiving bug reports, 304
areas of attention (in release notes), 319
as designed (bug state), 301
asking questions, 30
assigned (bug state), 300
attaching files to bug reports, 298-299
audio components, 120
Auto Layout, 23
automated testing, 48-49, 305, 314-315
availability of developer, problems with, 277

B

Back buttons, prototyping, 92
back end, front end versus, 43-44
background tasks, 134-135, 267
Balsamiq, 61, 79
barebones wireframing style, 62-63
being verified (bug state), 301
beta release cycles, 321-322
beta testers, 315-319
distribution tools, 159-160
diversity of, 317-318
expectations for, 318-319
finding, 316-317
bikeshedding, 16, 62
billable hours, 11, 25
blocked (bug state), 300
blogs, creating, 179
Blub Paradox, 201
Bluetooth, 121-122
boilerplate code, 265
books for improving programming skills, 166
Borman, Frank, 306
branching, 70, 141
Brooks, Fred, 24, 50, 235
Brooks's law, 235
budgeting, 40-42
considering in app development, 49
constraints, 250-251
cost overruns, reasons for, 24-26
bug reports
archiving, 304
attaching files to, 298-299
described, 291-292
how to write, 296-298
language usage in, 292-294
merging, 303
number of bugs in, 290-291
as placeholders, 294-295
reopening versus creating new, 301-302
splitting, 303
states, 300-301
writing, 146-147
bug trackers, 144-145
bug reports. *See* bug reports
bugs versus features, 147
as business continuity, 295
code comments versus, 295-296
as communication tools, 288-290
defined, 287
in feedback structure, 321
selecting, 145-146
when to use, 145
bugs, 144. *See also* testing phase
accuracy of estimates, effect on, 245-246
after app release, developer support, 224
as bug-report category, 291
cost overruns, 25
data-specific bugs, 299
defined, 287-288
as failures of imagination, 306-307
features versus, 147, 292-294
in hybrid frameworks, 108
in milestones, 272-274
number in bug reports, 290-291

- platform bugs, 279
- prioritizing, 323-326
- as rejection reasons, 338
- reproducing, 299-300
- testing, case study, 312
- in third-party frameworks, 112
- triage, 323
- verifying, 326
- whack-a-mole problem, 27-29

bug-tracking tools. *See* bug trackers
build verification tests, 157
building apps from source code, 258-260
built-in art assets, benefits of, 173-174
business continuity, bug trackers as, 295
buttons, gestures versus, 75

C

Cancel buttons, prototyping, 92
capabilities of devices, supporting, 103
capturing
- app-usage videos, 332
- screenshots, 298

Carroll, Lewis, 343
case studies
- API documentation, 19
- app development company outsourcing, 198-199
- Auto Layout UI code, 23
- cookie refreshing, 312
- design changes, 51
- Groovy and Grails languages, 114
- miscommunication with developer, 276
- missing source code, 157
- multiple bug reports, 302-303
- number comparison bug, 325
- optimization updates, 333-334
- outsourcing developers, 210
- plagiarism detection, 263-264
- regression testing, 46
- rejecting previously approved apps, 338
- resource sharing, 168
- signup wizards, 78
- spaghetti code, 14-15
- sprints, 231
- thread usage, 9-10
- vague requirements, 17-18
- WWDC (Worldwide Developers Conference), 336-337

centralized source control, 140
change requests
- avoiding last-minute changes, 333-334
- cost overruns, 25
- project failure reasons, 20

changing developers
- compensation for old developer, 284
- transition to new developer, 284-285

check-ins. *See* commits (source code)
CI (continuous integration) environment, 157-159, 310
classes for improving programming skills, 166
closed (bug state), 301
closed as duplicate (bug state), 301
Cocos2D, 118
code base. *See* source code
code comments, bug trackers versus, 295-296
code coverage, 262
code signing, 335
collaborative bug tracking, 289-290
collecting information. *See* analytics
comments
- in source code, 256-258
  - *bug trackers versus, 295-296*
  - *commented out code, 265*
- in source control, 254-256

commercials, writing, 55
commits (source code)
- comments in, 254-256
- defined, 254

communication
- budget constraints, 250-251
- Conway's law, 235-237
- expectations, 223
- importance of, 228
- local versus remote developers, 191-193
- miscommunication with developer, 275-276
- mistakes
  - *lack of coordination, 281*
  - *wasted effort, 280-281*
- proactive developer communication, 274-275
- project failure reasons, 29-30
- with reviewers, 336-337
- time constraints, 251-253
- uncertainty in schedule estimation, 240

