

Engineering Software as a Service: An Agile Approach Using Cloud Computing Second Edition, 2.0b6

Armando Fox and David Patterson

June 1, 2021

Copyright 2021 by Armando Fox and David Patterson.
You are free to make digital or printed copies of this material for your own personal use.
You may not redistribute this material in either digital or printed form, whether or not for financial gain, without the express permission of the copyright holders.

Book version: 2.0b6

The cover background is a photo of the **Aqueduct of Segovia**, Spain. We chose it as an example of a beautiful, long-lasting design. The full aqueduct is about 20 miles (32 km) long and was built by the Romans in the 1st or 2nd century CE. This photo is of the half-mile long, 92 foot high segment (0.8 km long, 28 m high) built using unmortared granite blocks. The Roman designers followed the architectural principles in the ten-volume series **De Architectura** (“On Architecture”), written in 15 BCE by Marcus Vitruvius Pollio. It was untouched until the 1500s CE, when King Ferdinand and Queen Isabella performed the first reconstruction of these arches. The aqueduct was in use and delivering water until recently.

Cover photo derived from an original photo by Bernard Gagnon, licensed under CC-BY-SA 3.0, https://commons.wikimedia.org/wiki/Aqueduct#/media/File:Aqueduct_of_Segovia_02.jpg

Both the print book and ebook were prepared with L^AT_EX and a series of Ruby packages, all freely available at <http://github.com/armandofox/latex2ebook>.

Arthur Klepchukov designed the covers and graphics for all versions.

Publisher's Cataloging-in-Publication

Fox, Armando.

Engineering software as a service : an agile approach using cloud computing / Armando Fox and David Patterson.

-- Second edition.

Includes bibliographical references.

ISBN 978-1-7352338-0-2

1. Software engineering. 2. Cloud computing.

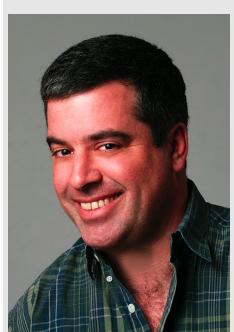
I. Patterson, David A. II. Title.

QA76.758.F684 2020 005.1

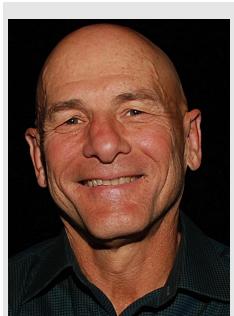
QBI14-600139

About the Authors

Armando Fox (pronouns: he, him, él) is a Professor of Computer Science, Diversity and Equity Officer at both the EECS Department level and Campus level, and Faculty Advisor for Digital Learning Strategy at UC Berkeley. He is an ACM Distinguished Scientist and in 2015 received the ACM Karl V. Karlstrom Outstanding Educator Award for his work on software engineering education. During his previous time at Stanford, he received teaching and mentoring awards from the Associated Students of Stanford University, the Society of Women Engineers, and Tau Beta Pi Engineering Honor Society. In 2016 he and co-author David Patterson received the Most Promising New Textbook award (“Texty”) from the Textbook and Academic Authors Association for the First Edition of this book. In previous lives he helped design the Intel Pentium Pro microprocessor, founded a successful startup to commercialize his UC Berkeley dissertation research on mobile computing including the world’s first mobile graphical web browser (Top Gun Wingman on Palm Pilot), and co-founded a couple of startups that were artistic successes. He received his BS in electrical engineering and computer science from MIT and his MS from the University of Illinois at Urbana-Champaign. He is also a classically-trained musician, freelance Music Director, and bilingual/bicultural (Cuban-American) New Yorker transplanted to San Francisco.



David Patterson (pronouns: he, him) recently retired from a 40-year career as a Professor of Computer Science at UC Berkeley. In the past, he served as Chair of Berkeley’s Computer Science Division, Chair of the Computing Research Association, and President of the Association for Computing Machinery. His best-known research projects are Reduced Instruction Set Computers (RISC), Redundant Arrays of Inexpensive Disks (RAID), and Networks of Workstations (NOW). This research led to many papers, 6 books, and more than 35 honors, including election to the National Academy of Engineering, the National Academy of Sciences, and the Silicon Valley Engineering Hall of Fame; being named a Fellow of the Computer History Museum, ACM, IEEE, and both AAAS organizations; and most recently, the ACM A.M. Turing Award, shared with Prof. John Hennessy of Stanford University for their work on RISC and their quantitative approach to computer architecture and design. His teaching awards include the UC Berkeley Distinguished Teaching Award, the ACM Karl V. Karlstrom Outstanding Educator Award, the IEEE Mulligan Education Medal, and the IEEE Undergraduate Teaching Award. Prior to winning the Textbook Excellence Award (“Texty”) for this book, he received one for his pioneering textbook on computer architecture. He received all his degrees from UCLA, which awarded him an Outstanding Engineering Academic Alumni Award. He grew up in California, and for fun he enters sporting events with his two adult sons, including weekly soccer games and charity bike rides.

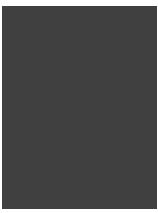


This is beta version 2.0b6. There are three “placeholders” for new programming assignments (CHIPS—see below) that are still being developed. Associations (CHIPS 5.7) and Caching and Indices (CHIPS 12.8) will provide hands-on practice for those concepts in the corresponding chapters.

We are also working on a JavaScript/AJAX CHIPS (6.9). JavaScript frameworks continue to proliferate, including many that barely existed when the First Edition was released, while JavaScript continues to bring unique debugging and programming challenges that are not as well supported by tools as Ruby and Rails. As a result of all this churn, the material in Chapter 6 is still evolving.

Quick Contents

Preface to the Second Edition	viii
1 Introduction to Software as a Service, Agile Development, and Cloud Computing	2
Part I: Software as a Service: Frameworks and Languages	
2 How to Learn a New Language	44
3 SaaS Application Architecture: Microservices, APIs, and REST ...	72
4 SaaS Framework: Rails as a Model–View–Controller Framework .	96
5 SaaS Framework: Advanced Programming Abstractions for SaaS .	124
6 Mobile and Desktop SaaS Clients: JavaScript Introduction	152
Part II: Agile Software Development	
7 Requirements: BDD and User Stories	204
8 Testing: Test-Driven Development	236
9 Software Maintenance: Enhancing Legacy Software Using Refactoring and Agile Methods	268
10 Agile Teams	302
11 Design Patterns for SaaS Apps	336
12 Dev/Ops	370
Afterword	410



Contents

Preface	viii
1 Introduction to Software as a Service, Agile Development, and Cloud Computing	2
1.1 Introduction	5
1.2 Software Development Processes: Plan-and-Document	6
1.3 Software Development Processes: The Agile Manifesto	11
1.4 Software Quality Assurance: Testing	16
1.5 Productivity: Conciseness, Synthesis, Reuse, and Tools	18
1.6 SaaS and Service Oriented Architecture	21
1.7 Deploying SaaS: Cloud Computing	24
1.8 Deploying SaaS: Browsers and Mobile	26
1.9 Beautiful vs. Legacy Code	30
1.10 Guided Tour and How To Use This Book	32
1.11 Fallacies and Pitfalls	35
1.12 Concluding Remarks: Software Engineering Is More Than Programming	36
I Software as a Service: Frameworks and Languages	43
2 How to Learn a New Language	44
2.1 Prelude: Learning to Learn Languages and Frameworks	46
2.2 Pair Programming	48
2.3 Introducing Ruby, an Object-Oriented Language	50
2.4 Ruby Idioms: Poetry Mode, Blocks, Duck Typing	59
2.5 CHIPS: Ruby Intro	64
2.6 Gems and Bundler: Library Management in Ruby	64
2.7 Fallacies and Pitfalls	67
2.8 Concluding Remarks: How (Not) To Learn a Language By Googling	69
3 SaaS Application Architecture: Microservices, APIs, and REST	72
3.1 The Web's Client–Server Architecture	74
3.2 SaaS Communication Uses HTTP Routes	76
3.3 CHIPS: HTTP and URIs	80

3.4	From Web Sites to Microservices: Service-Oriented Architecture	80
3.5	RESTful APIs: Everything is a Resource	84
3.6	RESTful URIs, API Calls, and JSON	89
3.7	CHIPS: Create and Deploy a Simple SaaS App	92
3.8	Fallacies and Pitfalls	92
3.9	Concluding Remarks: Continuity From CGI to SOA	94
4	SaaS Framework: Rails as a Model–View–Controller Framework	96
4.1	The Model–View–Controller (MVC) Architecture	98
4.2	Rails Models: Databases and Active Record	100
4.3	CHIPS: ActiveRecord Basics	105
4.4	Routes, Controllers, and Views	105
4.5	CHIPS: Rails Routes	110
4.6	Forms	111
4.7	CHIPS: Hangperson on Rails	116
4.8	Debugging: When Things Go Wrong	116
4.9	CHIPS: Hello Rails	120
4.10	Fallacies and Pitfalls	120
4.11	Concluding Remarks: Rails as a Service Framework	121
5	SaaS Framework: Advanced Programming Abstractions for SaaS	124
5.1	DRYing Out MVC: Partials, Validations and Filters	126
5.2	Single Sign-On and Third-Party Authentication	131
5.3	CHIPS: Rails Intro	136
5.4	Associations and Foreign Keys	136
5.5	Through-Associations	140
5.6	RESTful Routes for Associations	143
5.7	CHIPS: Associations	146
5.8	Other Types of Code	147
5.9	Fallacies and Pitfalls	149
5.10	Concluding Remarks: Languages, Productivity, and Beauty	149
6	Mobile and Desktop SaaS Clients: JavaScript Introduction	152
6.1	JavaScript: The Big Picture	154
6.2	Introducing ECMAScript	157
6.3	Classes, Functions and Constructors	163
6.4	The Document Object Model (DOM) and jQuery	166
6.5	The DOM and Accessibility	169
6.6	Events and Callbacks	173
6.7	AJAX: Asynchronous JavaScript And XML	178
6.8	Testing JavaScript and AJAX	183
6.9	CHIPS: AJAX Enhancements to RottenPotatoes	190
6.10	Single-Page Apps and JSON APIs	190
6.11	Fallacies and Pitfalls	195
6.12	Concluding Remarks: JavaScript Past, Present and Future	199

II Agile Software Development 203

7 Requirements: BDD and User Stories	204
7.1 Behavior-Driven Design and User Stories	206
7.2 SMART User Stories	209
7.3 Lo-Fi User Interface Sketches and Storyboards	211
7.4 Points and Velocity	213
7.5 Agile Cost Estimation	216
7.6 Cucumber: From User Stories to Acceptance Tests	217
7.7 CHIPS: Intro to BDD and Cucumber	220
7.8 Explicit vs. Implicit and Imperative vs. Declarative Scenarios	220
7.9 The Plan-And-Document Perspective on Documentation	223
7.10 Fallacies and Pitfalls	230
7.11 Concluding Remarks: Pros and Cons of BDD	233
8 Testing: Test-Driven Development	236
8.1 FIRST, TDD, and Red–Green–Refactor	238
8.2 Anatomy of a Test Case: Arrange, Act, Assert	240
8.3 Isolating Code: Doubles and Seams	243
8.4 Stubbing the Internet	248
8.5 CHIPS: Intro to RSpec on Rails	249
8.6 Fixtures and Factories	249
8.7 Coverage Concepts and Types of Tests	255
8.8 Other Testing Approaches and Terminology	258
8.9 CHIPS: The Acceptance Test/Unit Test Cycle	260
8.10 The Plan-And-Document Perspective on Testing	260
8.11 Fallacies and Pitfalls	264
8.12 Concluding Remarks: TDD vs. Conventional Debugging	266
9 Software Maintenance: Enhancing Legacy Software Using Refactoring and Agile Methods	268
9.1 What Makes Code “Legacy” and How Can Agile Help?	270
9.2 Exploring a Legacy Codebase	273
9.3 Establishing Ground Truth With Characterization Tests	277
9.4 Comments and Commits: Documenting Code	279
9.5 Metrics, Code Smells, and SOFA	281
9.6 Method-Level Refactoring: Replacing Dependencies With Seams	286
9.7 The Plan-And-Document Perspective on Working With Legacy Code	292
9.8 Fallacies and Pitfalls	297
9.9 Concluding Remarks: Continuous Refactoring	298
10 Agile Teams	302
10.1 It Takes a Team: Two-Pizza and Scrum	304
10.2 Using Branches Effectively	306
10.3 Pull Requests and Code Reviews	311
10.4 Delivering the Backlog Using Continuous Integration	315
10.5 CHIPS: Agile Iterations	320
10.6 Reporting and Fixing Bugs: The Five R’s	320

10.7	The Plan-And-Document Perspective on Managing Teams	322
10.8	Fallacies and Pitfalls	330
10.9	Concluding Remarks: From Solo Developer to Teams of Teams	331
11	Design Patterns for SaaS Apps	336
11.1	Patterns, Antipatterns, and SOLID Class Architecture	338
11.2	Just Enough UML	342
11.3	Single Responsibility Principle	345
11.4	Open/Closed Principle	347
11.5	Liskov Substitution Principle	351
11.6	Dependency Injection Principle	354
11.7	Demeter Principle	358
11.8	The Plan-And-Document Perspective on Design Patterns	362
11.9	6S: A Clean Code Checklist	363
11.10	Fallacies and Pitfalls	365
11.11	Concluding Remarks: Frameworks Capture Design Patterns	366
12	Dev/Ops	370
12.1	From Development to Deployment	373
12.2	Three-Tier Architecture	375
12.3	Responsiveness, Service Level Objectives, and Apdex	378
12.4	Releases and Feature Flags	382
12.5	Monitoring and Finding Bottlenecks	385
12.6	Improving Rendering and Database Performance With Caching	387
12.7	Avoiding Abusive Database Queries	391
12.8	CHIPS: Exploiting Caching and Indices	394
12.9	Security: Defending Customer Data in Your App	394
12.10	The Plan-And-Document Perspective on Operations	400
12.11	Fallacies and Pitfalls	402
12.12	Concluding Remarks: Beyond PaaS Basics	406
13	Afterword	410
13.1	Looking Backwards	412
13.2	Looking Forwards	413
13.3	Essential Readings	415
13.4	Last Words	416



Preface to the Second Edition

Why so many quotes? We think quotes make the book more fun to read, but they are also an efficient mechanism to pass along wisdom from the elders, and to help set cultural standards for good software engineering. We also want readers to pick up a bit of history of the field, which is why we feature quotes from Turing Award winners to open each chapter and throughout the text.

If you want to build a ship, don't drum up the men to gather wood, divide the work and give orders. Instead, teach them to yearn for the vast and endless sea.

—Antoine de Saint-Exupéry, *Citadelle*, 1948

If you're nostalgic for the Welcome from the First Edition, you can read it at <http://www.saasbook.info/welcome-1st-edition>¹.

We created the First Edition of ESaaS in 2014 to help other instructors and students of software engineering practice what we had discovered: Agile+SaaS is not just a great way to develop and deploy software, it's also a great fit for *teaching* software engineering. We have been humbled by the success of the book, the accompanying instructor materials (for which visit saasbook.info²), and the Massive Open Online Courses on edX as a way of “spreading the word!”

What's New: COD, CHIPS, and Codio

This Second Edition has an improved structure (in our opinion) as well as substantial revisions to nearly half of the material.

COD and CHIPS. Most of the book's sections are Content-Oriented Didactics (COD): the conceptual vocabulary that shows the learner *how to think about* an important idea. Interspersed after every few COD sections are Coding/Hands-on Integrated Programming activities (CHIPS), where students learn by doing, applying the ideas of the COD sections in hands-on exercises. Each CHIPS has a rating from one to three “aqueducts” reflecting the relative time and effort required to complete it.

All-in-one course using Codio. We have worked closely with Codio to integrate COD, CHIPS, and autograding into their education-focused IDE. We strongly urge instructors or students to use Codio to get started as quickly as possible, including built-in autograding for most of the CHIPS and a preconfigured curated environment with the correct versions of all tools needed. Visit codio.com/esaas³ to get started. If you're not using Codio, the starter code and student-facing documentation for each CHIPS are available in a public GitHub repository whose name is given as part of each CHIPS, and instructors can visit saasbook.info to gain access to reference solutions and Gradescope-compatible autograders.

What's New: Major Content Changes

Mobile-first, API-first exposition of SaaS. Since the First Edition, “cloud + client” has remained the dominant way that software is developed, but SaaS has transitioned from delivering primarily HTML views to delivering data to mobile clients over APIs. As well, much greater attention is being paid to designing for persons with disabilities. We therefore immediately motivate the use of a resource-based, API-centric approach to thinking about server design, and the use of mobile-first and mobile-friendly frameworks based on open standards, such as Bootstrap, for the client side. The new “API first” exposition should empower learners to think about resource-centric design of their apps and how a RESTful API exposes those resources to a client, and then transfer this thinking to the development of native mobile apps.

From “learning Ruby and Rails” to “learning a language and framework.” Recognizing that languages and frameworks continue to evolve, our expositions of Ruby, Rails, and JavaScript now suggest a more general strategy for learning new languages and frameworks rapidly, and on understanding the relationship between a framework and the language features that make it work well.

What Has Not Changed?

Our students at Berkeley still ask “Will this course teach me *X*?” where popular values of *X* in 2019 include React, AWS Lambda, MongoDB, and Node. Our answer has not changed since the First Edition. The software ecosystem evolves so rapidly that at any given time you will have many frameworks and tools from which to choose. Since our choices won’t please everyone and will probably be outdated in a few years anyway, we still choose the tools that best support our pedagogical goal of teaching a particular methodology for developing great software. Our hope is that learners can use our suggested approaches and principles to help learn new languages and frameworks rapidly.

Acknowledgments

Some acknowledgments are in order, as it truly takes a village to create and maintain a good set of course materials. Beyond the people we thanked in the First Edition, the following colleagues were particularly helpful in reviewing the technical accuracy of the Second Edition changes: Prof. Kristin Stephens-Martinez, Duke University (Chapter 1); Prof. Mark Smucker, University of Waterloo (Chapter 2); Blagovesta Kostova, EPFL (Chapter 3); Prof. Michael Verdicchio, The Citadel Military College of South Carolina (Chapter 4); Lic. Matías Mascazzini, independent Rails developer (Chapter 5); Prof. Tom Hastings, University of Colorado at Colorado Springs (Chapter 6); Prof. Hank Walker, Texas A&M University (Chapters 7 and 8); Prof. Prabhat Vaish, New Jersey Institute of Technology (Chapter 9); Prof. Anastasia Kurdia, Tulane University (Chapter 10); Prof. Ed Gehringer, North Carolina State University (Chapter 11); Prof. Daniel Cordeiro, Universidade de São Paulo (Chapter 12). Finally, many thanks to Peter Zhang, legal technologist, Melbourne, Australia, for an exceedingly thorough and precise proofreading of the entire 2nd Edition.

As always, the core members of the “Beta Gold” group, formed way back in the days of the First Edition, have stuck with us and proactively helped improve the course in ways too numerous to mention that benefit everyone who uses the materials: Michael and Hank

from the list above, plus Rose Williams, Binghamton University, and Kristen Walcott-Justice, University of Colorado at Colorado Springs.

Our colleagues at GitHub, and particularly Director of Developer Education Vanessa Gennarelli (@mozzadrella), continue to be more generous and supportive of our education efforts than we have any right to expect.

The Codio team, especially Elise Deitrick, Max Kraev, and CEO Phillip Snalune, have done phenomenal work in beautifully integrating both the book and exercises into the Codio platform, providing a one-stop shop for instructors and students wanting to get started quickly. We hope the seamlessness of that experience encourages many more learners to try ESaaS.

Finally, as always, we thank the thousands of UC Berkeley students and teaching assistants, and the hundreds of thousands of MOOC students, for their debugging help and their continuing interest in this material!

Armando Fox
January 2021
San Francisco, California

Notes

¹<http://www.saasbook.info/welcome-1st-edition>

²<http://www.saasbook.info>

³<https://codio.com/esaas>

1

Introduction to Software as a Service, Agile Development, and Cloud Computing

Donald Knuth (1938–), one of the most illustrious computer scientists, received the Turing Award in 1974 for major contributions to the analysis of algorithms and the design of programming languages, and in particular for his contributions to his multi-volume *The Art of Computer Programming*, arguably the definitive reference on analysis of algorithms. Knuth also invented the widely-used TeX typesetting system, with which this book was prepared.



Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

—Donald Knuth, *Literate Programming*, 1984

1.1	Introduction	5
1.2	Software Development Processes: Plan-and-Document	6
1.3	Software Development Processes: The Agile Manifesto	11
1.4	Software Quality Assurance: Testing	16
1.5	Productivity: Conciseness, Synthesis, Reuse, and Tools	18
1.6	SaaS and Service Oriented Architecture	21
1.7	Deploying SaaS: Cloud Computing	24
1.8	Deploying SaaS: Browsers and Mobile	26
1.9	Beautiful vs. Legacy Code	30
1.10	Guided Tour and How To Use This Book	32
1.11	Fallacies and Pitfalls	35
1.12	Concluding Remarks: Software Engineering Is More Than Programming	36

Prerequisites and Concepts

Each chapter opening starts with a brief summary of the chapter's prerequisites and big concepts. Prerequisites are skills or knowledge you should *already have* in order to get the most out of the material in the chapter. Big concepts are the main ideas we want you to take away *after finishing* the chapter when you step back from the details.

Prerequisites:

The first prerequisite is to determine whether this book is right for you!

This book is a good fit if you want to...	But not if you want to...
Allow software design principles to inform your evaluation and creation of new technologies	Just learn framework X, without understanding its design principles
Learn how to learn new frameworks and languages, and put them to use quickly	Follow a step-by-step "recipe" tutorial for a specific framework or language
Learn by doing	Learn by only reading and watching videos

For this chapter, the technical prerequisite is basic knowledge of **HTML**, the HyperText Markup Language that is the *lingua franca* of the web. We recommend the free Introduction to HTML¹ module from the Mozilla Developer Network. We suggest working through all the sections under *Guides* and the two *Assessments*.

Concepts:

The big concepts of this chapter are the contrasts between Plan-and-Document software development and Agile software development, and the synergy among Agile development, Software as a Service, and cloud computing.

- *Plan-and-Document* software development processes or *lifecycles* rely on careful, up-front planning, whereas *Agile software development* relies on incrementally refining a prototype with continuous feedback from the customer over the course of many 1–4 week *iterations*. Of the two, Agile has the superior track record for managing change, running compact projects with small teams, and delivering quality software on time and within budget.
- *Software quality* is defined as providing business value to both customers and developers and involves many kinds of testing. In Agile, the developers themselves, rather than a separate QA team, bear primary responsibility for software quality.
- Clarity via conciseness, synthesis, reuse, and automation via tools are four paths to improving developer productivity. *Ruby on Rails* employs all of them.
- *Software as a Service (SaaS)* is software deployed on Internet servers accessible to millions of users. Compared with *Software as a Product* (SaaP) that users install on their devices, SaaS is easier to upgrade and evolve because there is only a single copy deployed in the field. *Cloud Computing* supplies the dependable and scalable computation and storage for SaaS by utilizing *Warehouse Scale*

Computers containing as many as 100,000 servers. Economies of scale allow Cloud Computing to be offered as a utility, where you pay only for actual use.

- Mobile devices now account for the majority of visits to web sites. “Mobile-first” apps can be developed, tested, and deployed using the same tools as SaaS for desktop browsers, using ***responsive web design*** to automatically adapt to a variety of screen sizes while accommodating for users with disabilities.
- ***Legacy Code*** evolution is vital in the real world, yet often ignored in software engineering books and courses. Agile practices enhancing code each iteration, so the skills gained also apply to legacy code.

Topic	Amazon.com	ACA Oct	ACA Nov	ACA Dec
Customers/Day (Goal)	—	50,000	50,000	30,000
Customers/Day (Actual)	>10,000,000	800	3,700	34,300
Average Response time (seconds)	0.2	8	1	1
Downtime/Month (hours)	0.07	446	107	36
Availability (% up)	99.99%	40%	85%	95%
Error Rate	—	10%	10%	—
Secure	Yes	No	No	No

Figure 1.1: Comparing Amazon.com and HealthCare.gov during its first three months. (Thorp 2013) After its stumbling start, the deadline was extended from December 15, 2013 to March 31, 2014, which explains the lower goal in customers per day in December. Note that availability for ACA does *not* include time for “scheduled maintenance,” which Amazon does include (Zients 2013). The error rate was for significant errors on the forms sent to insurance companies (Horsley 2013). The site was widely labeled by security experts as insecure, as the developers were under tremendous pressure to get proper functionality, and little attention was paid to security (Harrington 2013).

1.1 Introduction

Now, this is real simple. It's a website where you can compare and purchase affordable health insurance plans, side-by-side, the same way you shop for a plane ticket on Kayak or the same way you shop for a TV on Amazon... Starting on Tuesday, every American can visit HealthCare.gov to find out what's called the insurance marketplace... So tell your friends, tell your family... Make sure they sign up. Let's help our fellow Americans get covered. (Applause.)

—President Barack Obama, Remarks on the Affordable Care Act, Prince George’s Community College, Maryland, September 26, 2013

...it has now been six weeks since the Affordable Care Act's new marketplaces opened for business. I think it's fair to say that the rollout has been rough so far, and I think everybody understands that I'm not happy about the fact that the rollout has been, you know, fraught with a whole range of problems that I've been deeply concerned about.

—President Barack Obama, Statement on the Affordable Care Act, The White House Press Briefing Room, November 14, 2013

When the *Affordable Care Act* (ACA) was passed in 2010, it was seen as the most ambitious US social program in decades, and it was perhaps the crowning achievement of the Obama administration. Just as millions shop for items on Amazon.com, HealthCare.gov—also known as the Affordable Care Act website—was supposed to let millions of uninsured Americans shop for insurance policies. Despite taking three years to build, it fell flat on its face when it debuted on October 1, 2013. Figure 1.1 compares Amazon.com to HealthCare.gov in the first three months of operation, demonstrating that not only was it slow, error prone, and insecure, it was also down much of the time.

Why is it that companies like Amazon.com can build software that serves a much large customer base so much better? While the media uncovered many questionable decisions, a surprising amount of the blame was placed on the *methodology* used to develop the software (Johnson and Reed 2013). Given their approach, as one commentator said, “The real news would have been if it actually did work.” (Johnson 2013a)

We’re honored to have the chance to explain how Internet companies and others build successful software services and extend the reach of those services to the billions of mobile devices out there. As this introduction illustrates, this field is not some dreary academic discipline where few care what happens: failed software projects can become infamous, and

can even derail Presidents. On the other hand, successful software projects can create services that billions of people use every day whose creators become household names. All involved with such services are proud to be associated with them, unlike the ACA.

The rest of this chapter explains why disasters like ACA can happen and how to avoid repeating this unfortunate history. We start our journey with the origins of software engineering itself, which began with software development methodologies that placed a heavy emphasis on planning and documenting, since that approach had worked well in other “big” engineering projects such as civil engineering. We next review the statistics on how well the *Plan-and-Document* methodologies worked, alas documenting that project outcomes like ACA are all too common, if not as well known. The frequently disappointing results of following conventional wisdom in software engineering inspired a few software developers to stage a revolt. While the *Agile Manifesto* was quite controversial when it was announced, over time Agile software development has overcome its critics. Agile allows small teams to outperform the industrial giants, especially for small projects. Our next step in the journey demonstrates how *service-oriented architecture* allows the successful composition of large software services like Amazon.com from many smaller software services developed and operated by small Agile teams.

As a final but critical point, it’s rare in practice for software developers to do “green field” development, in which they start from a blank slate. It’s much more common to enhance large existing code bases. The next step in our journey observes that unlike Plan-and-Document, which aims at a perfect design up front and then implements it, the Agile process spends almost all of its time enhancing working code. Thus, by getting good at Agile, you are also practicing the skills you need to evolve existing code bases.

To start us on our journey, we introduce the software methodology used to develop HealthCare.gov.

1.2 Software Development Processes: Plan-and-Document

If builders built buildings the way programmers wrote programs, then the first wood-pecker that came along would destroy civilization.

—Weinberg’s *Second Law*, 1978, attributed to Gerald Weinberg, University of Nebraska computer scientist

The general unpredictability of software development in the late 1960s, along with the software disasters similar to ACA, led to the study of how high-quality software could be developed on a predictable schedule and budget. Drawing the analogy to other engineering fields, the term **software engineering** was coined (Naur and Randell 1969). The goal was to discover methods to build software that were as predictable in quality, cost, and time as those used to build bridges in civil engineering.

One thrust of software engineering was to bring an engineering discipline to what was often unplanned software development. Before starting to code, come up with a plan for the project, including extensive, detailed documentation of all phases of that plan. Progress is then measured against the plan. Changes to the project must be reflected in the documentation and possibly to the plan.

The goal of all these “Plan-and-Document” software development processes is to improve predictability via extensive documentation, which must be changed whenever the goals change. Here is how textbook authors put it (Lethbridge and Laganiere 2002; Braude 2001):

Documentation should be written at all stages of development, and includes requirements, designs, user manuals, instructions for testers and project plans.

—Timothy Lethbridge and Robert Laganiere, 2002

Documentation is the lifeblood of software engineering.

—Eric Braude, 2001

This process is even embraced with an official standard of documentation: IEEE/ANSI standard 830/1993.

Governments like that of the US have elaborate regulations to prevent corruption when acquiring new equipment, which lead to lengthy specifications and contracts. Since the goal of software engineering was to make software development as predictable as building bridges, including elaborate specifications, government contracts were a natural match to Plan-and-Document software development. Thus, like many countries, US acquisition regulations left the ACA developers little choice but to follow a Plan-and-Document lifecycle.

Of course, like other engineering fields, the government has escape clauses in the contracts that let it still acquire the product even if it is late. Ironically, the contractor makes more money the longer it takes to develop the software. Thus, the art is in negotiating the contract and the penalty clauses. As one commentator on ACA noted (Howard 2013), “The firms that typically get contracts are the firms that are good at getting contracts, not typically good at executing on them.” Another noted that the Plan-and-Document approach is not well suited to modern practices, especially when government contractors focus on maximizing profits (Chung 2013).

An early version of this Plan-and-Document software development process was developed in 1970 (Royce 1970). It follows this sequence of phases:

1. Requirements analysis and specification
2. Architectural design
3. Implementation and Integration
4. Verification
5. Operation and Maintenance

Given that the earlier you find an error the cheaper it is to fix, the philosophy of this process is to complete a phase before going on to the next one, thereby removing as many errors as early as possible. Getting the early phases right could also prevent unnecessary work downstream. As this process could take years, the extensive documentation helps to ensure that important information is not lost if a person leaves the project and that new people can get up to speed quickly when they join the project.

Because it flows from the top down to completion, this process is called the **Waterfall** software development process or Waterfall software development **lifecycle**. Understandably, given the complexity of each stage in the Waterfall lifecycle, product releases are major events toward which engineers worked feverishly and which are accompanied by much fanfare. In the Waterfall lifecycle, the long life of software is acknowledged by a maintenance phase that repairs errors as they are discovered. New versions of software developed in the Waterfall model go through the same several phases, and take typically between 6 and 18 months.

(Sidebars like this one provide historical context or perspective. They are optional, but as George Santayana famously said, “Those who do not know history are condemned to repeat it.”) CGI Group won the contract for the back end of the ACA website. The initial estimate ballooned from US\$94M to \$292M (Begley 2013). This same company was involved in a Canadian firearms registry whose costs skyrocketed, from an initial estimate of US\$2M to \$2B (2×10^9). When MITRE investigated the problems with Massachusetts’ ACA website, it said CGI Group lacked expertise to build the site, lost data, failed to adequately test, and managed the project poorly (Bidgood 2014).

Windows 95 was heralded by a US\$300 million party² for which Microsoft hired comedian Jay Leno, lit up New York’s Empire State Building using the Microsoft Windows logo colors, and licensed “Start Me Up” by the Rolling Stones as the celebration’s theme song.

The Waterfall model can work well with well-specified tasks like NASA space flights, but it runs into trouble when customers change their minds about what they want. A Turing Award winner captures this observation:

Plan to throw one [implementation] away; you will, anyhow.

—Fred Brooks, Jr.

That is, it's easier for customers to understand what they want once they see a prototype and for engineers to understand how to build it better once they've done it the first time.

This observation led to a software development lifecycle developed in the 1980s that combines prototypes with the Waterfall model (Boehm 1986). The idea is to iterate through a sequence of four phases, with each iteration resulting in a prototype that is a refinement of the previous version. Figure 1.2 illustrates this model of development across the four phases, which gives this lifecycle its name: the **Spiral model**. The phases are:

1. Determine objectives and constraints of this iteration
2. Evaluate alternatives and identify and resolve risks
3. Develop and verify the prototype for this iteration
4. Plan the next iteration

Rather than document all the requirements at the beginning, as in the Waterfall model, the requirement documents are developed across the iteration as they are needed and evolve with the project. Iterations involve the customer before the product is completed, which reduces chances of misunderstandings. However, as originally envisioned, these iterations were 6 to 24 months long, so there is plenty of time for customers to change their minds during an iteration! Thus, Spiral still relies on planning and extensive documentation, but the plan is expected to evolve on each iteration.

Given the importance of software development, many variations of Plan-and-Document methodologies were proposed beyond these two. A recent one is called the **Rational Unified Process (RUP)** (Kruchten 2003), developed during the 1990s, which combines features of both Waterfall and Spiral lifecycles as well standards for diagrams and documentation. We'll use RUP as a representative of the latest thinking in Plan-and-Document lifecycles. Unlike Waterfall and Spiral, it is more closely allied to business issues than to technical issues.

Like Waterfall and Spiral, RUP has phases:

1. Inception: makes the business case for the software and scopes the project to set the schedule and budget, which is used to judge progress and justify expenditures, and initial assessment of risks to schedule and budget.
2. Elaboration: works with stakeholders to identify use cases, designs a software architecture, sets the development plan, and builds an initial prototype.
3. Construction: codes and tests the product, resulting in the first external release.
4. Transition: moves the product from development to production in the real environment, including customer acceptance testing and user training.

Big Design Up Front, abbreviated **BDUF**, is a name some use for software processes like Waterfall, Spiral, and RUP that depend on extensive planning and documentation. They are also known variously as **heavyweight**, **plan-driven**, **disciplined**, or **structured** processes.

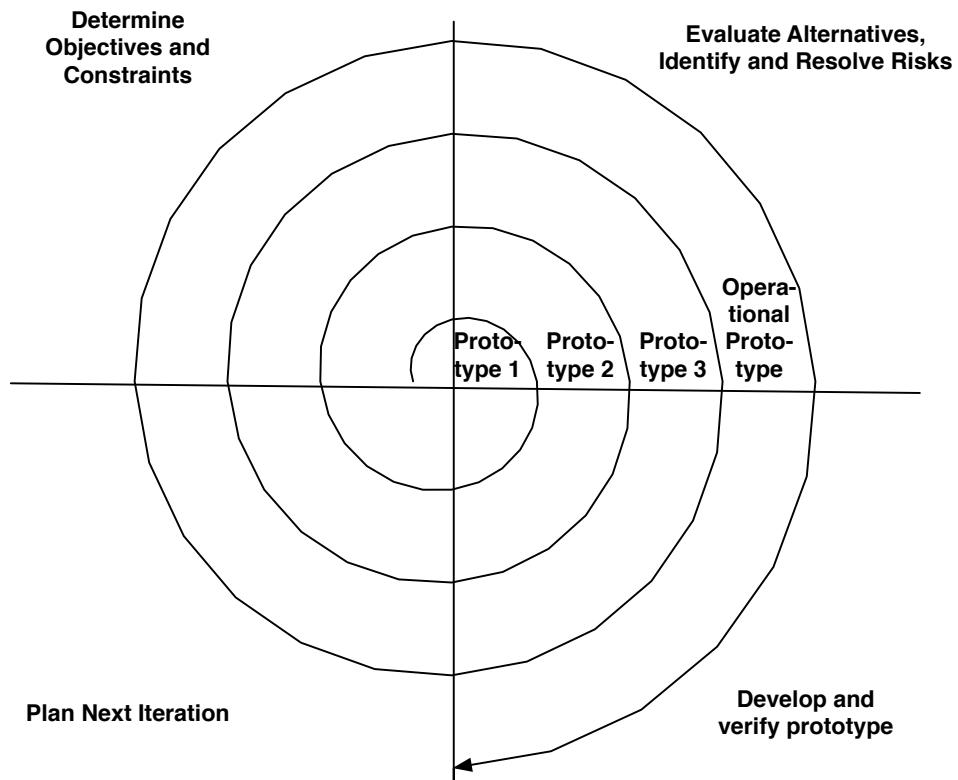


Figure 1.2: The Spiral lifecycle combines Waterfall with prototyping. It starts at the center, with each iteration around the spiral going through the four phases and resulting in a revised prototype until the product is ready for release.