communication tools, bug trackers as, 288
  collaborative tracking, 289-290
  comprehensive tracking, 289
  notifications, 290
compensation for developers, 284
competition, considering in app development, 49
compiler warnings, 264
complexity of apps, programming skills needed, 164-165
components
  accessibility, 122
  analytics, 119
    services, 119-120
    what to collect, 120
  audio and video, 120
  background tasks, 134-135
  conditional text formatting, 123-124
  custom/complex animations, 122-123
  data storage, 125-131
  defined, 97
  device support, 97
    form factors, selecting, 100
    newer software, 102-103
    older hardware, 101
    older software, 101-102
    platforms, selecting, 98-100
    rotation, 100-101
    sensors, 103
  HTML5 apps, defined, 104
  hybrid apps, 105-110
  localization, 124-125
  mobile web apps, defined, 104
  native apps, defined, 104
  peripherals, 121-122
  push notifications, 134
  servers, 131-133
  synchronization, 133-134
  third-party components, 164
    source control usage, 143
  third-party frameworks. *See* third-party frameworks
  third-party libraries, 260-261
  user preferences, 125
  websites as apps, 110-111
comprehensive bug tracking, 289
computer programming. *See* programming
conditional text formatting, 123-124
conferences for improving programming skills, 167
conflicts in synchronization, 134
consolidating screens, 73-74
Consumer Reports, 50
contingency plans, interviewing developers, 213-214
continuous integration (CI) environment, 157-159, 310
converting wireframes for interactive prototypes, 82-86
Conway's law, 235-237
cookie refreshing case study, 312
coordination, lack of, 281
copying
  screens, 65-66
  user experience design elements, 172
copyright misuse rejections, 338
copywriting, skills needed, 178-179
Core Data framework, 128
core experience, focus on, 54-58
corner cases, third-party frameworks, 118-119
cosmetic bugs, 325-326
cost estimates
  interviewing developers, 217-223
    fixed-price projects, 218-219
    hourly fees, 219-220
    invoice timing, 221-223
    milestone-based fees, 220-221
    revenue sharing, 218
  project failure reasons, 3
cost overruns, reasons for, 24-26
couldn't verify claim rejections, 339
crash bugs, 323
crash reports, 160-161, 320
  after app release, 340
crash/debugging rejections, 338
creative act, programming as, 167
creative agencies, 194-196
cross-platform applications, 105
custom animations, 122-123
customers, platform selection, 99

D

data consistency bugs, 324-325
data loss bugs, 324
data storage, 125-131
  access methods, 127-130
  amount needed, 126-127
  frequency of changes, 131
  performance, 131

security, 130
services, 152
types of information to store, 125-126
data-specific bugs, 299
deadlines, scheduling, 311
debugging. *See* bugs
defects, 144. *See also* bugs
deferred (bug state), 301
dependencies, 144
identifying, 233
unforeseen, 279
design, skills needed
game design, 182
graphic design, 173-176
sound design, 176-178
user experience design, 169-173
design elements, expense of, 44
design phase (app development), 34
case study, 51
deadline for, 311
early decisions, benefits of, 36-37
error handling, 37-38
first-run conditions, 38
initial steps, 35
interactive prototypes. *See* interactive prototypes
prototyping. *See* prototyping
scrolling and element obstructions, 38
text length assumptions, 39
transitions and animations, 38
wireframing. *See* wireframing
design testing, 45
designers, developers as, 41
developers
asking questions, 30
availability issues, 277
buggy milestones, 272
performance failures, 273-274
regression failures, 272-273
changing
compensation for old developer, 284
transition to new developer, 284-285
communication. *See* communication
cost overruns, 24
as designers, 41
discussing project failure, 271
dishonesty from, 275
evaluating, 42
experienced firms, problems with, 10-12
finding, 185
app development companies, 196-198
creative agencies, 194-196
independent developers, 199-203
local versus remote developers, 191-193
matchmaker sites, 189-191
template app sites, 185-189
training new developers, 203-204
hiring, 208
interviewing, 207-208
communication expectations, 223
contingency plans, 213-214
estimating and planning skills, 214-217
financial arrangements, 217-223
gap analysis, 212-213
previous work experience, 210-212
setting up interviews, 208, 210
support after app release, 224
technical support, 223-224
tool usage, 223
miscommunication with, 275-276
outsourcing case study, 210
poor skill set fit, 8-10, 201
poor task estimation, 277-282
proactive communication, 274-275
recovery plans for missed milestones, 270-271
as testers, 313-315
third-party framework experience, 113
trust, establishing, 137
unforeseen circumstances, 25
development environment, 154
IDEs (integrated development environments), 155
importance of, 156
simulators/emulators, 155-156
development phase (app development), 39
deadline for, 311
feature sequencing, 42-45
features, 40
scheduling, planning, budgeting, 40-42
screens, 39-40
device mismatch (performance failures), 273
device support, 97
form factors, selecting, 100
newer software, 102-103
older hardware, 101
older software, 101-102
platforms, selecting, 98-100