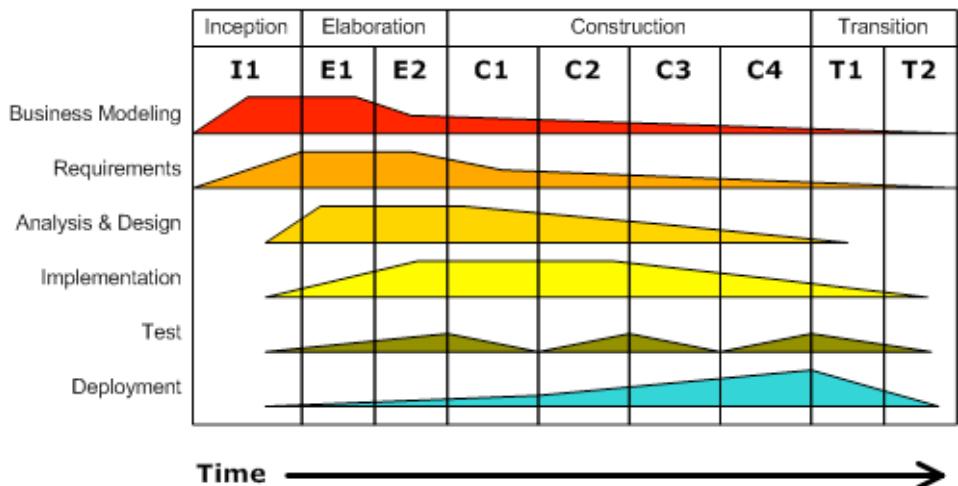


Figure 1.3: The Rational Unified Process lifecycle allows the project to have multiple iterations in each phase and identifies the skills needed by the project team, which vary in effort over time. RUP also has three “supporting disciplines” not shown in this figure: Configuration and Change Management, Project Management, and Environment. (Image from Wikimedia Commons by Dutchgilder.)

Unlike Waterfall, each phase involves iteration. For example, a project might have one inception phase iteration, two elaboration phase iterations, four construction phase iterations, and two transition phase iterations. Like Spiral, a project could also iterate across all four phases repeatedly.

In addition to the dynamically changing phases of the project, RUP identifies six “engineering disciplines” (also known as workflows) that people working on the project should collectively cover:

1. Business Modeling
2. Requirements
3. Analysis and Design
4. Implementation
5. Test
6. Deployment

These disciplines are more static than the phases, in that they nominally exist over the whole lifetime of the project. However, some disciplines get used more in earlier phases (like business modeling), some periodically throughout the process (like test), and some more towards the end (deployment). Figure 1.3 shows the relationship of the phases and the disciplines, with the area indicating the amount of effort in each discipline over time.

An unfortunate downside to teaching a Plan-and-Document approach is that students may find software development tedious (Nawrocki et al. 2002; Estler et al. 2012). Of course, this is hardly a strong enough reason not to teach it; the good news is that there are alternatives that work just as well for many projects that are a better fit to the classroom, as we describe in the next section.

Summary: The basic *activities* of software engineering are the same in all the software development process or **lifecycles**, but their interaction over time relative to product releases differs among the models. The Waterfall lifecycle is characterized by much of the design being done in advance of coding, completing each phase before going on to the next one. The Spiral lifecycle iterates through all the development phases to produce prototypes, but like Waterfall, the customers may only get involved every 6 to 24 months. The more recent Rational Unified Process lifecycle includes phases, iterations, and prototypes, while identifying the people skills needed for the project. All rely on careful planning and thorough documentation, and all measure progress against a plan.

■ Elaboration: SEI Capability Maturity Model (CMM)

(Elaborations are included for curious readers who want to know more about what is going on behind the curtain. Beginning readers can safely skip them the first time, but we hope as you become more experienced you'll find them more intriguing!)

The Software Engineering Institute at Carnegie Mellon University proposed the **Capability Maturity Model** (CMM) (Paultk et al. 1995) to evaluate organizations' software-development processes based on Plan-and-Document methodologies. The idea is that by modeling the software development process, an organization can improve them. SEI studies observed five levels of software practice:

1. Initial or Chaotic: undocumented/*ad hoc*/unstable software development.
2. Repeatable: not following rigorous discipline, but some processes repeatable with consistent results.
3. Defined: Defined and documented standard processes that improve over time.
4. Managed: Management can control software development using process metrics, adapting the process to different projects successfully.
5. Optimizing: Part of the management process is deliberate quantitative optimization of the development process.

CMM implicitly encourages an organization to move up the CMM levels. While not proposed as a software development methodology, many consider it one. For example, (Nawrocki et al. 2002) compares CMM Level 2 to the Agile software methodology (see next section).

Each section includes one or more questions to self-check whether you understood the material. It's okay to find that you sometimes need to re-read a section in order to get the self-check correct.

Self-Check 1.2.1. *What are a major similarity and a major difference between processes like Spiral and RUP versus Waterfall?*

- ◊ All rely on planning and documentation, but Spiral and RUP use iteration and prototypes to improve them over time versus a single long path to the product. ■

Self-Check 1.2.2. *What are the differences between the phases of these Plan-and-Document processes?*

- ◊ Waterfall phases separate planning (requirements and architectural design) from implementation. Testing the product before release is next, followed by a separate operations phase. The Spiral phases are aimed at an iteration: set the goals for an iteration; explore alternatives; develop and verify the prototype for this iteration; and plan the next iteration. RUP phases are tied more closely to business objectives: the inception phase makes the business case and sets schedule and budget; the elaboration phase works with customers to build an initial prototype; the construction phase builds and test the first version; and the transition phase deploys the product. ■

1.3 Software Development Processes: The Agile Manifesto

If a problem has no solution, it may not be a problem, but a fact—not to be solved, but to be coped with over time.

—Shimon Peres

While plan-and-development processes brought discipline to software development, there were still software projects that failed so disastrously that they live in infamy. Programmers have heard these sorry stories of the **Ariane 5 rocket explosion**, the **Therac-25** lethal

radiation overdose, **Mars Climate Orbiter** disintegration, and the FBI **Virtual Case File** project abandonment so frequently that they are clichés. No software engineer would want these projects on their résumés.

One article even listed a “Software Wall of Shame” with dozens of highly-visible software projects that collectively were responsible for losses of \$17B, with the majority of these projects abandoned (Charette 2005).

Figure 1.4 summarizes four surveys of software projects. With just 10% to 16% on time and on budget, more projects were cancelled or abandoned than met their mark. A closer look at the 13% of projects in survey (b) that were successful is even more sobering, as fewer than 1% of new development projects met their schedules and budgets. Although the first three surveys are 10 to 25 years old, survey d) is from 2013. Nearly 40% of these large projects were cancelled or abandoned, and 50% were late, over budget, and missing functionality. Using history as our guide, poor President Obama had only a one in ten chance that HealthCare.gov would have a successful debut.

Perhaps the “Reformation moment” for software engineering was the **Agile Manifesto** in February 2001. A group of software developers met to develop a lighter-weight software lifecycle. Here is exactly what the **Agile Alliance** nailed to the door of the “Church of Plan-and-Document”:

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

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

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

This alternative development model is based on *embracing change as a fact of life*: developers should continuously refine a working but incomplete prototype until the customer is happy with the result, with the customer offering feedback on each iteration. Agile emphasizes **test-driven development (TDD)** to reduce mistakes by writing the tests *before* writing the code, **user stories** to reach agreement and validate customer requirements, and **velocity** to measure project progress. We’ll cover these topics in detail in later chapters.

Regarding software lifetimes, the Agile software lifecycle is so quick that new versions are available every week or two—with some even releasing every day—so they are not even special events as in the Plan-and-Document models. The assumption is one of basically continuous improvement over its lifetime.

We mentioned in the prior section that newcomers can find Plan-and-Document processes tedious, but this is not the case for Agile. This perspective is captured by a software engineering instructor’s early review of Agile:

Remember when programming was fun? Is this how you got interested in computers in the first place and later in computer science? Is this why many of our majors enter the discipline—because they like to program computers? Well, there may be promising and respectable software development methodologies that are perfectly suited to these kinds of folks. ... [Agile] is fun and effective, because not only do we not bog down the process in

Ariane 5 flight 501. On June 4, 1996, 37 seconds after liftoff, the rocket’s guidance system experienced a math overflow error: a floating point number was converted to a shorter integer, leading to an explosion at liftoff³. This exception could not have occurred on the slower Ariane 4 rocket for which the software was originally developed. As this incident shows, reusing software components without thorough system testing can be expensive: satellites worth \$370M were lost.

Agile is also known variously as a *lightweight* or *undisciplined* process.

Variants of Agile There are many variants of Agile software development (Fowler 2005). The one we use in this book is **Extreme Programming**, which is abbreviated **XP**, and credited to Kent Beck.

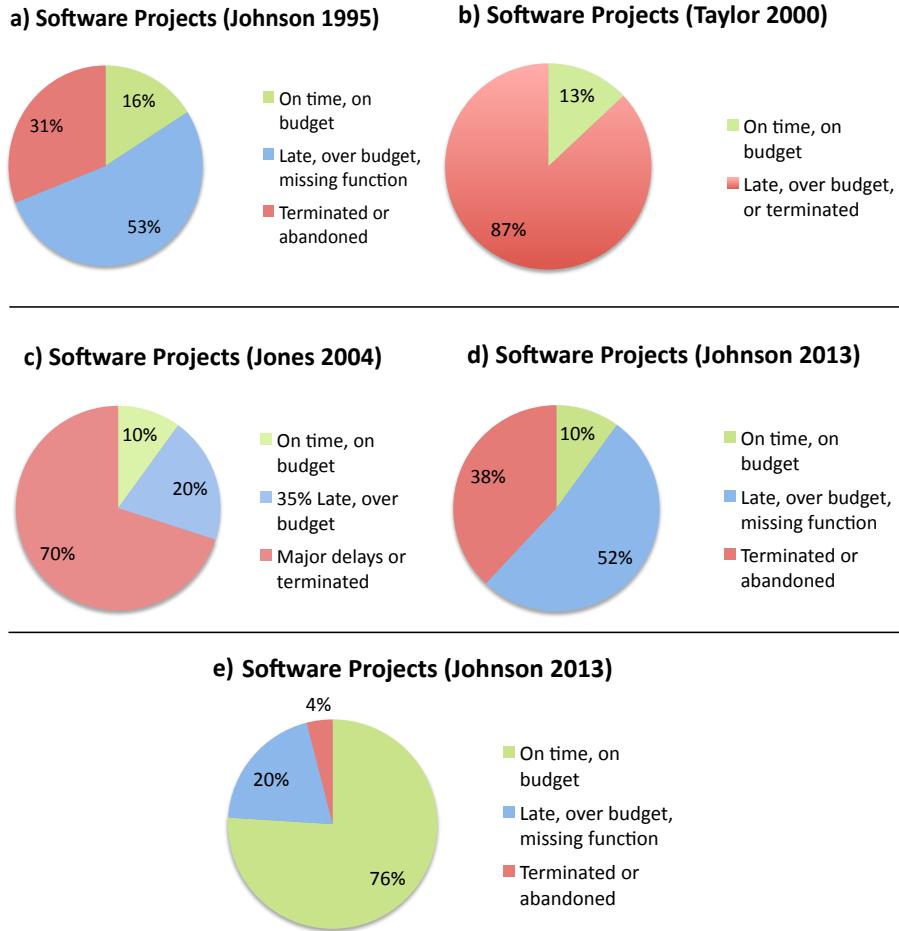


Figure 1.4: (a) A 1995 study of software projects found that 53% of projects exceeded their budgets and schedules by factors of 3, and another 31% were cancelled before completion (Johnson 1995). The estimated annual cost in the United States for such software projects was \$100B. (b) A 2000 survey of members of the British Computer Society found that only 130 of 1027 projects met their schedule and budget. Half of all projects were maintenance or data conversion projects and half new development projects, but the successful projects divided into 127 of the former and just 3 of the latter (Taylor 2000). (c) Survey of 250 large projects, each with the equivalent of more than a million lines of C code, found similarly disappointing results (Jones 2004). (d) The dismal outcomes for “large” (at least \$10M) projects in this survey of 50,000 projects (Johnson 2013b) suggest that HealthCare.gov had just a 10% chance of success. (e) Some good news: the “small” (under \$1M) projects in the same survey were largely completed on time and within budget, motivating the use of Agile.

mountains of documentation, but also because developers work face-to-face with clients throughout the development process and produce working software early on.

—Renee McCauley, “Agile Development Methods Poised to Upset Status Quo,”
SIGCSE Bulletin, 2001

By de-emphasizing planning, documentation, and contractually binding specifications, the Agile Manifesto ran counter to conventional wisdom of the software engineering intelligentsia, so it was not universally welcomed with open arms (Cormick 2001):

[The Agile Manifesto] is yet another attempt to undermine the discipline of software engineering... In the software engineering profession, there are engineers and there are hackers... It seems to me that this is nothing more than an attempt to legitimize hacker behavior... The software engineering profession will change for the better only when customers refuse to pay for software that doesn't do what they contracted for... Changing the culture from one that encourages the hacker mentality to one that is based on predictable software engineering practices will only help transform software engineering into a respected engineering discipline.

—Steven Ratkin, “Manifesto Elicits Cynicism,” *IEEE Computer*, 2001

One pair of critics even published the case against Agile as a 432-page book! (Stephens and Rosenberg 2003)

“The battle lines are drawn. Hostilities have broken out between armed camps of the software development community. This time the rallying cry is, “XP!” ... What XP uncovered (again) is an ancient, sociological San Andreas fault that runs under the software community—programming versus software engineering (a.k.a. the scruffy hackers versus the tweedy computer scientists).”

The software engineering research community went on to compare Plan-and-Document lifecycles to the Agile lifecycle in the field and found—to the surprise of some cynics—that Agile could indeed work well, depending on the circumstances. Figure 1.5 shows 10 questions from a popular software engineering textbook (Sommerville 2010) whose answers suggest when to use Agile and when to use Plan-and-Document methods.

While Figure 1.4(d) shows the disappointing results for large software projects, which do not use Agile, Figure 1.4(e) shows the success of small software projects—defined as costing less than \$1M—that typically do use Agile. With three-fourths of these projects on time, on budget, and with full functionality, the results are in stark contrast to the other charts in the figure. Success has fanned Agile’s popularity, and recent surveys peg Agile as the primary development method for 60% to 80% of all programming teams in 2013 (ET Bureau 2012, Project Management Institute 2012). One paper even found Agile was used by the majority of programming teams that are geographically distributed, which is much more difficult to pull off (Estler et al. 2012).

Thus, we concentrate on Agile in the six software development chapters in Part II of the book, but each chapter also gives the perspective of the Plan-and-Document methodologies on topics like requirements, testing, project management, and maintenance. This contrast allows readers to decide for themselves when each methodology is appropriate. Part I introduces SaaS and SaaS programming environments, including Ruby, Rails, and Javascript.

Agile is a family of methodologies, not a single methodology. We follow **Extreme Programming** (XP), which includes one- to two-week iterations, behavior driven design (see Chapter 7), test-driven development (see Chapter 8), and pair programming (Section 2.2). Another popular variant is **Scrum** (Section 10.1), where self-organizing teams use two- to

	Question: A no answer suggests Agile; a yes suggests Plan-and-Document
1	Is specification required?
2	Are customers unavailable?
3	Is the system to be built large?
4	Is the system to be built complex (e.g., real time)?
5	Will it have a long product lifetime?
6	Are you using poor software tools?
7	Is the project team geographically distributed?
8	Is team part of a documentation-oriented culture?
9	Does the team have poor programming skills?
10	Is the system to be built subject to regulation?

Figure 1.5: Ten questions to help decide whether to use an Agile lifecycle (the answer is no) or a Plan-and-Document lifecycle (the answer is yes) (Sommerville 2010). We find it striking that when asking these questions for projects done by student teams in a class, virtually all answers point to Agile. As this book attests, open source software tools are excellent, thus available to students (question 6). Our survey of industry (see Preface) found that graduating students do indeed have good programming skills (question 9). The other eight answers are clearly no for student projects.

four-week iterations called **sprints**, and then regroup to plan the next sprint. A key feature of many Agile methodologies is a daily standup meeting to identify and overcome obstacles. While there are multiple roles in the scrum team, the norm is to rotate the roles over time. The **Kanban** approach is derived from Toyota's just-in-time manufacturing process, which in this case treats software development as a pipeline. Here the team members have fixed roles, and the goal is to balance the number of team members so that there are no bottlenecks with tasks stacking up waiting for processing. One common feature is a wall of cards to illustrate the state of all tasks in the pipeline. There are also hybrid lifecycles that try to combine the best of two worlds. For example, *ScrumBan* uses the daily meetings and sprints of Scrum but replaces the planning phase with the more dynamic pipeline control of the wall of cards from Kanban.

While we now see how to build some software successfully, not all projects are small. We next show how to design software to enable composing smaller pieces into large services like Amazon.com.

Summary: In contrast to the Plan-and-Document lifecycles, the Agile lifecycle works with customers to continuously add features to working prototypes until the customer is satisfied, allowing customers to change what they want as the project develops. Documentation is primarily through user stories and test cases, and it does not measure progress against a predefined plan. Progress is gauged instead by recording **velocity**, which essentially is the rate that a project completes features.

■ Elaboration: Reforming Government Acquisition Regulations

President Obama belatedly recognized the difficulties of software acquisition. On November 14, 2013, he said in a speech: “...when I do some Monday morning quarterbacking on myself, one of the things that I do recognize is since I know how we purchase technology in the federal government is cumbersome, complicated and outdated ... it’s part of the reason why, chronically, federal IT programs are over budget, behind schedule... since I [now] know that the federal government has not been good at this stuff in the past, two years ago as we were thinking about this... we might have done more to make sure that we were breaking the mold on how we were going to be setting this up.”

Indeed, long before the ACA website, there were calls to reform software acquisition, as in this US National Academies study of the Department of Defense (DOD):

“The DOD is hampered by a culture and acquisition-related practices that favor large programs, high-level oversight, and a very deliberate, serial approach to development and testing (the waterfall model). Programs that are expected to deliver complete, nearly perfect solutions and that take years to develop are the norm in the DOD... These approaches run counter to Agile acquisition practices in which the product is the primary focus, end users are engaged early and often, the oversight of incremental product development is delegated to the lowest practical level, and the program management team has the flexibility to adjust the content of the increments in order to meet delivery schedules... Agile approaches have allowed their adopters to outstrip established industrial giants that were beset with ponderous, process-bound, industrial-age management structures. Agile approaches have succeeded because their adopters recognized the issues that contribute to risks in an IT program and changed their management structures and processes to mitigate the risks.” (National Research Council 2010)

Self-Check 1.3.1. *True or False: A big difference between Spiral and Agile development is building prototypes and interacting with customers during the process.*

- ◊ False: Both build working but incomplete prototypes that the customer helps evaluate. The difference is that customers are involved every two weeks in Agile versus up to two years in with Spiral. ■

Self-Check 1.3.2. *True or False: A big difference between Waterfall and Agile development is that Agile does not use requirements.*

- ◊ False: While Agile does not develop extensive requirements documents as does Waterfall, the interactions with customers lead to the creation of requirements as user stories, as we shall see in Chapter 7. ■

1.4 Software Quality Assurance: Testing

*And the users exclaimed with a laugh and a taunt:
“It’s just what we asked for, but not what we want.”*

—Anonymous

A standard definition of **quality** for any product is “fitness for use,” which must provide business value for both the customer and the manufacturer (Juran and Gryna 1998). For software, quality means both satisfying the customer’s needs—easy to use, gets correct answers, does not crash, and so on—and being easy for the developer to debug and enhance. **Quality Assurance (QA)** also comes from manufacturing, and refers to processes and standards

that lead to manufacture of high-quality products and to the introduction of manufacturing processes that improve quality. Software QA, then, means both ensuring that products under development have high quality and creating processes and standards in an organization that lead to high quality software. As we shall see, some Plan-and-Document software processes even use a separate QA team that tests software quality (Section 8.10).

Determining software quality involves two terms that are commonly interchanged but have subtle distinctions (Boehm 1979):

- **Verification:** Did you build the thing *right*? (Did you meet the specification?)
- **Validation:** Did you build the right *thing*? (Is this what the customer wants? That is, is the specification correct?)

Software prototypes that are the lifeblood of Agile typically help with validation rather than verification, since customers often change their minds on what they want once they begin to see the product work.

Infeasibility of exhaustive testing Suppose it took just 1 nanosecond to test a program and it had just one 64-bit input that we wanted to test exhaustively. (Obviously, most programs take longer to run and have more inputs.) Just this simple case would take 2^{64} nanoseconds, or 500 years!

The main approach to verification and validation is **testing**; the motivation for testing is that the earlier developers find mistakes, the cheaper it is to repair them. Given the vast number of different combinations of inputs, testing cannot be exhaustive. One way to reduce the space is to perform different tests at different phases of software development. Starting bottom up, **unit testing** makes sure that a single procedure or method does what was expected. The next level up is **module testing**, which tests across individual units. For example, unit testing works within a single class whereas module testing works across classes. Above this level is **integration testing**, which ensures that the interfaces between the units have consistent assumptions and communicate correctly. This level does not test the functionality of the units. At the top level is **system testing** or **acceptance testing**, which tests to see if the integrated program meets its specifications. In Chapter 8, we'll describe an alternative to testing, called **formal methods**.

As mentioned briefly in Section 1.3, the approach to testing for the XP version of Agile is to write the tests *before* you write the code. You then write the minimum code you need to pass the test, which ensures that your code is always tested and reduces the chances of writing code that will be later discarded. XP splits this test-first philosophy into two parts, depending on the level of the testing. For system, acceptance, and integration tests, XP uses *Behavior-Driven Design (BDD)*, which is the topic of Chapter 7. For unit and module tests, XP uses *Test-Driven Development (TDD)*, which is the topic of Chapter 8.

Summary: Testing reduces the risks of errors in designs.

- In its many forms, testing helps **verify** that software meets the specification and **validates** that the design does what the customer wants.
- To attack the infeasibility of exhaustive testing, we divide in order to conquer by focusing on **unit testing**, **module testing**, **integration testing**, and full **system testing** or **acceptance testing**. Each higher-level test delegates more detailed testing to lower levels.
- Agile attacks testing by writing the tests before writing the code, using either *Behavior Driven Design* or *Test Driven Design*, depending on the level of the test.

■ Elaboration: Testing: Plan-and-Document vs. Agile lifecycles

For the Waterfall development process, testing happens after each phase is complete and in a final verification phase that includes acceptance tests. For Spiral, it happens on each iteration, which can last one or two years. Assurance for the XP version of Agile comes from test-driven development, in that the tests are written *before* the code when coding from scratch. When enhancing existing code, test-driven design means writing the tests before writing the enhancements. The amount of testing depends on whether you are enhancing beautiful code or legacy code, with the latter needing a lot more.

Self-Check 1.4.1. While all of the following help with verification, which form of testing is most likely to help with validation: Unit, Module, Integration, or Acceptance?

- ◊ Validation is concerned with doing what the customer really wants versus whether code met the specification, so acceptance testing is most likely to point out the difference between doing the thing right and doing the right thing. ■

1.5 Productivity: Conciseness, Synthesis, Reuse, and Tools

Moore's Law meant hardware resources have doubled every 18 months for nearly 50 years. These faster computers with much larger memories could run much larger programs. To build bigger applications that could take advantage of the more powerful computers, software engineers needed to improve their productivity.

Engineers developed four fundamental mechanisms to improve their productivity:

1. Clarity via conciseness
2. Synthesis
3. Reuse
4. Automation via Tools

Clarity via conciseness reflects one of the driving assumptions of improving programmer productivity: if programs are easier to understand, they will have fewer bugs and be easier to maintain. A closely related corollary is that if the program is smaller, it's generally easier to understand. We capture this notion with our motto of “clarity via conciseness.”

Programming languages do this in two ways. The first is simply offering a syntax that lets programmers express ideas naturally and in fewer characters. For example, below are two ways to express a simple assertion:

- `assert_greater_than_or_equal_to(a, b)`
- `expect(a).to be >= b`

It's easy to imagine momentary confusion about the order of arguments in the first version in addition to the higher cognitive load of reading twice as many characters. The second version (which happens to be legal Ruby) is shorter and easier to read and understand, and will likely be easier to maintain.

The other way to improve clarity is to raise the level of abstraction. That initially meant the invention of higher-level programming languages such as Fortran and COBOL. This

step raised the engineering of software from assembly language for a particular computer to higher-level languages that could target multiple computers simply by changing the compiler.

As computer hardware performance continued to increase, more programmers were willing to delegate tasks to the compiler and runtime system that they formerly performed themselves. For example, Java and similar languages took over memory management from the earlier C and C++ languages. Scripting languages like Python and Ruby have raised the level of abstraction even higher. Examples are **reflection**, which allows programs to observe themselves, and **higher order functions**, which allows higher-level behaviors to be reused by passing functions as arguments to other functions. This higher level of abstraction made programs more concise and therefore (usually) easier to read, understand, and maintain. To highlight examples that improve productivity via conciseness, we use the “Concise” icon.

 **Synthesis** refers to code that is generated automatically rather than created manually. Logic synthesis for hardware engineers meant that they could describe hardware as Boolean functions and receive highly optimized transistors that implemented those functions. The classic software synthesis example is **Bit blit**. This graphics primitive combines two bitmaps under control of a mask. The straightforward approach would include a conditional statement in the innermost loop to choose the type of mask, but it was slow. The solution was to write a program that could synthesize the appropriate special-purpose code *without* the conditional statement in the loop. We’ll highlight examples that improve productivity by generating code with this “CodeGen” gears icon. The Rails framework makes extensive use of the Ruby language’s facilities for **metaprogramming**, which allows Ruby programs to automatically synthesize code at runtime.

 **Reuse** of portions from past designs, rather than writing everything from scratch, is a third way to improve productivity. As it is easier to make small changes in software than in hardware, software is even more likely than hardware to reuse a component that is almost but not quite a correct fit. We highlight examples that improve productivity via reuse with this “Reuse” recycling icon.

Procedures and functions were invented in the earliest days of software so that different parts of the program could reuse the same code with different parameter values. Standardized libraries for input/output and for mathematical functions soon followed, so that programmers could reuse code developed by others.

Procedures in libraries let you reuse implementations of individual tasks. But more commonly, programmers want to reuse and manage **collections** of tasks. The next step in software reuse was therefore **object-oriented programming** (OOP), where you could reuse the same tasks with different objects via the use of inheritance in languages like C++ and Java.

While inheritance supported reuse of implementations, another opportunity for reuse is a general strategy for doing something even if the implementation varies. **Design patterns**, inspired by work in civil architecture (Alexander et al. 1977), arose to address this need. Language support for reuse of design patterns includes **dynamic typing**, which facilitates composition of abstractions, and **mix-ins**, which offer ways to collect functionality from multiple methods without some of the pathologies of multiple inheritance found in some OOP languages. Python and Ruby are examples of languages with features that help with reuse of design patterns.

Note that reuse does *not* mean copying and pasting code so that you have very similar code in many places. The problem with copying and pasting code is that you may not change all the copies when fixing a bug or adding a feature. Here is a software engineering guideline

that guards against repetition:

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

—Andy Hunt and Dave Thomas, 1999

This guideline has been captured in the motto and acronym: ***Don't Repeat Yourself (DRY)***. We'll use a towel as the “DRY” icon to show examples of DRY in the following chapters.



Ruby and JavaScript, which we use in this book, are typical of modern scripting languages in including automatic memory management, dynamic typing, support for higher-order functions, and various mechanisms for code reuse. By including important advances in programming languages, Ruby goes beyond languages like Perl in supporting multiple programming paradigms such as object-oriented and ***functional programming***.

Automation, our fourth and final productivity-enhancing mechanism, reflects another core value of computer engineering: replacing tedious manual tasks with tools to save time, improve accuracy, or both. For software development, obvious tools include compilers and interpreters that raise the level of abstraction and generate code as mentioned above, but there are also more subtle productivity tools like Makefiles and version control systems (Section 10.2) that automate tedious tasks. We highlight tool examples with the hammer icon.



The tradeoff is always the time it takes to learn a new tool versus the time saved in applying it. Other concerns are the dependability of the tool, the quality of the user experience, and how to decide which one to use if there are many choices. Nevertheless, one of the software engineering tenets of faith is that a new tool can make our lives better.

Your authors embrace the value of automation and tools. That is why we show you several tools in this book to make you more productive. The good news is that any tool we show you will have been vetted to ensure its dependability and that time to learn will be paid back many times over in reduced development time and in the improved quality of the final result. For example, Chapter 7 shows how ***Cucumber*** helps automate turning user stories into integration tests and how ***Pivotal Tracker*** automatically measures ***Velocity***, which is a measure of the rate of adding features to an application. Chapter 8 introduces ***RSpec***, which helps automate the unit testing process. The bad news is that you'll need to learn several new tools. However, we think the ability to quickly learn and apply new tools is a requirement for success in engineering software, so it's a good skill to cultivate.

Learning new tools
Proverbs 14:4 in the King James Bible discusses improving productivity by taking the time to learn and use tools: *Where there are no oxen, the manger is clean; but abundant crops come by the strength of oxen.*



Thus, our fourth productivity enhancer is automation via tools. We highlight examples that use automation with the robot icon, although they are often also associated with tools.

Summary: Moore's Law inspired software engineers to improve their productivity by:

- Coveting conciseness, in using compact syntax and by raising the level of design by using higher-level languages. Examples include **reflection**, which allows programs to observe themselves at runtime, and **higher-order functions**, which allow higher-level behaviors to be reused by passing functions as arguments to other functions.
- Synthesizing implementations.
- Reusing designs by following the principle of **Don't Repeat Yourself (DRY)** and by relying upon innovations that help reuse, such as procedures, libraries, object-oriented programming, and design patterns.
- Using (and inventing) tools to automate tedious tasks.

■ Elaboration: Productivity: Plan-and-Document vs. Agile lifecycles

Productivity is measured in the engineer-hours to implement a new function. The difference is the cycles are much longer in Waterfall and Spiral vs. Agile—on the order of 6 to 24 months vs. 1/2 month—so much more work is done between releases that the customer sees, and hence the chances are greater that more work will ultimately be rejected by the customer.

Self-Check 1.5.1. Which mechanism is the weakest argument for productivity benefits of compilers for high-level programming languages: Clarity via conciseness, Synthesis, Reuse, or Automation and Tools?

◊ Compilers make high-level programming languages practical, enabling programmers to improve productivity via writing the more concise code in a HLL. Compilers do synthesize lower-level code based on the HLL input. Compilers are definitely tools. While you can argue that HLL makes reuse easier, reuse is the weakest of the four for explaining the benefits of compilers. ■

1.6 SaaS and Service Oriented Architecture

As the Web started to reach large audiences in the mid 1990s, a new idea started to emerge: rather than relying on users to install software on their computers, why not run the software centrally on Internet-based servers, and allow users to access it via a Web browser? Salesforce was arguably the first large company to fully embrace this new model, which was dubbed **Software as a Service (SaaS)**. Examples of SaaS that many of us now use every day include searching, social networking, and watching videos. But even apps such as word processing, contact management, and calendars, for which the previously dominant software delivery model was for users to install the software on their devices, have largely migrated to SaaS. The advantages of SaaS for both users and developers explain the popularity of SaaS:

SaaP (Software as a Product) is a retronym that appeared around 2015 to describe software that must be installed on each device when it's released or updated, in contrast to SaaS, in which the user is always using the latest version of a Web-based app.

1. Since customers do not need to install the application, they don't have to worry whether their hardware is the right brand or fast enough, nor whether they have the correct version of the operating system.

<i>SaaS Programming Framework</i>	<i>Programming Language</i>	<i>Introduced</i>
Active Server Pages (ASP.NET)	C#, VB.NET	1996
Enterprise Java Beans (EJB)	Java	1997
JavaServer Pages (JSP)	Java	1999
Spring	Java	2002
Rails	Ruby	2004
Django	Python	2005
Zend	PHP	2006
Sinatra	Ruby	2007

Figure 1.6: Examples of SaaS programming frameworks and the programming languages they are written in.

2. The data associated with the service is generally kept with the service, so customers need not worry about backing it up, losing it due to a local hardware malfunction, or even losing the whole device, such as a phone or tablet.
3. When a group of users wants to collectively interact with the same data, SaaS is a natural vehicle.
4. When data is large and/or updated frequently, it may make more sense to centralize data and offer remote access via SaaS.
5. Only a single copy of the server software runs in a uniform, tightly-controlled hardware and operating system environment selected by the developer. Although different Web browsers still have some incompatible behaviors (a topic we address in Chapter 6), developers overwhelmingly avoid the compatibility hassles of distributing binaries that must run on different users' computers.
6. Since the only copy of the server software is under the developers' control, they can upgrade the software and even the hardware as long as they don't violate the external application program interfaces (API), and they can pre-test new versions of the application on a small fraction of the real customers first, all without pestering users to upgrade their installed applications.
7. SaaS companies compete regularly on bringing out new features to help ensure that their customers do not abandon them for a competitor who offers a better service.

Given the popularity of SaaS, Figure 1.6 lists just a few of the many programming frameworks that claim to help create SaaS applications. In this book, we use the Rails framework written in the Ruby language ("Ruby on Rails"), although the ideas we cover will work with other programming frameworks as well. We chose Rails because it came from a community that had already embraced the Agile lifecycle, so the tools support Agile particularly well. If you are not already familiar with Ruby or Rails, this gives you a chance to practice an important software engineering skill: use the right tool for the job, even if it means learning a new tool or new language! Indeed, an attractive feature of the Rails community is that its contributors routinely improve productivity by inventing new tools to automate tasks that were formerly done manually.

Note that frequent upgrades of SaaS—due to only having a single copy of the software—perfectly align with the Agile software lifecycle. Hence, Amazon, eBay, Facebook, Google,

and other SaaS providers all rely on the Agile lifecycle, and traditional software companies like Microsoft are increasingly using Agile in their product development. The Agile process is an excellent match to the fast-changing nature of SaaS applications.

Despite all the advantages of SaaS, it was still missing one critical advantage in the area of software reuse. When creating SaaP, developers could make extensive use of software **libraries** containing code to perform tasks common to many different applications. Because these libraries were often written by others (so-called third-party libraries), they embodied the advantage of software reuse. By the mid 2000s, a similar phenomenon began to take shape in SaaS: the rise of **service-oriented architecture** (SOA), in which a SaaS service could call upon other services built and maintained by other developers for common tasks. Services that were highly specialized to a narrow range of tasks came to be called **microservices**; today's common examples include credit card processing, search, driving directions, and more. As standards solidified for representing and interacting with such external services, the important benefit of reuse finally arrived for SaaS. Chapter 3 delves into more detail about SOA and microservices.

Of course, we have yet to address one major difference between SaaS and SaaP: the underlying hardware on which the apps will run. With SaaP, that hardware consists of the PCs of millions of individual users. In the next section we explore the underlying hardware that makes SaaS possible.

Summary: **Software as a Service (SaaS)** is attractive to both customers and providers because the universal client (the Web browser) makes it easier for customers to use the service and the single version of the software at a centralized site makes it easier for the provider to deliver and improve the service. Given the ability and desire to frequently upgrade SaaS, the Agile software development process is popular for SaaS, and so there are many frameworks to support Agile and SaaS. This book uses Ruby on Rails.

Self-Check 1.6.1. *Some of Google's most popular SaaS apps are Search, Maps, Gmail, Calendar, and Documents. For each of these apps, give one advantage of delivering the app as SaaS rather than SaaP.*

- ◊ Many answers are correct, but here are ours:
 1. No user installation: Documents
 2. Can't lose data: Gmail, Calendar.
 3. Users cooperating: Documents.
 4. Large/changing datasets: Search, Maps, YouTube.
 5. Software centralized in single environment: Search.
 6. No field upgrades when improve app: Documents.

■

Self-Check 1.6.2. *True or False: If you are using the Agile development process to develop SaaS apps, you could use Python and Django or languages based on the Microsoft's .NET framework and ASP.NET instead of Ruby and Rails.*

- ◊ True. Programming frameworks for Agile and SaaS include Django and ASP.NET. ■

1.7 Deploying SaaS: Cloud Computing

If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility ... The computer utility could become the basis of a new and important industry.

—John McCarthy, at MIT centennial celebration in 1961

SaaS places three demands on our information technology (IT) infrastructure:

1. Communication, to allow any customer to interact with the service.
2. Scalability, in that the central facility running the service must deal with the fluctuations in demand during the day and during popular times of the year for that service as well as a way for new services to add users rapidly.
3. Availability, in that both the service and the communication vehicle must be continuously available: every day, 24 hours a day (“24×7”). The gold standard for availability, set by the US public phone system, is 99.999% (“five nines”), or about 5 minutes of downtime per year. Amazon.com aims for four nines, which is difficult to achieve even for well-run SaaS.

The Internet and broadband to the home easily resolve the communication demand of SaaS. Although some early web services were deployed on expensive large-scale computers—in part because such computers were more reliable and in part because it was easier to operate a few large computers—a contrarian approach soon overtook the industry. Collections of commodity small-scale computers connected by commodity Ethernet switches, which became known as **clusters**, offered several advantages over the “big iron” hardware approach:

- Because of their reliance on Ethernet switches to interconnect, clusters are much more scalable than conventional servers. Early clusters offered 1000 computers, and today’s datacenters contain 100,000 or more.
- Careful selection of the type of hardware to place in the datacenter and careful control of software state made it possible for a very small number of operators to successfully run thousands of servers. In particular, some datacenters rely on **virtual machines** to simplify operation. A virtual machine monitor is software that imitates a real computer so successfully that you can even run an operating system correctly on top of the virtual machine abstraction that it provides (Popek and Goldberg 1974). The goal is to imitate with low overhead, and one popular use is to simplify software distribution within a cluster. In this way, multiple apps can share hardware with each app even believing it has its own copy of the operating system. If the apps are also able to share the operating system, an even more efficient way to share hardware is to use **OS-level virtualization**; the popular tool Docker, which allows each app to run in its own *container* on a shared OS, is one example.
- Two senior architects at Google showed that the cost of the equivalent amount of processors, memory, and storage is much less for clusters than for “big iron,” perhaps by a factor of 20 (Barroso and Hoelzle 2009).