rotation, 100-101
sensors, 103
direct database queries, 128
discoverability in app stores, 110-111
dishonest developers, 275
distributed source control, 140
document format apps, 129
documentation
as contingency plans, 213-214
out-of-date documentation, 18-19
poorly defined requirements, 16-18
system of record, maintaining, 80-82
for third-party frameworks, 116
drag-this-and-drop-that apps, 187, 189
drawing tools for wireframing, 61-62
duplicating
code, 264-265
effort, 280-281
screens in interactive prototypes, 89-91
dynamic screens, wireframing, 64-65

E

early builds, testing, 159
early design decisions, benefits of, 36-37
element obstruction in design phase, 38
elevator pitches, 55
emulators, 155-156
end-user feedback, 161-162, 321
enhancements
as bug-report category, 291
bugs versus, 292-294
epiphany, scheduling for, 308-309
error functions, wireframing, 69-70
error handling in design phase, 37-38
errors
cost overruns, 25
whack-a-mole problem, 27-29
estimates
accuracy of, 239
bug fixes, 245-246
familiarity, 239-240
granularity, 243
isolation, 241-242
padding, 242-243
renovation versus construction, 243-245
tasks included, 243
uncertainty, 240-241
cost overruns, 26
poor estimation issues, 277-282
project failure reasons, 3
reevaluating, 246
skills in, interviewing developers, 214-217
evaluating
developers, 42
project failure. *See* project failure, evaluating
recovery plans, 277-282
events, 27
examples. *See* case studies
exception testing, defined, 306
execution, ideas versus, 12-15
exit strategy, 98
expectation mismatch (performance failures), 273-274
expectations for beta testers, 318-319
expensive design features, whether to use, 44
experience in programming (skill gap analysis), 164-165
experienced developer firms, problems with, 10-12
external dependencies, unforeseen, 279
external services, 150
advertising, 154
analytics, 153-154
app store accounts, 151
data storage, 152
in-app purchasing, 154
platform/software/application as a service, 152-153
push notifications, 152
social media, 151
external testers, 315-319

F

Facebook, 92, 151, 172
failures of imagination, 306-307. *See also* project failure
familiarity (estimate accuracy), 239-240
features
as bug-report category, 291
bugs versus, 147, 292-294
defined, 40
determining necessary features, 50-51
of devices, supporting, 103
sequencing, 42-45

feedback, 161-162, 319
after app release, 340
crash reports, 340
infrastructure for, 342
reviews, 341-342
beta release cycles, 321-322
from beta testers, 318-319
bug triage, 323
from end users, 161-162
infrastructure for, 320-321
prioritizing bugs, 323-326
in prototyping, 94
in schedule management, 246
verifying bugs, 326
on wireframes, 72-73
file system mismanagement rejections, 338
files, attaching to bug reports, 298-299
fill-out-this-form apps, 187
financial arrangements, interviewing developers, 217-223
fixed-price projects, 218-219
hourly fees, 219-220
invoice timing, 221, 223
milestone-based fees, 220-221
revenue sharing, 218
finding. *See also* hiring developers
beta testers, 316-317
developers, 185
app development companies, 196-198
creative agencies, 194-196
independent developers, 199-203
local versus remote developers, 191-193
matchmaker sites, 189-191
template app sites, 185-189
training new developers, 203-204
graphic designers, 175-176
sound designers, 178
user experience designers, 172
first-run conditions
in design phase, 38
wireframing, 70-71
fixed (bug state), 301
fixed-price project work, 218-219
focus, importance of, 234-235, 278
forced-failure testing, 313-314
forks, 115
form factors, selecting, 100
formatting text, 123-124
forms, building websites from, 187
FOSS (free or open-source software). *See* third-party frameworks
fractured third-party components, 115-116
frame rate, 273
frequency of milestones, 228-230
front end, back end versus, 43-44
full text search, 129
functionality
defining, 3
options versus, 43