John McCarthy
 (1927–2011) received the Turing Award in 1971 and was the inventor of Lisp and a pioneer of timesharing large computers. Clusters of commodity hardware and the spread of fast networking have helped make his vision of timeshared “utility computing” a reality.



- Although the cluster components are less reliable than conventional servers and storage systems, the cluster software infrastructure makes the whole system dependable via extensive use of redundancy in both hardware and software. The low hardware cost makes the redundancy at the software level affordable. Modern service providers also use multiple datacenters that are distributed geographically so that a natural disaster cannot knock a service offline.

Luiz Barroso, VP of Engineering at Google and winner of the 2020 ACM/IEEE Eckert-Mauchly Award, gives an excellent brief history of warehouse-scale computing at the beginning of his award acceptance speech⁴.

As Internet datacenters grew, some service providers realized that their per capita costs were substantially below what it cost others to run their own smaller datacenters, in large part due to economies of scale when purchasing and operating 100,000 computers at a time. They also benefit from higher utilization given that many companies could share these giant datacenters, which (Barroso and Hoelzle 2009) call *Warehouse Scale Computers*, whereas smaller datacenters often run at only 10% to 20% utilization. Thus, these companies realized they could profit from making their datacenter hardware available on a pay-as-you-go basis.

The result is called *public cloud services*, **utility computing**, or often simply **cloud computing**, which offers computing, storage, and communication at pennies per hour (Armbrust et al. 2010). Moreover, there is no additional cost for scale: Using 1000 computers for 1 hour costs no more than using 1 computer for 1000 hours. Leading examples of “infinitely scalable” pay-as-you-go computing are Amazon Web Services, Google AppEngine, and Microsoft Azure. The public cloud means that today anyone with a credit card and a good idea can start a SaaS company that can grow to millions of customers without first having to build and operate a datacenter.

From 2010–2020, Cloud Computing and SaaS began a major transformation of the computer industry. The full impact of this revolution will take the rest of this decade to determine. What is clear is that engineering SaaS for Cloud Computing is radically different from engineering shrink-wrap software (SaaP) for PCs and servers, which is why you’re reading this book.

FarmVille had 1 million players within 4 days after it was announced, 10 million after 2 months, and 75 million after 9 months. (The prior record for number of users of a social networking game was 5 million.) Fortunately, FarmVille used the Elastic Compute Cloud (EC2) from Amazon Web Services, and kept up with its popularity by simply paying to use larger clusters.

Summary

- The Internet supplies the communication for SaaS.
- Cloud Computing** provides the scalable and dependable hardware computation and storage for SaaS.
- Cloud computing consists of **clusters** of commodity servers that are connected by local area network switches, with a software layer providing sufficient redundancy to make this cost-effective hardware dependable.
- These large clusters or *Warehouse Scale Computers* offer economies of scale.
- Taking advantage of economies of scale, some Cloud Computing providers offer this hardware infrastructure as low-cost **utility computing** that anyone can use on a pay-as-you-go basis, acquiring resources immediately as your customer demand grows and releasing them immediately when it drops.

Self-Check 1.7.1. True or False: Internal datacenters could get the same cost savings as *Warehouse Scale Computers* (WSCs) if they embraced SOA and purchased the same type of

hardware.

- ◊ False. While imitating best practices of WSC could lower costs, the major cost advantage of WSCs comes from the economies of scale, which today means 100,000 servers, thereby dwarfing most internal datacenters. ■

1.8 Deploying SaaS: Browsers and Mobile

Beginning around 1994, the stunning success of the Web quickly led to the phasing-out of many SaaP desktop apps. The proprietary client UIs of fee-based services such as AOL and CompuServe were replaced by free Web-based portals such as Yahoo!. Specialized SaaP apps for accessing Internet-based services, such as Eudora for email, were replaced by browser-based email such as Hotmail. Even productivity apps such as Microsoft Word began to feel pressure from browser-based competitors such as Google Docs. The browser thus became a *universal client*: any site the browser visited could deliver all the information necessary to render that site's user interface using HTML, the Hypertext Markup Language. As we'll see, JavaScript entered the picture later as a way to enrich the interactive experience of Web pages, but the actual visible page content always consists of HTML.

As its name implies, HTML is an example of a **markup language**: it combines text with markup (annotations about the text's structure) in a way that makes it easy to syntactically distinguish the two. Technically, HTML 5 (the current widely-used version) is really just one type of document that can be expressed in **XML**, an eXtensible Markup Language that can be used both to represent data and to describe other markup languages.

Separating an HTML document's *logical structure* from its *appearance* confers many benefits. Structure refers to the kind of content each logical component of a page represents, such as a major or minor heading, a bulleted list, a paragraph of text, and so on. Some components are simple, such as a page title or a dropdown menu of choices, and correspond to an HTML element with no children. More often, a component consists of an HTML `div` element with other elements nested inside it. `divs` often group together logically-related elements, and may be nested. Appearance refers not only to basic typography such as fonts and colors, but to the layout of elements on a page. For example, a navigation menu that is best displayed as a set of horizontal tabs on a full-size screen may work better if displayed as a drop-down menu on a mobile phone screen, even though the menu choices and their meanings are the same.

The key to separating structure and appearance is the use of **Cascading Style Sheets** (CSS), introduced in 1996 as a way to associate visual rendering information with HTML elements. The key concept of CSS is that of a **selector**—an expression that matches one or more HTML elements in a document. Even if you're not the developer who will be in charge of visual appearance, understanding CSS selectors is important because, as we will see in Chapter 6, selectors are a key mechanism used by JavaScript frameworks such as jQuery that allow you to create rich interactive Web pages. While there are multiple ways a selector can match an element, by far the most widely used is to associate the selector with an element's `class` attribute, since multiple elements of the same or different types on a page can share the same class attribute(s). Figure 1.7 shows a very simple HTML page. The basic mechanism of using CSS to “style” HTML is as follows:

As a reminder, the Concepts & Prerequisites page suggests self-study materials for basic HTML and CSS.

1. When the browser loads the page, it looks for one or more `link` elements, which should

```
https://gist.github.com/edb6a189f7892a17b0bc06f0ec2e6d34
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <link rel="stylesheet" href="https://getbootstrap.com/docs/4.0/dist/css/
5              bootstrap.min.css">
6          <title>Dietary Preferences of Penguins</title>
7      </head>
8      <body>
9          <div class="container">
10             <h1>Introduction</h1>
11             <p class="lead">
12                 This article is a review of the book
13                 <i>Dietary Preferences of Penguins</i>,
14                 by Alice Jones and Bill Smith. Jones and Smith's controversial work
15                 makes three hard-to-swallow claims about penguins:
16             </p>
17             <ul class="list-group">
18                 <li class="list-group-item">
19                     First, that penguins actually prefer eating tropical foods to fish
20                 </li>
21                 <li class="list-group-item">
22                     Second, that eating tropical foods makes them smell unattractive to
23                         predators
24                 </li>
25             </ul>
26         </div>
27     </body>
28 </html>
```

Figure 1.7: At its simplest, an HTML 5 document is a file of text beginning with the XML *document type declaration*, followed by a single `html` element whose child elements represent the components on the page. The use of angle brackets for HTML tags comes from *SGML* (Standard Generalized Markup Language), a codified standardization of IBM's *Generalized Markup Language*, developed in the 1960s for encoding computer-readable project documents.

be children of the HTML document's `head` element, that specify stylesheets to be used in conjunction with this document. In this case, the link's `href` (target) refers to the main stylesheet of Bootstrap CSS, which we discuss next.

2. The browser loads each referenced CSS stylesheet. A stylesheet contains a set of selectors and, for each selector, a set of rules for how to display elements matching that selector. These rules can specify typography, layout on the page, colors, and more.
3. When displaying the page, the browser matches up the CSS rules with the matching elements on each displayed page. In this example, Bootstrap provides basic style rules for each element type (`h1`, `p`, `ul`, and so on), and the `class` attributes on various elements are there to match particular CSS selectors in Bootstrap that further "tweak" the formatting of specific page elements.

While CSS syntax is simple, creating visually appealing stylesheets requires graphic design and typography skills. Those of us who lack those skills are better served by using existing stylesheets designed by professionals, collections of which are sometimes referred to as *CSS frameworks*, or more commonly, *front-end frameworks* if they also include JavaScript code to further enhance visual effects by adding animations, fades, and so on that are impossible using CSS alone. A widely-used front-end framework to which we'll refer throughout the book is Bootstrap, an open source project contributed by Twitter. A good CSS framework provides at least four main benefits:

1. A set of high-level components that combine multiple HTML elements into a logical unit. For example, a navigation menu with dropdowns can be managed as a single component, even though it includes many HTML elements.
2. A grid metaphor for specifying the layout of components on a page. In the case of Bootstrap, the page is divided into 12 columns, and any component can be specified to span any number of columns, with the same component having different layout instructions for smaller vs. larger screens.
3. **Responsive** behavior on a variety of display sizes. For example, a navigation menu that normally displays as a set of horizontal tabs will be automatically rendered as vertically-stacked choices when the display is too small, even if you haven't explicitly provided such instructions. The page <https://getbootstrap.com/docs/4.0/examples/navbars/>⁵ shows examples of how navigation bars in Bootstrap behave as the screen is resized.
4. Support for accessibility for users with disabilities, such as by providing styles for content that should be visually hidden but remain accessible to assistive technologies such as screen readers.

The **CSS Zen Garden** shows how dramatically different the same HTML content can be made to look with different CSS stylesheets.

HTML/CSS frameworks have become particularly important with the takeover of mobile devices. Although Apple's introduction of the iPhone in 2007 was definitely not the first **smartphone** to feature installable apps or Web browsing, it was the first to become wildly successful and widely copied. By 2017, just ten years later, about a third of the world's population had smartphones, and these accounted for more visits to Web sites than desktops or laptops Enge 2018. Up to a point, carefully-designed CSS styles can make the same HTML content usable on a wide range of screen sizes. For this reason, while Figure 1.8

Advantages	Disadvantages
Mobile-first/responsive Web site (HTML, CSS, JavaScript)	
Use same languages, tools, and framework as desktop SaaS Portable across devices, so no need to develop/maintain multiple versions User never needs to install updates Can be made to work even when disconnected from the Internet ⁷ Icon placement on user's home screen	Not listed in app stores May lack access to advanced platform hardware features Depending on app complexity, performance may be noticeably lower than native app
“Wrapped” app	
Same benefits as mobile website approach except for zero-install updates Can be listed in app stores	Must rely on users to download and install updates Additional software framework such as PhoneGap required to “package” app for distribution
Native app for Android (Java) or iOS (Objective-C)	
Best performance Can be listed in app stores Guaranteed access to all platform hardware features	Must install and learn new platform, development environment, testing framework, and deployment pipeline Must rely on users to download and install updates in a timely way, and commit to supporting old versions until they do Supporting multiple platforms requires maintaining multiple codebases

Back to SaaP? The proliferation of native apps has rolled back a major benefit of SaaS—users must once again manually update their apps when security bugs are found or when they upgrade their devices. And updates for mobile apps are far more frequent than they ever were for SWS—in 2014, the Twitter mobile app was updated about every 20 days on average.⁶

Figure 1.8: Three approaches to implementing mobile clients. Today, the vast majority of mobile platform features, including disconnected operation and fingerprint-based authentication, are now available via open Web standards as well as via the native programming environment.

shows that there are many approaches to developing client apps, in this book we recommend creating “mobile-first” apps using HTML5, CSS, and JavaScript, thus taking advantage of the extensive existing tooling available in that ecosystem, especially the Bootstrap presentation framework and the jQuery DOM library, which we discuss in Chapter 6.

Despite the differences among the ways of building mobile or desktop SaaS, all such apps are structurally similar: they provide a local user interface, possibly including local storage, and they use open SaaS standards and protocols to communicate with one or more remote servers.

Summary

- An **HTML** (HyperText Markup Language) document consists of a hierarchically nested collection of elements. Each element begins with a **tag** in <angle brackets> that may have optional **attributes**. Some elements enclose content. In general, the elements describe the logical structure of the parts of the document, but not how the document should appear when rendered on a screen.
- **Cascading Style Sheets** (CSS) is a stylesheet language describing visual attributes of elements on a Web page. A stylesheet associates sets of visual properties with selectors that match one or more page elements. There are many ways to express selectors that match different elements, but the most common is to associate one or more CSS classes with the element and write selectors that match elements based on class.
- CSS pages are separate from HTML documents, and `link` element(s) inside the `head` element of an HTML document associate one or more stylesheets with that document.
- Mobile devices now account for the majority of visits to SaaS apps. One way to build “mobile-first” client apps that work well on smartphones is to use CSS frameworks such as Bootstrap. These frameworks provide different sets of CSS formatting rules for the same HTML elements depending on the kind of device on which the HTML is being viewed.
- Another way to build “mobile-first” client apps is to create so-called “native” installable smartphone apps. Native apps may allow access to some device features unavailable from HTML, but they also negate important SaaS advantages such as eliminating the need to install updates or maintain a separate codebase for each mobile device.

Self-Check 1.8.1. *How would you ensure the same CSS stylesheet(s) are used for all pages in your site or your app?*

- ◊ Each individual HTML document must include its own stylesheet links, so you’d ensure that the same `<link>` element appears within the `<head>` of each page of your site or app. ■

1.9 Beautiful vs. Legacy Code

To me programming is more than an important practical art. It is also a gigantic undertaking in the foundations of knowledge.

—Grace Murray Hopper

Unlike hardware, software is expected to grow and evolve over time. Whereas hardware designs must be declared finished before they can be manufactured and shipped, initial software designs can easily be shipped and later upgraded over time. Basically, the cost of upgrade in the field is astronomical for hardware and affordable for software.

Hence, software can achieve a high-tech version of immortality, potentially getting better over time while generations of computer hardware decay into obsolescence. The drivers of

Grace Murray Hopper
(1906–1992) was one of the first programmers and developed the first compiler. “Amazing Grace” became a Rear Admiral in the US Navy, and in 1997, a warship was named for her, the USS Hopper.



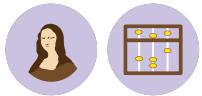
software evolution are not only fixing faults, but also adding new features that customers request, adjusting to changing business requirements, improving performance, and adapting to a changed environment. Software customers expect to get notices about and install improved versions of the software over the lifetime that they use it, perhaps even submitting bug reports to help developers fix their code. They may even have to pay an annual maintenance fee for this privilege!

The Oldest Living Program might be MOCAS⁸ (“Mechanization of Contract Administration Services”), which was originally purchased by the US Department of Defense in 1958 and was still in use as of 2005.

Just as novelists fondly hope that their brainchild will be read long enough to be labeled a classic—which for books is 100 years!—software engineers should hope their creations would also be long lasting. Of course, software has the advantage over books of being able to be improved over time. In fact, a long software life often means that others maintain and enhance it, letting the creators of original code off the hook.

This brings us to a few terms we’ll use throughout the book. The term **legacy code** refers to software that, despite its old age, continues to be used because it meets customers’ needs. Sixty percent of software maintenance costs are for adding new functionality to legacy software, vs. only 17% for fixing bugs, so legacy software is successful software.

The term “legacy” has a negative connotation, however, in that it indicates that the code is difficult to evolve because it has an inelegant design or uses antiquated technology. In contrast to legacy code, we use the term **beautiful code** to indicate long-lasting code that is easy to evolve. The worst case is not legacy code, however, but *unexpectedly short-lived code* that is soon discarded because it doesn’t meet customers’ needs. We’ll highlight examples that lead to beautiful code with the Mona Lisa icon. Similarly, we’ll highlight text that deals with legacy code using an abacus icon, which is certainly a long-lasting but little changed calculating device.



Abacuses are still in use today in many parts of the world despite being thousands of years old.

In the following chapters, we show examples of both beautiful code and legacy code that we hope will inspire you to make your designs simpler to evolve. Surprisingly, despite the widely accepted importance of enhancing legacy software, this topic is traditionally ignored in college courses and textbooks. We feature such software in this book for three reasons. First, you can reduce the effort to build a program by finding existing code that you can reuse. One supplier is open source software. Second, it’s advantageous to learn how to build code that makes it easier for successors to enhance, since such code is more likely to enjoy a long life. Finally, unlike Plan-and-Document, in Agile you revise code continuously to improve the design and to add functionality starting with the second iteration. Thus, the skills you practice in Agile are exactly the ones you need to evolve legacy code—no matter how it was created—and the dual use of Agile techniques makes it much easier for us to cover legacy code within a single book.

Summary: Successful software can live decades and is expected to evolve and improve, unlike computer hardware that is finalized at time of manufacture and can be considered obsolete within just a few years. One goal of this book is to teach you how to increase the chances of producing beautiful code so that your software lives a long and useful life.

Self-Check 1.9.1. *Programmers rarely set out to write bad code. Given the ideas of Section 1.5 about productivity, explain briefly how software written a long time ago that was considered high quality at the time might be viewed as difficult-to-maintain legacy software today.*

- ◊ Because of the continuously increasing level of abstraction of software tools, developers today can often create the same functionality in far fewer (and more beautiful) lines of code

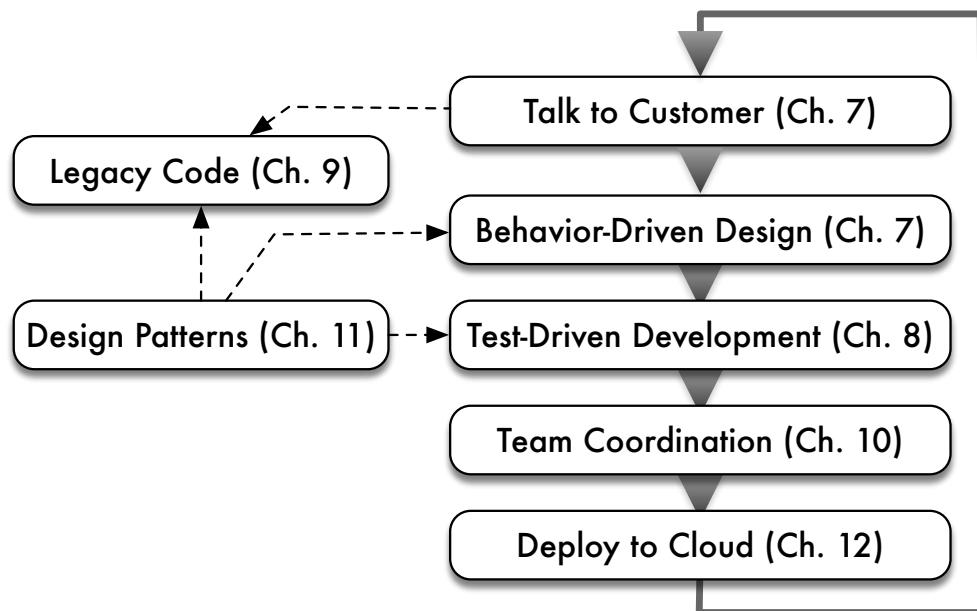


Figure 1.9: An iteration of the Agile software lifecycle and its relationship to the chapters in Part II of this book. The dashed arrows indicate a more tangential relationship between the steps of an iteration, while the solid arrows indicate the typical flow. As mentioned earlier, the Agile process applies equally well to legacy applications and new applications.

than would have been possible a few decades ago, so by comparison the old code is harder to maintain, even though at the time it was written it may have represented the state of the art. Doubtless the code we write today will be viewed as archaic in another few decades! ■

1.10 Guided Tour and How To Use This Book

As this chapter’s Concepts and Prerequisites described, becoming a skilled software engineer requires *both* conceptual understanding and plenty of hands-on practice. Therefore, our goal in each chapter is to give you the necessary conceptual foundations to work on the exercises, where the real learning happens.