G

game building, skills needed, 181-183
game design, skills needed, 182
game engines, 117-118
programming skills needed, 181
Gantt charts, 149-150
gap analysis. *See* skill gap analysis
general testing, defined, 305
gestures, buttons versus, 75
git (source control system), 141
GitHub, 112-113, 146
golden path, 54, 69
testing, 47
Google, interviewing developers, 207
Graham, Paul, 252
Grails, 114
granularity of estimates, 243
graphic design
finding professional designers, 175-176
skills needed, 173-176
app icon design, 175
built-in art assets in platforms, 173-174
games, 182-183
stock photos, 174
vector icon packages, 174
Groovy, 114
grouping screen data, 74-75
GUI tools for source control, 142

H

hardware
adapting to new features for third-party frameworks, 116
peripherals, 121-122
providing to app reviewers, 332
supporting older hardware, 101

hg (source control system), 141
high-risk items at end of project (project failure reasons), 20-23
hiring developers, 208. *See also* finding
  interviewing, 207-208
  NDAs (nondisclosure agreements), 208
Hofstadter's law, 242
hourly fees, 219-220
HTML5 apps
  comparison with native and hybrid apps, 105-110
  defined, 104
hung, defined, 9
hybrid apps
  comparison with HTML5 and native apps, 105-110
  defined, 105
  third-party frameworks. *See* third-party frameworks

I

icons
  app icon design, 175
  vector icon packages, 174
ideas, execution versus, 12-15
IDEs (integrated development environments), 155
if statement complexity, 266
improvements
  as bug-report category, 291
  bugs versus, 292-294
  scheduling for, 308-309
improving programming skills, 165-167
in test (bug state), 301
in-app purchasing services, 154
in-progress (bug state), 300
independent developers, 199-203
inexperienced developers, 10
information collection. *See* analytics
integrated development environments (IDEs), 155
interactive prototypes, 76-77
  Back and Cancel buttons, 92
  benefits of, 77-78
  converting wireframes for, 82-86
  duplicating screens, 89-91
  linking screens, 86-89
  tools for creating, 78
  wireframes versus, 80-82
interactive wireframing style, 63
internationalization, 124-125
interviewing developers, 207-208
  communication expectations, 223
  contingency plans, 213-214
  estimating and planning skills, 214-217
  financial arrangements, 217-223
  gap analysis, 212-213
  previous work experience, 210-212
  setting up interviews, 208, 210
  support after app release, 224
  technical support, 223-224
  tool usage, 223
investors, platform selection, 98
invoices, submitting, 221, 223
iOS threads, 8
isolation (estimate accuracy), 241-242
issue trackers. *See* bug trackers
issues, 144
iteration, defined, 59

J–K

Jobs, Steve, 34, 51

kanban boards, 149-150
keyboards
  Bluetooth, 122
  prototyping, 93
Keynote, 79-80
known issues (in release notes), 319

L

language usage in bug reports, 290-294
last-minute changes, avoiding, 333-334
launching apps, 340
learning
  copywriting, 179
  marketing, 180
  programming skills, 165-167
licensing third-party frameworks, 116
linking screens in interactive prototypes, 86-89
local developers, remote developers versus, 191-193
localization, 124-125
locked-up, defined, 9

log/console uploading, 320
logged out workflow, prototyping, 93-94
login screens in first-run workflow, 71

M

magic numbers, 265-266
main threads, 267
"Maker's Schedule, Manager's Schedule" (Graham), 252
mapped objects, 128
marketing, skills needed, 179-181
marketing materials, creating, 330-331
matchmaker sites (finding developers), 189-191
measuring
  development, 39-40
  progress, 228-230
meetings
  communication in, 29-30
  expectations for, 223
  time constraints and, 251-252
meet-up groups for improving programming skills, 167
Mercurial, 141
merging, 141
  bug reports, 303
metadata
  for app marketing, creating, 330-331
  rejections, 339
methods, defined, 262
micromanaging, 15
milestone-based pricing, 220-221
milestones
  bugs in, 272-274
  focus, 234-235, 278
  frequency of, 228-230
  missed. *See* missed milestones
  organization with, 232-233
  sequencing, 233-234, 277
  sprints versus, 230-231
miscommunication with developer, 275-276
missed milestones, 270
  developer availability, 277
  evaluating recovery plans, 277
  poor task estimation, 277-282
  proactive developer communication, 274-275
  recovery plans for, 270-271
  timing of, 282-283
missing source code case study, 157
mistakes. *See also* bugs
  lack of coordination, 281
  wasted effort, 280-281
mobile development, web development versus, 5-8. *See also* app development
mobile device support. *See* device support
mobile platforms. *See* platforms
mobile web apps, defined, 104. *See also* HTM5 apps
mockups. *See* wireframing
monetization, 181
  of apps versus websites, 111
multimedia components, 120
multiple bug reports case study, 302-303
multiple platforms, building on, 99-100
multitasking, 134-135
music
  in games, 182-183
  skills needed, 176-178
The Mythical Man-Month (Brooks), 24, 50, 235

N

naming conventions, 266-267
native apps
  comparison with HTML5 and hybrid apps, 105-110
  defined, 104
navigation bugs, 323-324
NDAs (nondisclosure agreements), 208, 316
  case study, 168
necessary features, determining, 50-51
negative testing, 47
newer software, supporting, 102-103
niche bugs, 326
nondisclosure agreements (NDAs), 208, 316
  case study, 168
notifications
  from bug trackers, 290
  push notifications, 134, 152
number comparison bug case study, 325

O

object relational mapper (ORM), 128
obstruction on screen in design phase, 38
offline mode failure rejections, 338
older hardware, supporting, 101
older software, supporting, 101-102

OmniGraffle, 61
open (bug state), 300
OpenGL, 117-118
optimization case study, 333-334
options, functionality versus, 43
organizing
  with milestones, 232-233
  screens, 73-74
  source code files, 261-262
ORM (object relational mapper), 128
out-of-date requirements documentation, 18-19
outsourcing case study, 198-199, 210
overhead items, 11, 279-280

P

PaaS (platform-as-a-service), 133, 152-153
package-your-website apps, 187
padding estimates, 242-243
PDF viewers, 79
performance
  data storage and retrieval, 131
  hybrid apps, 106
performance failures, 273-274, 325
performance testing case study, 23
peripherals, 121
  Bluetooth, 121-122
  Wi-Fi, 122
permutation testing, defined, 306
phone screens in tablets, 75-76
photorealistic wireframing style, 63
photos, stock, 174
  in games, 183
Pitch Perfect (Sadun and Sande), 180, 330
placeholder bug reports, 294-295
plagiarism, detecting, 262-264
planning, 40-42
  importance of, 30
  as mandatory, 42
  skills, interviewing developers, 214-217
platform bugs, 279
platform-as-a-service (PaaS), 133, 152-153
platforms
  built-in art assets, benefits of, 173-174
  selecting, 98-100
popularity versus quality in third-party frameworks, 112-113
PowerPoint, 79-80
preferences, 125
preprogrammed components. *See* third-party components
previous work experience, interviewing developers, 210-212
pricing, 181
  interviewing developers, 217-223
    fixed-price projects, 218-219
    hourly fees, 219-220
    invoice timing, 221, 223
    milestone-based fees, 220-221
    revenue sharing, 218
prioritizing bugs, 323-326
private API use rejections, 339
proactive developer communication, 274-275
procrastination, 148
  avoiding, 227-228
production design, 176
productivity of billable time, 25
professional testers, 311-313
profitability of projects, 4-5, 201-202
programming
  apps, 5-8
  as creative act, 167
  skills needed, 163-164
    app complexity, 164-165
    experience, 164-165
    game programming, 181
    improving your skills, 165-167
    third-party components, 164
progress, measuring, 228-230
project failure
  changing developers
    compensation for old developer, 284
    transition to new developer, 284-285
  defined, 2-3
  discussing with developer, 271
  evaluating, 269-270
    buggy milestones, 272-274
    dishonest developers, 275
    miscommunication with developer, 275-276
    missed milestones, 270
    proactive developer communication, 274-275
    recovery plans, 270-271
    timing of missed milestones, 282-283
  percentage of, 1-2