The rest of the book is divided into two parts. Part I explains Software as a Service, and Part II explains modern software development, with a heavy emphasis on Agile.

Chapter 3 starts Part I with an explanation of the architecture of a SaaS application, and how the Web went from a collection of static pages to an ecosystem of services characterized by RESTful APIs—that is, Application Programming Interfaces based on the design stance of Representational State Transfer.

Since languages and frameworks evolve rapidly, we believe *learning how to learn* new languages and frameworks is a more valuable skill than knowing a specific language or framework. Thus, Chapter 2 introduces our methodology for doing so, using Ruby as an example, for programmers already familiar with another modern language such as Java or Python.

Similarly, today the main reason for learning a new language is often the desire to use a framework that relies on that language. A good framework both reifies a particular application architecture and takes advantage of the features of a particular language to make

development easy when it conforms to that architecture. Chapter 4 introduces the basics of Rails and its central metaphor of the Model–View–Controller architecture. Chapter 5 covers more advanced Rails features and shows in more depth how Rails takes advantage of Ruby’s language features. Splitting the material this way supports readers who want to get started writing an app as soon as they can, which just requires Chapter 4. Readers already familiar with Ruby and Rails may want to skim or skip these chapters.

Using the same strategy for learning new languages and frameworks, Chapter 6 introduces JavaScript, the jQuery framework, and the Jasmine testing tool. The Jasmine discussion assumes knowledge of testing, so readers may prefer to read that material after Chapter 8. Just as Rails amplifies the power and productivity of Ruby for SaaS servers, jQuery amplifies the power and productivity of JavaScript for the client.

Given this background, the next six chapters of Part II illustrate important software engineering principles using Rails tools to build and deploy a SaaS app. Figure 1.9 shows one iteration of the Agile lifecycle, which we use as a framework on which to hang the next chapters of the book.

Chapter 7 discusses how to work with the customer. ***Behavior-Driven Design (BDD)*** advocates writing ***user stories*** describing application use cases in terms that nontechnical customers can understand, and Chapter 7 shows how to turn user stories into integration and acceptance tests, using the ***Cucumber*** tool. The chapter also explains how ***velocity*** can be used to measure progress in delivering features, and introduces the ***Pivotal Tracker*** tool to track and calculate velocity.

Chapter 8 covers ***Test-Driven Development (TDD)***. The chapter demonstrates how to write good, testable code and introduces the ***RSpec*** testing tool for writing unit tests, the ***Guard*** tool for automating test running, and the ***SimpleCov*** tool to measure test coverage.

Chapter 9 describes how to deal with existing code, including how to enhance legacy code. Helpfully, it shows how to use BDD and TDD to both understand and refactor code and how to use the Cucumber and RSpec tools to make this task easier.

Chapter 10 gives advice on how to organize and work as part of an effective team using the ***Scrum*** principles mentioned above. It also describes how the version control system ***Git*** and the corresponding service ***GitHub*** can let team members work on different features without interfering with each other or causing chaos in the release process.

To help you practice Don’t Repeat Yourself, Chapter 11 introduces design patterns, which are proven structural solutions to common problems in designing how classes work together, and shows how to exploit Ruby’s language features to adopt and reuse the patterns. The chapter also offers guidelines on how to write good classes. It introduces just enough ***UML (Unified Modeling Language)*** notation to help you notate design patterns and to help you make diagrams that show how the classes should work.

Note that Chapter 11 is about software architecture whereas prior chapters in Part II are about the Agile development process. We believe in a college course setting that this order will let you start an Agile iteration sooner, and we think the more iterations you do, the better you will understand the Agile lifecycle. However, as Figure 1.9 suggests, knowing design patterns will be useful when writing or refactoring code, since it is fundamental to the BDD/TDD process.

Chapter 12 offers practical advice on how to first deploy and then improve performance and scalability in the cloud, and briefly introduces some reliability and security techniques that are uniquely relevant to deploying SaaS.

We conclude with an Afterword that reflects on the material in the book and projects what



might be next.

CHIPS. As Confucius said: “I hear and I forget, I see and I remember, I do and I understand.” The goal of the book is to give you just enough content to get a conceptual handle on the Coding/Hands-On Integrated Projects (CHIPS) interspersed with the text. Each CHIPS exercise contains significant guidance and hints for the self-learning you’ll have to do to complete it. If you’re using this book in conjunction with online course materials from Codio (either in a classroom setting, in self-learning, or in the edX course sequence), switching between the content-oriented didactic material (COD) and the coding/hands-on integrated projects (CHIPS) is especially easy, and your assignments will be automatically graded for you. Instructors and self-learners, please see www.saasbook.info for more information on all of these options.

Terminology. You will encounter many new technical terms (and buzzwords) as you dive into this rich ecosystem. To help you identify important terms, text formatted *like this* refers to terms with corresponding Wikipedia entries. (In the Kindle book, PDF document, and Codio book, the terms link to the appropriate Wikipedia page.) Depending on your background, we suspect you’ll need to read some chapters more than once before you get the hang of it.

Each chapter concludes with a section called *Fallacies and Pitfalls*, which explains common misconceptions or problems that are easy to experience if you’re not vigilant, and Concluding Remarks to provide resources for those who want to dig more deeply into some of the chapter’s concepts.

Summary:

- Software engineering can only be learned by doing, and learning by doing is not about following a recipe or cutting and pasting code. The text in this book (COD, or content-oriented didactics) gives you the conceptual foundation to work on the CHIPS (coding/hands-on integrated projects). Both are essential to learning the material.
- If you’re using the book in conjunction with the Codio IDE (either in your classroom, on your own, or in the edX courses), the programming assignments are automatically graded for you and all necessary software is preinstalled.
- Each chapter begins with a list of the big ideas of that chapter and the prerequisite knowledge for the chapter.
- Don’t skip the Fallacies & Pitfalls! Even experts run into them, which is why they get a section to themselves in each chapter.

Self-Check 1.10.1. Which is most important for rapidly learning SaaS development: understanding the conceptual foundations, reading code, or writing code?

◊ All are important. You won’t learn much by copying-and-pasting code if you don’t understand why it works (or doesn’t). On the other hand, just reading *about* code doesn’t get anything working. Inspecting others’ high-quality code, which we hope your instructors will emphasize, not only shows you good examples but also helps cement your understanding of the conceptual foundations.

■ 1.11 Fallacies and Pitfalls

Lord, give us the wisdom to utter words that are gentle and tender, for tomorrow we may have to eat them.

—Sen. Morris Udall

As mentioned above, this section near the end of a chapter explains ideas of a chapter from another perspective, and gives readers a chance to learn from the mistakes of others. *Fallacies* are statements that seem plausible (or are actually widely held views) based on the ideas in the chapter, but they are not true. *Pitfalls*, on the other hand, are common dangers associated with the topics in the chapter that are difficult to avoid even when you are warned.



Fallacy: The Agile lifecycle is best for all software development.

Agile is a nice match to many types of software, particularly SaaS, which is why we use it in this book. However, Agile is *not* best for everything. Agile may be ineffective for safety-critical apps, for example.

Our experience is that once you learn the classic steps of software development and have a positive experience in using them via Agile, you will use these important software engineering principles in other projects no matter which methodology is used. Each chapter in Part II concludes with contrasting Plan-and-Document perspective to help you understand these principles and to help you use other lifecycles should the need arise.

Nor will Agile be the last software lifecycle you will ever see. We believe that new development methodologies develop and become popular in response to new opportunities, so expect to learn new methodologies and frameworks in your future.



Pitfall: Ignoring the cost of software design.

Since there is essentially no cost to distribute software, the temptation is to believe there is almost no cost to changing it so that it can be “remanufactured” the way the customer wants. However, this perspective ignores the cost of design and test, which can be a substantial part of the overall costs for software projects. Zero manufacturing cost is also one rationalization used to justify pirating copies of software and other electronic data, since pirates apparently believe no one should pay for the cost of development, just for manufacturing.



Pitfall: Ignoring the historical context of software technology.

Those who cannot remember the past are condemned to repeat it.

—George Santayana

Software engineering is a relatively young engineering field, but a fast-moving one. If you try to learn software technologies while ignoring the historical context in which they arose, you risk making underinformed choices about what tools to use, or worse, “reinventing the wheel” without learning from the experiences of others. For example, if you’re debating

with colleagues about the advisability of using Node.js as your application server, but you are unfamiliar with the long-running “threads vs. events” debates in the systems software community, at best you will be having an under-informed discussion, and at worst you will be quickly beset by woe. Similarly, the feature creep of “NoSQL” databases mirrors the progression of events that led to the invention of the ***relational model*** and its eventual dominance over the older ***hierarchical database model***, which “baseline” NoSQL databases strongly resemble. Reinventing the wheel isn’t always necessarily a bad thing. Sometimes the existing wheel really isn’t a great fit for your needs—as Douglas Crockford is said to have remarked, “The good thing about reinventing the wheel is that you can get a round one.” Our hope is that learners of this material will choose to take a few extra minutes to gain a broader perspective on *why* various things are the way they are (or not). We believe this will not only help you decide whether a particular wheel reinvention is a good one, but also help you avoid techno-fetishism—the belief that a new “rockstar” technology is important and worth learning simply because it’s new (or fast, or lean, or whatever), without a well-grounded perspective of its strengths and weaknesses or of how it builds on ideas that have been explored previously.



Pitfall: Being overly focused on learning framework X as rapidly as possible.

Possible values of *X* change so quickly that in any given leap year, the “new hot tech” for building software is probably different from what it was during the previous leap year. Indeed, since the first edition of this book in 2013, “hot tech” for building front-end apps has changed from Prototype.js to jQuery to Angular to Ember to Backbone to React, with Vue now another contender. Therefore, your authors believe that it’s more valuable to *learn how to learn* new languages and frameworks, by understanding the fundamental principles of software architecture and design on which they’re built, by continuously acquiring fluency in multiple frameworks and tools, and by adopting an ecumenical approach to the question of “which language or framework is best” for a given project.

1.12 Concluding Remarks: Software Engineering Is More Than Programming

The Concluding Remarks at the end of each chapter give the learner some perspective on what the chapter has covered: Where did the technical ideas or innovations come from? What, if anything, can we say about where they are going, given that history? Where can an interested reader learn more about these topics? These sections never contain specific technical skill content, so if you’re in a rush, you can skip them; but if you want to become a seasoned practitioner and a good designer of software and tools, you probably shouldn’t.

But if Extreme Programming is just a new selection of old practices, what’s so extreme about it? Kent’s answer is that it takes obvious, common sense principles and practices to extreme levels. For example:

- *If short iterations are good, make them as short as possible—hours or minutes or seconds rather than days or weeks or years.*
- *If simplicity is good, always do the simplest thing that could possibly work.*
- *If testing is good, test all the time. Write the test code before you write the code to test.*
- *If code reviews are good, review code continuously, by programming in pairs, two programmers to a computer, taking turns looking over each other’s shoulders.*

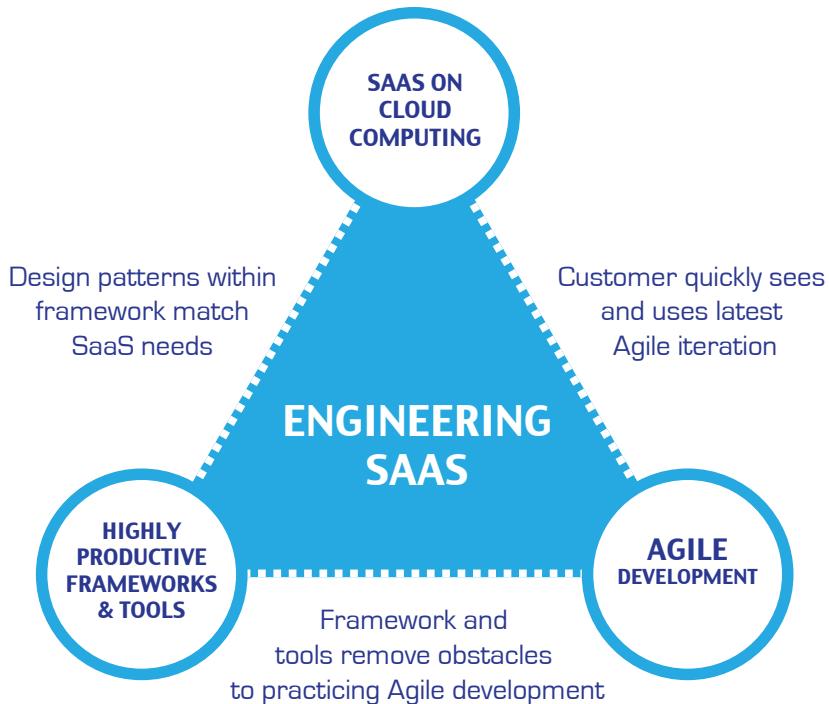


Figure 1.10: The Virtuous Triangle of Engineering SaaS is formed from the three software engineering crown jewels of (1) SaaS on Cloud Computing, (2) Agile Development, and (3) Highly Productive Framework and Tools.

—Michael Swaine, interview with Kent Beck, (Swaine 2001)

This single quote gives a good deal of the rationale behind the extreme programming (XP) version of Agile that we cover in this book. We keep iterations short, so that the customer sees the next version of the incomplete but working prototype every week or two. You write the tests *before* you write the code, and then you write the least amount of code it takes to make it pass the test. Pair programming means the code is under continuous review, rather than just on special occasions. Agile went from software methodology heresy to the dominant form of development in just a dozen years, and when combined with service oriented architecture, allows complex services to be built reliably.

While there is no inherent dependency among SaaS, Agile, and highly productive frameworks like Rails, Figure 1.10 suggests there is a synergistic relationship among them. Agile development means continuous progress while working closely with the customer, and SaaS on Cloud Computing enables the customer to use the latest version immediately, thereby closing the feedback loop (see Chapters 7 and Chapter 12). SaaS on Cloud Computing matches the Model–View–Controller design pattern (see Chapter 11), which Highly-Productive SaaS Frameworks expose (see Chapters 3, 4, and 5). Highly Productive Frameworks and Tools designed to support Agile development remove obstacles to practicing Agile (see Chapters 7, 8, and 10). We believe these three “crown jewels” form a “virtuous triangle” that leads to on-time and on-budget engineering of beautiful Software as a Service, and they form the foundation of this book.



This virtuous triangle also helps explain the innovative nature of the Rails community, where new important tools are frequently developed that further improve productivity, simply because it's so easy to do. We fully expect that future editions of this book will include tools not yet invented that are so helpful that we can't imagine how we got our work done without them!

As teachers, since many students find the Plan-and-Document methods tedious, we are pleased that the answers to the 10 questions in Figure 1.5 strongly recommend using Agile for student team projects. Nevertheless, we believe it is worthwhile for readers to be familiar with the Plan-and-Document methodology, as there are some tasks where it may be a better match, some customers require it, and it helps explain parts of the Agile methodology. Thus, we include sections near the end of all chapters in Part II that offer the Plan-and-Document perspective.

As researchers, we are convinced that software of the future will increasingly be built and rely on services in the Cloud, and thus Agile methodology will continue to increase in popularity in part given the strong synergy between them. Hence, we are at a happy point in technology where the future of software development is more fun both to learn and to teach. Highly productive frameworks like Rails let you understand this valuable technology by *doing* in a remarkably short time. The main reason we wrote this book is to help more people become aware of and take advantage of this extraordinary opportunity.

Cloud computing had existed for only a few years prior to the First Edition of this book, and has evolved spectacularly since then. Clusters of commodity computers had long been the basis of SaaS, but cloud computing changed how those clusters are used. Until the late 1990s, it was common for a particular computer to be dedicated to a particular SaaS app and have preinstalled all of the software components needed to run it. In contrast, starting in the mid 2000s, **virtual machine** technology made it possible for a single physical computer to emulate many computers, such that the software running in each virtual computer believed it was running on the real hardware. Like many other SaaS-relevant technologies, virtual machines had been around for decades—in this case, since at least the 1960s—but falling hardware costs and the dominance of the Intel architecture in server computers made high-performance virtual machines a practical tool for hosting many different SaaS apps on a single computer, even those requiring different operating systems and software packages. A typical SaaS app only cares about the type of virtual machine it's running in, and can remain largely ignorant of the details of the hardware and OS on which that virtual machine is hosted. Since the mid 2000s, further evolution of virtual machine technology led to lightweight **container** frameworks such as Docker, which isolate software packages from each other while sharing a single operating system kernel image. The most recent phase of virtualization is Function as a Service (FaaS), since the developer now specifies only the code of one or more functions and pays per function invocation. An early example is Amazon Lambda⁹. Although FaaS is also referred to as “serverless computing”, it is of course not truly serverless, as the functions have to run somewhere. The key distinction from SaaS is that developers do not deal with a software stack consisting of an app server, HTTP server, and so on; they write only the functions. Serverless computing is still evolving and has both pros and cons depending on the type of app to be deployed (Castro et al. 2019).

In the venerable LISP language, functions were called lambda-expressions, since the language was heavily inspired by the **lambda calculus** formalism.

We believe if you learn the contents of this book in conjunction with doing the suggested assignments and activities (CHIPS), you can build your own (simplified) version of a popular software service like FarmVille or Twitter while learning and following sound software engineering practices. While being able to imitate currently successful services and deploy them

in the cloud in a few months is impressive, we are even more excited to see what *you* will invent given this new skill set. We look forward to your beautiful code becoming long-lasting, and to becoming some of its passionate fans!