reasons for
bikeshedding, 16
communication problems, 29-30
constantly changing requirements, 20
cost overruns, 24-26
difficulty of app programming, 5-8
high-risk items done last, 20-23
ideas versus execution, 12-14
micromanaging, 15
out-of-date requirements documentation, 18-19
poor developer skill set fit, 8-9, 201
poorly defined requirements, 16-17
problems with experienced developer firms, 10-12
project management problems, 31
scope of projects, 4
testing at end of project only, 26-27
time and cost estimates, 3
unprofitability, 4-5, 201-202
whack-a-mole problem, 27-29
recovery plans. *See* recovery plans
salvaging projects, 283-284
sunk cost fallacy, 269
project fees, 218-219
project management
automated testing, 262
budget constraints, 250-251
communication, importance of, 228
Conway's law, 235-237
effort versus project size, 253
estimates, accuracy of. *See* estimates, accuracy of
plagiarism, detecting, 262-264
procrastination, 227-228
project failure reasons, 31
reevaluating schedules, 246
risk mitigation
focus, 234-235, 278
frequent milestones, 228-230
milestones versus sprints, 230-231
organization with milestones, 232-233
sequencing milestones, 233-234, 277
scheduling software, importance of, 237-239
skills, interviewing developers, 216-217
source code. *See* source code
time constraints, 251-253
tools, 148-150
project plans, importance of, 30. *See also* planning
projects
defined, 261
functionality, defining, 3
salvaging, 283-284
scope, 4
prototype testing, 46
prototyping. *See also* interactive prototypes; wireframing
Back and Cancel buttons, 92
benefits of, 53-54
core experience, focus on, 54-58
logged out workflow, 93-94
screen titles, 92
user feedback, 94
virtual keyboard, 93
publishing apps to testers, 310-311
purpose of apps, determining, 35
push notifications, 134, 152

Q–R

QA (quality assurance)
departments, 311-313
skills needed, 168
quality versus popularity in third-party frameworks, 112-113
questions, asking, 30

Reachability (Apple library), 338
recovery plans
evaluating, 277
developer availability, 277
poor task estimation, 277-282
for missed milestones, 270-271
reevaluating schedules, 246
refactoring, 262, 265
refining screens, 73-74
Reflector, 332
regression failures, 272-273
regression testing, 27, 46, 158
case study, 46
defined, 306
rejections, 335
communicating with reviewers, 336-337
copyright misuse rejection, 338
couldn't verify claim rejection, 339
crash/debugging rejection, 338

file system mismanagement rejection, 338
metadata rejection, 339
offline mode failure rejection, 338
of previously approved apps, 338
private API use rejection, 339
relational data, 127
release notes, 319
releasing apps, 340
after review, 335
developer support after, 224
remote developers, local developers versus, 191-193
renovating code bases (accuracy of estimates), 243-245
reopened (bug state), 301
reopening bugs, creating new versus, 301-302
repeating app development life cycle, 49
budget considerations, 49
competition considerations, 49
necessary features, determining, 50-51
schedule considerations, 50
user testing, 50
reports
bug reports. *See* bug reports
crash reports, 160-161, 320
after app release, 340
status reports, 228
expectations for, 223
reproducing bugs, 299-300
reproduction steps in bug reports, 296
requirements
constantly changing, 20, 25
out-of-date documentation, 18-19
poorly defined requirements, 16-18
resolved (bug state), 301
resource juggling, 11
resource sharing case study, 168
REST (representational state transfer), 132
resubmitting apps, 340
return on investment (ROI) of apps versus websites, 111
revenue sharing, 218
reviewers
case study, 336-337
communicating with, 336
instructions, writing, 331-332
reviews
after app release, 341-342
app store requirements, 329
releasing apps after, 335
revision control. *See* source control
rewritten third-party components, 115-116
risk mitigation. *See also* contingency plans
focus, 234-235, 278
frequent milestones, 228-230
milestones versus sprints, 230-231
organization with milestones, 232-233
project failure reasons, 20-23
sequencing milestones, 233-234, 277
third-party frameworks, 117
"Roach Motel" bugs, 323-324
rock fetching, 13
ROI (return on investment) of apps versus websites, 111
rotation, supporting, 100-101
RubyMotion, 118

S

SaaS (software as a service), 152-153
Sadun, Erica, 180, 330
salvaging projects, 283-284
Sande, Steve, 180, 330
saving bug reports, 304
schedule management
elements needed for, 232
estimates, accuracy of. *See* estimates, accuracy of
reevaluating schedules, 246
sequencing milestones, 233-234, 277
time constraints, 251-253
schedule tracking tools
benefits of, 148-149
Gantt charts, 149-150
importance of, 237-239
kanban boards, 149-150
scheduling, 40-42
considering in app development, 50
deadlines, 311
testing, 308-311
for epiphany, 308-309
publishing apps for testers, 310-311
sequencing problems, 309
user-facing functionality, 308
scope of projects, 4
screen navigation bugs, 323-324
screens
adapting for tablets, 67
consolidating and refining, 73-74
copying, 65-66

in development phase, 39-40
duplicating in interactive prototypes, 89-91
dynamic screens, wireframing, 64-65
grouping like data, 74-75
linking in interactive prototypes, 86-89
login screens in first-run workflow, 71
phone screens in tablets, 75-76
simplified scrolling, 75
sketching, 59-61
titles, prototyping, 92
screenshots, capturing, 298
scripts in use case development, 58
scrolling
in design phase, 38
simplifying, 75
SDKs (software development kits), 133
searching with full text search, 129
security, data storage, 130
selecting. *See also* finding; hiring developers
bug trackers, 145-146
form factors, 100
platforms, 98-100
source control system, 142
tools based on project size, 138
self-reporting capable attributes (use cases), 56
sensors, supporting, 103
sequencing
features, 42-45
milestones, 233-234, 277
in testing schedule, 309
servers, 131-132
estimation accuracy, 241
PaaS (platform-as-a-service), 133
SDKs (software development kits), 133
for source control, 142
support and troubleshooting skills needed, 168-169
writing your own, 132-133
services
analytics, 119-120
external services. *See* external services
marketing, 180
settings, user preferences, 125
signup wizards case study, 78
simulators, 155-156
situations in use case development, 57
sketching screens, 59-61
skill gap analysis
copywriting, 178-179
game building, 181-183
graphic design, 173-176
app icon design, 175
built-in art assets in platforms, 173-174
finding professional designers, 175-176
games, 182-183
stock photos, 174
vector icon packages, 174
interviewing developers, 212-213
marketing, 179-181
programming, 163-164
app complexity, 164-165
as creative act, 167
experience, 164-165
game programming, 181
improving your skills, 165-167
third-party components, 164
server support and troubleshooting, 168-169
sound design, 176-178
games, 182-183
testing and quality assurance (QA), 168
user experience design, 169-173
skinning, defined, 44
slicing, 63
social media services, 133, 151
soft landing problem, 38
software
adapting to new features for third-party frameworks, 116
newer software, supporting, 102-103
older software, supporting, 101-102
software as a service (SaaS), 152-153
software development kits (SDKs), 133
sound design, skills needed, 176-178
games, 182-183
source code
accessing, 253-254
automated testing, 262
building app from, 258-260
missing source code case study, 157
organizing files, 261-262
plagiarism, detecting, 262-264
third-party libraries, 260-261
understanding
commented out code, 265
comments in code, 256-258

comments in source control, 254-256
compiler warnings, 264
duplicated code, 264-265
effort versus project size, 253
if/switch statement complexity, 266
importance of, 252-253
magic numbers, 265-266
naming conventions, 266-267
threads, 267
source control
benefits of, 138-140
branching, 141
comments in, 254-256
GUI tools, 142
selecting, 142
servers for, 142
third-party component usage, 143
types of, 140-141
usage by app creators, 142-143
what to include, 144
spaghetti code case study, 14-15
spikes, 24, 41
splitting bug reports, 303
sprints
case study, 231
milestones versus, 230-231
SpriteKit, 118
SQL (Structured Query Language) databases, 127
standard art assets, benefits of, 173-174
stand-up meetings, 29
state testing, defined, 306
states
of bug reports, 300-301
of screens
copying, 65-66
wireframing, 64-65
status reports, 228
expectations for, 223
stealing app ideas, 179-180
stencil wireframing style, 63
stock photos, 174
in games, 183
stock sound effects, 177
storing data, 125-131
access methods, 127-130
amount of storage needed, 126-127
frequency of changes, 131
performance, 131
security, 130
services, 152
types of information to store, 125-126
structured data, 127
subcontractors, 11, 195, 198
submitting apps
avoiding last-minute changes, 333-334
marketing materials, creating, 330-331
optimization updates case study, 333-334
rejections. *See* rejections
resubmission, 340
reviewer instructions, writing, 331-332
submission process, 334-335
Subversion, 141
sunk cost fallacy, 269
surveys for beta testers, 319
SVN (source control system), 141
switch statement complexity, 266
synchronization, 133-134
synthesizing sounds, 177
system of record, maintaining, 80-82

T

tablets
adapting screens for, 67
phone screens in, 75-76
tasks
as bug-report category, 291
in estimates, 243
poor estimation of, 277-282
unexpected, 280
in use case development, 57
team members
developers as testers, 313-315
external testers, 315-319
professional testers, 311-313
technical support, interviewing developers, 223-224
template app sites, 185-189
drag-this-and-drop-that apps, 187, 189
fill-out-this-form apps, 187
package-your-website apps, 187
test coverage, 262
TestFlight, 159, 310
testing phase (app development), 45. *See also* bugs
at end of project only (project failure reasons), 26-27
automated testing, 48-49, 262, 314-315
beta testing, distribution tools, 159-160