- C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction (Cess Center for Environmental)*. Oxford University Press, 1977. ISBN 0195019199.
- M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM (CACM)*, 53(4):50–58, Apr. 2010.
- L. A. Barroso and U. Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines (Synthesis Lectures on Computer Architecture)*. Morgan and Claypool Publishers, 2009. ISBN 159829556X. URL <http://www.morganclaypool.com/doi/10.2200/S00193ED1V01Y200905CAC006>.
- S. Begley. As Obamacare tech woes mounted, contractor payments soared. *Reuters*, October 17, 2013. URL <http://www.nbcnews.com/politics/politics-news/stress-tests-show-healthcare-gov-was-overloaded-v21337298>.
- J. Bidgood. Massachusetts appoints official and hires firm to fix exchange problems. *New York Times*, February 7, 2014. URL <http://www.nytimes.com/news/affordable-care-act/>.
- B. W. Boehm. Software engineering: R & D trends and defense needs. In P. Wegner, editor, *Research Directions in Software Technology*, Cambridge, MA, 1979. MIT Press.
- B. W. Boehm. A spiral model of software development and enhancement. In *ACM SIGSOFT Software Engineering Notes*, 1986.
- E. Braude. *Software Engineering: An Object-Oriented Perspective*. John Wiley and Sons, 2001. ISBN 0471692085.
- P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. The rise of serverless computing. *Communications of the ACM (CACM)*, 62(12), Dec 2019.
- R. Charette. Why software fails. *IEEE Spectrum*, 42(9):42–49, September 2005.
- L. Chung. Too big to fire: How government contractors on HealthCare.gov maximize profits. *FMS Software Development Team Blog*, December 7, 2013. URL <http://blog.fmsinc.com/too-big-to-fire-healthcare-gov-government-contractors>.
- M. Cormick. Programming extremism. *Communications of the ACM*, 44(6):109–110, June 2001.
- E. Enge. Mobile vs desktop usage in 2018: Mobile takes the lead. Stone Temple Consulting, Apr 2018. URL <https://www.stonetemple.com/mobile-vs-desktop-usage-study>.
- H.-C. Estler, M. Nordio, C. A. Furia, B. Meyer, and J. Schneider. Agile vs. structured distributed software development: A case study. In *Proceedings of the 7th International Conference on Global Software Engineering (ICGSE'12)*, pages 11–20, 2012.

- ET Bureau. Need for speed: More it companies switch to agile code development. *The Economic Times*, August 6, 2012. URL http://articles.economictimes.indiatimes.com/2012-08-06/news/33065621_1_thoughtworks-software-development-iterative.
- M. Fowler. The New Methodology. *martinfowler.com*, 2005. URL <http://www.martinfowler.com/articles/newMethodology.html>.
- E. Harrington. Hearing: Security flaws in Obamacare website endanger AmericansHealthCare.gov. *Washington Free Beacon*, 2013. URL <http://freebeacon.com/hearing-security-flaws-in-obamacare-website-endanger-americans/>.
- S. Horsley. Enrollment jumps at HealthCare.gov, though totals still lag. *NPR.org*, December 12, 2013. URL <http://www.npr.org/blogs/health/2013/12/11/250023704/enrollment-jumps-at-healthcare-gov-though-totals-still-lag>.
- A. Howard. Why Obama's HealthCare.gov launch was doomed to fail. *The Verge*, October 8, 2013. URL <http://www.theverge.com/2013/10/8/4814098/why-did-the-tech-savvy-obama-administration-launch-a-busted-healthcare-website>.
- C. Johnson and H. Reed. Why the government never gets tech right. *New York Times*, October 24, 2013. URL http://www.pmi.org/en/Professional-Development/Career-Central/Must_Have_Skill_Agile.aspx.
- J. Johnson. The CHAOS report. Technical report, The Standish Group, Boston, Massachusetts, 1995. URL <http://blog.standishgroup.com/>.
- J. Johnson. HealthCare.gov chaos. Technical report, The Standish Group, Boston, Massachusetts, October 22, 2013a. URL http://blog.standishgroup.com/images/audio/HealthcareGov_Chaos_Tuesday.mp3.
- J. Johnson. The CHAOS manifesto 2013: Think big, act small. Technical report, The Standish Group, Boston, Massachusetts, 2013b. URL <http://www.standishgroup.com>.
- C. Jones. Software project management practices: Failure versus success. *CrossTalk: The Journal of Defense Software Engineering*, pages 5–9, Oct. 2004. URL <http://cross5talk2.squarespace.com/storage/issue-archives/2004/200410/200410-Jones.pdf>.
- J. M. Juran and F. M. Gryna. *Juran's quality control handbook*. New York: McGraw-Hill, 1998.
- P. Kruchten. *The Rational Unified Process: An Introduction, Third Edition*. Addison-Wesley Professional, 2003. ISBN 0321197704.
- T. Lethbridge and R. Laganiere. *Object-Oriented Software Engineering: Practical Software Development using UML and Java*. McGraw-Hill, 2002. ISBN 0072834951.
- National Research Council. *Achieving Effective Acquisition of Information Technology in the Department of Defense*. The National Academies Press, 2010. ISBN 9780309148283. URL http://www.nap.edu/openbook.php?record_id=12823.

- P. Naur and B. Randell. *Software engineering*. Scientific Affairs Div., NATO, 1969.
- J. R. Nawrocki, B. Walter, and A. Wojciechowski. Comparison of CMM level 2 and extreme programming. In *7th European Conference on Software Quality*, Helsinki, Finland, 2002.
- M. Paulk, C. Weber, B. Curtis, and M. B. Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, 1995. ISBN 0201546647.
- G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.
- Project Management Institute. Must-have skill: Agile. *Professional Development*, February 28, 2012. URL http://www.pmi.org/en/Professional-Development/Career-Central/Must_Have_Skill_Agile.aspx.
- W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of WESCON*, pages 1–9, Los Angeles, California, August 1970.
- I. Sommerville. *Software Engineering, Ninth Edition*. Addison-Wesley, 2010. ISBN 0137035152.
- M. Stephens and D. Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Apress, 2003.
- M. Swaine. Back to the future: Was Bill Gates a good programmer? What does Prolog have to do with the semantic web? And what did Kent Beck have for lunch? *Dr. Dobb's The World of Software Development*, 2001. URL <http://www.drdobbs.com/back-to-the-future/184404733>.
- A. Taylor. IT projects sink or swim. *BCS Review*, Jan. 2000. URL <http://archive.bcs.org/bulletin/jan00/article1.htm>.
- F. Thorp. ‘Stress tests’ show HealthCare.gov was overloaded. *NBC News*, November 18, 2013. URL <http://www.nbcnews.com/politics/politics-news/stress-tests-show-healthcare-gov-was-overloaded-v21337298>.
- J. Zients. HealthCare.gov progress and performance report. Technical report, Health and Human Services, December 1, 2013. URL <http://www.hhs.gov/digitalstrategy/sites/digitalstrategy/files/pdf/healthcare.gov-progress-report.pdf>.

Notes

¹https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML#Guides

²<http://www.youtube.com/watch?v=DeBi2ZxUZiM>

³<https://www.bbc.com/future/article/20150505-the-numbers-that-lead-to-disaster>

⁴<https://youtu.be/AP7lVJphbSk>

⁵<https://getbootstrap.com/docs/4.0/examples/navbars/>

⁶<https://sensortower.com/blog/25-top-ios-apps-and-their-version-update-frequencies>

⁷<https://www.w3.org/TR/2011/WD-html5-20110525/offline.html>

⁸<http://developers.slashdot.org/story/08/05/11/1759213/>

⁹<https://aws.amazon.com/lambda>

Part I

Software as a Service: Frameworks and Languages

2

How to Learn a New Language

Barbara Liskov (1939–)
was one of the first women in the USA to receive a Ph.D. in computer science (in 1968) and received the 2008 Turing Award for foundational innovations in programming language design. Her inventions include abstract data types and iterators, both of which are central to Ruby.



*You never need optimal performance, you need good-enough performance
...Programmers are far too hung up with performance.*

—Barbara Liskov, 2011

2.1	Prelude: Learning to Learn Languages and Frameworks	46
2.2	Pair Programming	48
2.3	Introducing Ruby, an Object-Oriented Language	50
2.4	Ruby Idioms: Poetry Mode, Blocks, Duck Typing	59
2.5	CHIPS: Ruby Intro	64
2.6	Gems and Bundler: Library Management in Ruby	64
2.7	Fallacies and Pitfalls	67
2.8	Concluding Remarks: How (Not) To Learn a Language By Googling	69

Prerequisites and Concepts

Software engineers who are unable to quickly learn and use new languages and frameworks risk finding themselves obsolete or out of a job every few years. This chapter focuses on building those skills, starting with a developer's-eye inhalation of the Ruby language. In Chapter 6 we will repeat the process for JavaScript.

Prerequisites:

- You should be comfortable programming in some modern object-oriented (OO) language, such as Java or Python, including OO concepts such as class vs. instance variables and methods, public vs. private methods, and so on.
- Helpful, but not strictly required, is familiarity with basic operations on collections such as those seen in *functional programming* languages and borrowed by the Python language. For example, `map` takes a function (or *lambda expression*) and a collection, and returns a new collection resulting from the application of the function to each element of the original collection. `filter` takes a Boolean-valued function and a collection, and returns a new collection consisting of those elements from the original collection for which the function returns true.
- *Regular expressions*, sometimes abbreviated *regexp*s or *regexes*, are sequences of characters that define a search pattern. All modern programming languages use them. Rubular¹ is a web app that lets you practice regexes in Ruby.

Concepts:

- Learning a language and learning a framework often go together: you learn Ruby so that you can use Rails. This means you must understand three things: the new language, the structure or architectural model the framework prescribes for applications (how the “moving parts” of an application work together), and how the language’s features are used to expose that structure.
- Learning a new language requires understanding its basic object-orientation and encapsulation mechanisms (classes, inheritance, composition), its basic imperative mechanics (variables, naming conventions, control flow), and how it manages complexity (namespacing, libraries, library and package management).
- At the core of Ruby is the idea that *everything is an object*, and even “basic” operations like addition are defined in terms of sending a message to an object asking the object to do something.
- Learning the idioms that make a language unique is a key aspect of mastering the language. Idioms pervasive in Ruby but less common in other modern languages include mix-ins (duck typing) and blocks (anonymous lambdas).
- Learning a language is largely about learning its libraries and how they are managed. Ruby libraries (“gems”) and the Bundler tool allow an app to specify fine-grained dependencies on many interrelated libraries.



2.1 Prelude: Learning to Learn Languages and Frameworks

We will use the term *stack* to refer loosely to any set of technologies—typically languages, frameworks, and subsystems—used in the development of a particular type of application. A major part of most stacks is a *framework* for building a particular type of app using a particular language and relying on other elements in the stack. Today, most developers learn new programming languages not to use them standalone, but because they want to use a particular framework or stack: Native mobile apps for iOS are written in Objective C; React.js apps are written in JavaScript; and the Ruby language had existed for 10 years before the highly productive Rails framework made it popular.

As Figure 1.6 showed, though, stacks and frameworks come and go. Thus, our approach is inspired by a Chinese proverb:

Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime.

—Chinese proverb

Following this advice, our goal is not so much to give you a fish (introduce you as quickly as possible to a particular framework or stack) but rather to help you learn to fish—by giving you guidance on how to develop the conceptual vocabulary to rapidly learn new ones. In this chapter and the next, we will also give you a starter fish, in the form of the Ruby language and Rails framework. In Chapter 6, we will do the same for JavaScript and jQuery.

For concreteness, we will limit our discussion of learning a new language to imperative, object-oriented (OO) languages. Besides Ruby and JavaScript, which we introduce in this book, other languages in this family include Java, Python, C++, C#, Scala, Perl, PHP, Lua, Tcl, and dozens more. As you will see, from the point of view of learning new languages, all of these languages are far more alike than they are different, because they all foster an imperative (linear, step-by-step) approach to solving programming problems and they all provide comparable facilities for managing complexity by encapsulating data along with the operations on that data.

Proficient software engineers can rapidly *learn* new languages, and the stacks or frameworks that use them, by mastering a technical vocabulary consisting of three main components:

1. Learn what's different about the language: Most imperative OO languages have straightforward machinery for primitives (variables, types, control flow), reuse (inheritance, interfaces), complexity management (composition, inheritance, data hiding), debugging, and library usage (importing, package/dependency management). What idioms or facilities are *different* from other recently-popular languages? How does the language manage libraries, the learning of which typically consumes far more time than learning the language itself?
2. Understand the app architecture implied by the framework: What is the application structure or set of patterns **reified** by the framework? What are the major “moving parts” of an application built using that framework, and how does their arrangement influence the way we formulate an application’s functionality in terms of that framework?

While many of these include elements of other language families such as functional languages, the primary way they are used is imperatively.

3. Associate the language's features with the frameworks' structure. How does the language help application writers by using language mechanisms to expose the framework's architecture and patterns? One example, as we will see, is that the Rails framework relies heavily on *convention over configuration*: if you follow certain naming rules, you need not provide configuration files explaining (for example) which class in your app mediates access to which database table. Ruby supports convention over configuration through its use of reflection and metaprogramming.

Here, then, is our 8-point plan for learning a new imperative object-oriented language:

1. **Types and typing.** Is the language strongly or weakly typed, and is typing static or (as in Ruby and JavaScript) **dynamic**? In C++ and Java, a variable's type must be declared at compile time and cannot change during program execution (static typing), and only objects of that type or a compatible subtype may ever be assigned to the variable (strong typing). In Java, for example, within the same scope, the same variable cannot be assigned first to an integer and later to a string. In Ruby, as we will see, a variable does not have a type declared at compile time (dynamic typing) and can be assigned to an object of any type (weak typing). This policy makes certain kinds of type errors easier to commit, but also enables an extremely powerful kind of code reuse called *mix-ins*, which are analogous to interfaces in Java but far more flexible.
2. **Primitives.** What are primitive types (numbers, strings, collections, and so on)? What are the rules or conventions for naming things (variables, functions, classes, namespaces, and so on)? What are the basic mechanisms for variable assignment, variable scope, and control flow? What are the basic ways in which strings are manipulated, including the use of **regular expressions** and **string interpolation**?
3. **Methods.** How are methods (functions, procedures) defined and called? How are they named? How are class (static) methods differentiated from instance methods?
4. **Abstraction and Encapsulation.** How are classes defined, subclassed, and composed? What are the mechanics of specifying instance methods and variables, class (static) methods and variables, interfaces, and so on?
5. **Idioms.** What idioms differentiate the language from others you probably know, and how are they used? Prominent examples in Ruby include symbols (akin to immutable strings), blocks (also known as anonymous lambdas or closures, and heavily used in Ruby to implement **iterators**), and functional-programming idioms for operating on collections.
6. **Libraries.** What facilities does the language have for managing libraries? How are libraries and the functions available in them named, imported, and used? What tools for **package management** (also called *dependency management*) are available to ensure that an application can be reproducibly deployed by other developers or on production systems with the correct versions of all the libraries on which it depends? We return to this topic in Chapter 12.
7. **Debugging.** What debugging tools are available? Can you drop into an interactive debugger from a running program? Can you set **breakpoints** or watchpoints (data-value-based breakpoints) to stop a running program and inspect or modify its state?



Scala is an interesting hybrid: it is dynamically but strongly typed, with **type inference** applied at runtime to determine whether a particular expression is legal.

Is there easy access to an interactive console, ***read-eval-print loop*** (REPL), or other mechanism to try out short bits of code interactively?

8. **Testing.** How do you create and run automated tests? We will introduce the topic here but we will have much more to say about it in Chapter 8.

Heeding Confucius’ “I do and I understand,” we also ask: what tools are available to allow a new developer to quickly start experimenting with the language features and come up to speed, perhaps without requiring an elaborate installation procedure on their own computer?



We will use the above steps to learn just enough Ruby to allow us to dive into the popular server-side SaaS framework called Rails. Beware, though: while experience in other languages can help you more quickly learn a new language, avoid the pitfall of trying to map everything directly over. It’s common for two languages to have some features in common or at least some features that are analogous, but if there were *no* conceptual differences between two languages, one of the two would be redundant. Therefore, resist the temptation to ask “Is feature *x* in Ruby the same as feature *y* in Python or Java?” Look for analogies and similarities, but don’t expect complete isomorphism.

Summary of how to learn a new language effectively:

- Most imperative object-oriented languages are philosophically more alike than they are different. The basic elements of a new language—types and typing, primitives, method definition, control flow, abstraction and encapsulation—are therefore usually easy to pick up.
- That said, a developer learns a new language in order to use a particular framework. Therefore, you should expect that the language may have specific features that make it a particularly good fit for that framework. Those features, or idioms, are likely to be heavily used by an app framework that uses the language effectively. They are more likely to be unfamiliar to you, but more critical to master in order to wield the language effectively.
- Finally, using a language effectively also requires learning how to use its debugging facilities and its libraries, especially how to manage dependencies among multiple interdependent libraries.

2.2 Pair Programming

Interviewer: At Google, you share an office, and you even code together.

Sanjay: We usually sit, and one of us is typing and the other is looking on, and we’re chatting all the time about ideas, going back and forth.

—Interview with Jeff Dean and Sanjay Ghemawat, creators of MapReduce (Hoffmann 2013)

The name Extreme Programming (XP), which is the variant of the Agile lifecycle we follow in this book, suggests a break from the way software was developed in the past. One



Figure 2.1: Sarah Mei and JR Boyens at Pivotal Labs engaged in pair programming. Sarah is driving and JR is observing.

Although two keyboards are visible, only Sarah is coding; the computer in front of JR is for documentation and other relevant information such as Pivotal Tracker, as you can see in the right-hand photo. All the pairing stations (visible in the background) are identical and don't have email or other software installed; other computers away from the pairing stations are provided for checking email. Photo by Tonia Fox, courtesy of Pivotal Labs.

new option in this brave new software world is ***pair programming***. The goal is improved software quality by having more than one person developing the same code. But while pair programming emerged as a practice used by Agile teams and developers, your authors believe it's also a way to accelerate the learning of a new language and framework. That is why we introduce it in this chapter, whose theme is *learning how to learn* new languages and frameworks.

Although pair programming is properly considered a software engineering process rather than being associated with a particular language, we introduce it here to encourage its use early, and especially while learning a new language. As the name suggests, in pair programming two developers share one computer. Each takes on a different role:

- The *driver* enters the code and thinks tactically about how to complete the current task, explaining his or her thoughts out loud as appropriate while typing.
- The *observer* or *navigator*—following the automobile analogy more closely—reviews each line of code as it is typed in, and acts as a safety net for the driver. The observer is also thinking strategically about future problems that will need to be addressed, and makes suggestions to the driver.

Normally a pair will take alternate driving and observing as they perform tasks. Figure 2.1, shows engineers at Pivotal Labs—makers of Pivotal Tracker—who spend most of the day doing pair programming.(Moore 2011)

Pair programming is cooperative, and should involve a lot of talking. It focuses effort on the task at hand, and two people working together increases the likelihood of following good development practices. But it is effective only if both collaborators share a common focus throughout the process (Rodríguez et al. 2017); if one partner is silent or checking email, then it's not pair programming, just two people sitting near each other. In fact, it is normal for the navigator to talk more than the driver, giving constant feedback. If one collaborator is uncertain about what's going on, both collaborators should recognize that fact and pause for dialogue to get back in sync. If a large proportion of the pair's dialogue focuses on uncertainty, the pair may benefit from outside help, such as consulting another team member or experienced developer.

Dilbert on Pair Programming The comic strip Dilbert comments humorously on pair programming in these two² strips³.

Pair programming has the side effect of transferring knowledge between the pair, including programming idioms, tool tricks, company processes, customer desires, and so on. Thus, to widen the knowledge base, some teams purposely swap partners per task so that eventually everyone is paired together. For example, *promiscuous pairing* of a team of four leads to six different pairings.