feedback. *See* feedback
forced-failure testing, 313-314
negative testing, 47
overly optimistic mindset, 46
performance testing case study, 23
regression testing, 46
scheduling, 308-311
with simulators/emulators, 155-156
skills needed, 168
team members
developers as testers, 313-315
external testers, 315-319
professional testers, 311-313
types of, 305-306
usability testing, 47
user testing, 50
when to test, 45-48
text
conditional formatting, 123-124
full text search, 129
length assumptions in design phase, 39
third-party components, 164
source control usage, 143
third-party frameworks, 111
abandoned components, 115-116
adapting to new features, 116
bugs, 112
case study, 114
corner cases, 118-119
developer experience with, 113
documentation, 116
game engines, 117-118
licensing, 116
locking into, 113
quality versus popularity, 112-113
risk mitigation, 117
savings with, 112
third-party libraries, 260-261
third-party services, 133
thread safe, defined, 302
threads, 8
case study, 9-10
understanding, 267
Through the Looking Glass (Carroll), 343
time constraints, 251-253
time estimates (project failure reasons), 3
time-and-materials work, 219-220
timing
for independent developer projects, 202-203
invoice submissions, 221, 223
of missed milestones, 282-283
TIOBE Code Quality Indicator, 262
titles of screens, prototyping, 92
tools
beta testing distribution, 159-160
bug trackers. *See* bug trackers
CI (continuous integration) servers, 157-159
crash reporting, 160-161
development environment, 154-156
end-user feedback, 161-162
external services. *See* external services
interactive prototype creation, 78
interviewing developers about, 223
project/schedule management, 148-150
selecting based on project size, 138
source control. *See* source control
for wireframing, 61-62
tracer bullets, defined, 41
tracking schedules. *See* schedule tracking tools
training new developers, 203-204
transitions in design phase, 38
triage (of bugs), 323
troubleshooting server connections, skills needed, 168-169
trust, establishing, 137, 193
tutorial screens in first-run workflow, 71
Twitter, 151
typical users in use case development, 56

U

UI stencil wireframing style, 63
UI threads, 267
uncertainty (estimate accuracy), 240-241
unexpected tasks, 280
unforeseen circumstances, cost overruns, 25
unforeseen external dependencies, 279
unit tests, 262
Unity, 118
universal apps, phone screens in tablets, 75-76
unnecessary work, cost overruns, 24-25
updates, new app releases, 342-343
usability testing, 47
use cases, developing
scripts, 58
situations, 57

tasks, 57
typical users, 56
user experience design, skills needed, 169-173
user feedback. *See* feedback
user interface case study, 23
user preferences, 125
user testing, 50
user-facing functionality in testing schedule, 308

V

vector icon packages, 174
verification testing, defined, 305
verifying bugs, 326
version control. *See* source control
versions
in beta release cycles, 321-322
new releases, 342-343
video components, 120
videos
of app usage, creating, 332
for improving programming skills, 166
virtual keyboard, prototyping, 93
Visio, 61

W–Z

warnings (compiler), 264
wasted effort, 280-281
cost overruns, 24-25
web API servers, 131
web development, mobile development versus, 5-8. *See also* app development
web services, 132
websites
as apps, 110-111
building apps from, 187
whack-a-mole problem, 27-29, 272
white box testing, defined, 306
whole-screen data files, 128-129
Wi-Fi peripherals, 122
wireframing, 35-36, 58-59
adapting screens for tablets, 67
additional workflows, 67-68
barebones style, 62-63
buttons versus gestures, 75
consolidating and refining screens, 73-74
converting for interactive prototypes, 82-86
copying screens, 65-66
drawing tools for, 61-62
dynamic screens, 64-65
error functions, determining, 69-70
feedback on, 72-73
first-run workflow, 70-71
grouping screen data, 74-75
initial screen sketches, 59-61
interactive prototypes versus, 80-82
interactive style, 63
phone screens in tablets, 75-76
photorealistic style, 63
simplified scrolling, 75
stencil style, 63
wizards, 78
won't fix (bug state), 301
workflows
additional workflows, 67-68
core experience, 54-58
documentation as part of, 18
error functions, determining, 69-70
first-run workflow, 70-71
golden path. *See* golden path
logged out workflow, prototyping, 93-94
prototyping. *See* prototyping
Worldwide Developers Conference (WWDC), 336-337
writing
bug reports, 146-147, 296-298
commercials, 55
copywriting, skills needed, 178-179
reviewer instructions, 331-332
servers, 132-133
WWDC (Worldwide Developers Conference), 336-337

Xamarin, 118

Zawinsky's law, 35
Zuckerberg, Mark, 109