The studies of pair programming versus solo programming support the claim of reduced development time and improvement in software quality. For example, Cockburn and Williams 2001 found a 20% to 40% decrease in time and that the initial code failed to 15% of the tests instead of 30% by solo programmers. However, it took about 15% more hours collectively for the pair of programmers to complete the tasks versus the solo programmers. The majority of professional programmers, testers, and managers with 10 years of experience at Microsoft reported that pair programming worked well for them and produced higher-quality code (Begel and Nagappan 2008). A study of pair programming studies concludes that pair programming is quicker when programming task complexity is low—perhaps one point tasks on the Tracker scale—and yields code solutions of higher quality when task complexity is high, or three points on our Tracker scale. In both cases, it took more total effort than solo programming (Hannay et al. 2009).

The experience at Pivotal Labs suggests that these studies may not factor in the negative impact on productivity of the distractions of our increasingly interconnected modern world: email, Twitter, Facebook, and so on. Pair programming forces both programmers to pay attention to the task at hand for hours at a time. Indeed, new employees at Pivotal Labs go home exhausted since they were not used to concentrating for such long stretches.

Even if pair programming takes more effort, one way to leverage the productivity gains from Agile and Rails is to “spend” it on pair programming. Having two heads develop the code can reduce the time to market for new software or improve quality of end product. We recommend you try pair programming to see if you like it, which some developers love.

Summary: When it’s time to start coding, one approach is pair programming, which promises higher quality and shorter development time but perhaps higher programming costs due to two people doing the work. The pair splits into a driver and an observer, with the former working tactically to complete the task at hand and the latter thinking strategically about future challenges and making suggestions to the driver.

Self-Check 2.2.1. *True or False: Research suggests that pair programming is quicker and less expensive than solo programming.*

◊ False. While there have not been careful experiments that would satisfy human subject experts, and it is not clear whether they account for the lack of distractions when pair programming, the current consensus of researchers is that pair programming is more expensive—more programmer hours per task—than solo programming. ■

Self-Check 2.2.2. *True or False: A pair will eventually figure out who is the best driver and who is the best observer, and then stick primarily to those roles.*

◊ False: An effective pair will alternate between the two roles, as it’s more beneficial (and more fun) for the individuals to both drive and observe. ■

2.3 Introducing Ruby, an Object-Oriented Language

Ruby is a minimalist language: while its libraries are rich, there are relatively few mechanisms *in the language itself*. Its world view might be described as “extreme object orientation.” Two principles will help you quickly learn to read and write Ruby:

1. Everything is an object—even an integer—and it is literally the case that every operation is a method call on some object and every method call returns a value.
2. Like Java and Python, Ruby has conventional classes; but unlike Java public attributes or Python instance variables, only a class’s instance methods—not its instance variables—are visible outside the class. In other words, *all* access to instance variables from outside the class must take place via public **accessor methods**; instance variables lacking public accessor methods are effectively private. (Python supports a similar approach but doesn’t make it mandatory.)

Let’s break down our investigation of Ruby according to the elements proposed in the previous section and in light of the above principles.

Types, typing, and names. Ruby is dynamically typed: variables don’t have types, though the objects they refer to do. Hence `x='foo'` ; `x=3` is legal. As row 1 of Figure 2.2 shows, a single or double @-sign precedes names of instance or class (static) variables, while local variables are “barewords”; all must begin with lowercase letters, and `snake_case` is strongly preferred over `camelCase`. Row 2 shows the syntax for other named entities such as classes and constants; all except globals (which you should never use anyway) *must* begin with a capital letter, with `UpperCamelCase` used for class names. (So even though strictly speaking `lowerCamelCase` is legal for local and instance variables, it’s *highly* discouraged because it is visually difficult to distinguish from `UpperCamelCase` and because of the ease with which a typo can change the former into the latter and cause errors.) The namespaces for each kind of named entity are separate, so that `foo`, `@foo`, `@@foo`, `F00`, `Foo`, `$F00` are all distinct.

In learning any new language, an annoying type-related eye-poke is having to memorize how the language handles Boolean evaluation of non-Boolean expressions. Some languages have special Boolean types and values, such as Python’s `True` and `False` (which have special type `Bool`), JavaScript `true` and `false` (type `boolean`), and Ruby’s `true` and `false` (`TrueClass` and `FalseClass` respectively). To avoid confusion with such actual Boolean literals, developers often say *truthy* or *falsy* to describe the value of a non-Boolean expression `e` when used in a conditional of the form `if (e)...`. Unfortunately, the rules for truthiness are different and largely arbitrary in each language. In Ruby, the literals `false` and `nil` are falsy, but *all other values*, including the number zero, the empty string, the empty array, and so forth, are truthy. In contrast, in Python, zero is falsy, but the empty string is truthy; in JavaScript, zero and the empty string are both falsy, as are the special values `undefined` and `null`, but the empty array is truthy; and so on. In languages that include both a true Boolean type and unary logical negation (usually `!`), writing as `!!x` forces the expression to have a Boolean-valued result (for example, if `x` is falsy, then `!!x` is the actual Boolean value for false).

Primitives. Figure 2.2 shows the mostly unsurprising syntax of basic Ruby elements. Ruby has special Boolean values (row 3) including the special value `nil`, which is the usual result of an operation that otherwise would yield no meaningful return value, such as looking up a nonexistent key in a hash or a nonexistent value in an array.

An editor with language-specific highlighting and indentation is an essential tool for software writers. Popular choices today include the open-source Ace⁴ and the commercial product Sublime Text⁵. Some old-timers, including one of your authors, will use **Emacs** until it is pried from his cold, dead hands.

Ruby has no separate “empty result” value such as Python `None` or JavaScript `null`. That is to say: a JavaScript variable whose value is `null` means that the variable references nothing in particular, rather than signifying “falseness” in a Boolean sense, whereas Ruby `nil` may signal either Boolean falseness or a variable that refers to nothing.

In addition to strings (row 4), Ruby also includes a type called ***symbol*** (row 4), such as `:octocat`, essentially an immutable “token” whose value is itself. It is typically used for enumerations, like an `enum` type in C or Java, though it has other purposes as well. A symbol is not the same as a string, but as the figure shows, strings and symbols can be easily converted to each other.

Row 6 and Figure 2.3 summarize Ruby’s straightforward support for manipulating regular expressions and capturing the results of regex matches. Given the amount of text handling done by modern SaaS apps, mastering regexes and understanding how a new language provides access to a regex engine is *de rigueur* for programmers.

Collections (rows 7–9: arrays and hashes) can combine keys and values of different types. Hashes in particular, also called associative arrays or hashmaps in other languages, are ubiquitous in Ruby.

Every Ruby statement is an expression that returns a value; assignments return the value of their left-hand side, that is, the value of the variable or other ***L-value*** that was just assigned to.

Methods. A method is defined with `def method_name(arg1, ..., argN)` and ends with `end`. All statements in between are the method definition. All methods return a value; if a method doesn’t have an explicit `return` statement, the value of the last expression evaluated in the method is its return value, which is always well-defined since every Ruby statement results in a value.

Everything in Ruby, even a lowly integer, is a full-fledged object that is an instance of some class. Every operation, without exception, is performed by calling a method on an object. The notation `obj.meth()` calls method `meth` on the object `obj`, which is said to be the *receiver* and is expected to be able to *respond to* `meth`. For example, the expression `5.class()` sends the method call `class` with no arguments to the object `5`. The `class` method happens to return the class that an object belongs to, in this case `Fixnum`.

As we’ll see in more detail in the next section, Ruby allows omitting parentheses around argument lists when doing so does not result in ambiguous parsing. Hence `5.class` is equivalent to `5.class()`.

Furthermore, since everything is an object, the result of every expression is, by definition, something on which you can call other methods. Hence `(5.class).superclass` tells you what `Fixnum`’s superclass is, by sending the `superclass` method call with no arguments to `Fixnum`, an object representing the class to which `5` belongs. Method calls associate to the left, so this example could be written `5.class.superclass`. Such *method chaining* is extremely idiomatic in Ruby.

■ Elaboration: Reflection

This example gives a glimpse of Ruby’s comprehensive ***reflection***—the ability to ask objects about themselves. `5.respond_to?('class')` tells you that the object `5` would be able to respond to the method `class` if you asked it to. `5.methods` lists all methods to which the object `5` responds, including those defined in its ancestor classes. `5.method('+')` reveals that the `+` method is defined in class `Fixnum`, whereas `5.method('ceil')` reveals that the `ceil` method is defined in `Integer`, an ancestor class of `Fixnum`.

Smalltalk, which inspires Ruby’s object model, was itself inspired by ideas in **Simula**, the first object-oriented programming language, whose inventors won the Turing Award for their contribution.

A Ruby class such as `Fixnum` is itself an instance of `Class`, which is a class whose instances are also classes (we say that `Class` is a ***metaclass***). Unless you’re a languages geek, don’t think too hard about this right away or it will make your head hurt.

1. Variables	<code>local_variable, @instance_variable</code>	<code>@@class_variable,</code>
2. Constants	<code>ClassName, CONSTANT, \$GLOBAL, \$global</code>	
3. Booleans	<code>false, nil</code> are false; <code>true</code> and <i>everything else</i> (zero, empty string, etc.) is true.	
4. Strings and Symbols	<code>"string", 'also a string', %q{like single quotes}, %Q{like double quotes}, :symbol</code> special characters (<code>\n</code>) expanded in double-quoted but not single-quoted strings	
5. Expressions in <i>double-quoted</i> strings	<code>@foo = 3 ; "Answer is #{@foo}" ; %Q{Answer is #{@foo+1}}</code>	
6. Regular expression matching (Fig. 2.3)	<code>"hello" =~ /lo/</code> or <code>"hello".match(Regexp.new 'lo')</code>	
7. Arrays	<code>a = [1, :two, 'three'] ; a[1] == :two</code>	
8. Hashes	<code>h = { :a => 1, 'b' => "two" } ; h['b'] == "two" ; h.has_key?(:a) == true</code>	
9. Hashes (alternate notation, Ruby 1.9+)	<code>h = { a: 1, 'b': "two" }</code>	
10. Instance method	<code>def method(arg, arg)...end</code> (use *args for variable number of arguments)	
11. Class (static) method	<code>def ClassName.method(arg, arg)...end,</code> <code>def self.method(arg, arg)...end</code>	
12. Special method names <i>Ending these methods' names in ? and ! is optional but idiomatic</i>	<code>def setter=(arg, arg)...end</code> <code>def boolean_method?(arg, arg)...end</code> <code>def dangerous_method!(arg, arg)...end</code>	
Conditionals	Iteration (see Section 2.4)	Exceptions
<code>if cond (or unless cond) statements [elsif cond statements] [else statements] end</code>	<code>while cond (or until cond) statements end</code> <code>1.upto(10) do i ...end</code> <code>10.times do...end</code> <code>collection.each do elt ...end</code>	<code>begin</code> <code>statements</code> <code>rescue AnError => e</code> <i>e is an exception of class AnError;</i> <code>multiple rescue clauses OK</code> <code>[ensure</code> <i>this code is always executed]</i> <code>end</code>

Figure 2.2: Basic Ruby elements and control structures, with optional items in [square brackets]. Statements are separated by newlines (most commonly) or semicolons (rarely). Indentation is insignificant. Besides the usual basic primitive types and collection types, hashes (also known as associative arrays or hashmap; Ruby class `Hash`) are ubiquitous.

	<i>Symbol</i>	<i>Meaning</i>	<i>Example</i>	<i>Matches</i>	<i>Example</i>	<i>Mismatch</i>
<i>Count</i>	*	0 or more	a*		aaaa	b
	+	1 or more	a+	a	aaaa	
	?	0 or 1	a?		a	aaaa
<i>Anchors, Sets, Range, Append</i>	^	start of line, also NOT in set	^a	a	ab	ba
	\$	end of line	a\$	a	dcba	ab
	()	group, also captures that group in Ruby	(ab)+	ababab	ab	b
	[]	set	[ab]	a	b	ab
	[x-y]	character range	[0-9]	3	9	a
		OR	(It's It is)	It's	It is	Its
	[^]	NOT (opposite) in set	[^"]	b	9	"
	.	any character (except newline)	.{3}	abc	1+2	aa
	\	used to match meta-characters, also for classes	\.\$	The End.	.	a
	i	append to pattern to specify case insensitive match	\ab\i	Ab	ab	a
<i>Classes</i>	\d	decimal digit ([0-9])	\d	3	9	a
	\D	not decimal digit ([^0-9])	\D	a	=	3
	\s	whitespace character	\s			a
	\S	not whitespace character	\S	a	=	
	\w	"word" character ([a-zA-Z0-9_])	\w	a	9	=
	\W	"nonword" character ([^\w])	\W	=	\$	a
	\n	newline	\n	--	--	a

Figure 2.3: Like most modern languages, Ruby supports *Perl compatible regular expressions* (PCRE), often shortened to *regex(es)* or *regexp(s)*. The name PCRE is inspired by the vastly expanded regular-expression capabilities that first appeared in the *Perl* scripting language.

Sugared	De-sugared	Explicit send
<code>10 % 3</code>	<code>10.modulo(3)</code>	<code>10.send(:modulo, 3)</code>
<code>5+3</code>	<code>5.+ (3)</code>	<code>5.send(:+, 3)</code>
<code>x == y</code>	<code>x.==(y)</code>	<code>x.send(:==, y)</code>
<code>a * x + y</code>	<code>a.*(x).+(y)</code>	<code>a.send(:*, x).send(:+, y)</code>
<code>a + x * y</code>	<code>a.+(x.*(y))</code>	<code>a.send(:+, x.send(:*, y))</code> <i>(operator precedence preserved)</i>
<code>x[3]</code>	<code>x.[](3)</code>	<code>x.send(:[], 3)</code>
<code>x[3] = 'a'</code>	<code>x.[]=(3, 'a')</code>	<code>x.send(:[]=, 3, 'a')</code>
<code>/abc/,%r{abc}</code>	<code>Regexp.new("abc")</code>	<code>Regexp.send(:new, 'abc')</code>
<code>str =~ regex</code>	<code>str.match(regex)</code>	<code>str.send(:match, regex)</code>
<code>regex =~ str</code>	<code>regex.match(str)</code>	<code>regex.send(:match, str)</code>
<code>\$1...\$n</code> (regex capture)	<code>Regexp.last_match(n)</code>	<code>Regexp.send(:last_match, n)</code>

Figure 2.4: The first column is Ruby’s syntactic sugar for common operations, the second column shows the explicit method call, and the third column shows how to perform the same method call using Ruby’s `send`, which accepts either a string or (more idiomatically) a symbol for the method name.

As Figure 2.4 shows, even basic math operations and array references are actually method calls on their receivers. Hence, concepts such as **type casting** rarely apply in Ruby: while you can certainly call `5.to_s` or `"5".to_i` to convert between strings and integers, for example, writing `a+b` means calling method `+` on receiver `a`, so the behavior depends entirely on how `a`’s class (or one of its ancestors or mix-ins) implements the instance method `+`. Hence, both `3+2` and `"foo"+"bar"` are legal Ruby expressions, but the first one calls `+` as defined in `Numeric` (the ancestor class of `Fixnum`) whereas the second calls `+` as defined in `String`. Rubyists write `ClassName#method` to indicate the instance method `method` in `ClassName` and `ClassName.method` to indicate the class (static) method `method` in `ClassName`. We can therefore say that the expression `3+2` results in calling `Fixnum#+` on the receiver `3`.

Abstraction and encapsulation. Ruby supports traditional inheritance, using the notation `class SubFoo<Foo` to indicate that `SubFoo` is a subclass of `Foo`. A class can inherit from at most one superclass (Ruby lacks multiple inheritance), and all classes ultimately inherit from `BasicObject`, sometimes called the *root class*, which has no superclass. As with most languages that support inheritance, if an object receives a call for a method not defined in its class, the call will be passed up to the superclass, and so on until the root class is reached or an *undefined method* exception is raised. The default constructor for a class must be a method named `initialize`, but it is always called as `Foo.new`—that is an idiosyncrasy of the language. Classes can have both class (static) methods and instance methods, and both class (static) variables and instance variables. Class variable names begin with `@@` and instance variable names begin with `@`. Class and instance method names look the same.

Probably the biggest surprise to newcomers learning about Ruby’s class machinery is that there is *no direct access to class or instance variables at all* from outside the class. In other languages, certain instance variables of a class can be declared public, such as attributes in Java. In Ruby, access to class or instance state must be through **getter and setter methods**, also collectively called *accessor methods*. Figure 2.5 shows examples of getters (lines 10–12, 16), setters (lines 13–15: note that setter methods conventionally have names ending in `=`, allowing syntax such as line 33 shows), and a simple instance method that accesses other instance variables (line 18). From the caller’s point of view in lines 33–34, it is impossible to

You can verify this by evaluating
`"foobar".method(:+)`
and `5.method(:+)`.

Mix-ins The truth is a bit more subtle: mix-ins, which we’ll describe shortly, can handle an undefined method call *before* punting up to the superclass.

<https://gist.github.com/450367d90330578a8ee4d355979fc7d1>

```

1  class Movie
2    def initialize(title, year)
3      @title = title
4      @year = year
5    end
6    # class (static) methods - 'self' refers to the actual class
7    def self.find_in_tmdb(title_words)
8      # call TMDb to search for a movie...
9    end
10   def title
11     @title
12   end
13   def title=(new_title)
14     @title = new_title
15   end
16   def year ; @year ; end
17   # note: no way to modify value of @year after initialized
18   def full_title ; "#{@title} (#{@year})"; end
19 end
20
21 # A more concise and Rubyistic version of class definition:
22 class Movie
23   def self.find_in_tmdb(title_words)
24     # call TMDb to search for a movie...
25   end
26   attr_accessor :title # can read and write this attribute
27   attr_reader :year     # can only read this attribute
28   def full_title ; "#{@title} (#{@year})"; end
29 end
30
31 # Example use of the Movie class
32 beautiful = Movie.new('Life is Beautiful', '1997')
33 beautiful.title = 'La vita e bella'
34 beautiful.full_title # => "La vita e bella (1997)"
35 beautiful.year = 1998 # => ERROR: no method 'year='

```

Figure 2.5: A simple class definition in Ruby showing that explicit getter and setter methods are the only way to access instance variables from outside a class, and that Ruby provides shortcuts (lines 19–20) that avoid having to define every accessor method explicitly. Rather than distinguish “private” vs. “public” instance and class variables, one simply provides public accessor methods (read-only, write-only, or read/write) for state that should be publicly visible.

tell whether a given method simply “wraps” access to an instance variable (as `title` does) or produces its result by computing something (as `full_title` does). This design choice illustrates Ruby’s hard-line position on the ***Uniform Access Principle***, which concerns one aspect of ***encapsulation*** in object-oriented programming: It should be impossible to determine the implementation details of an object’s state or its operations from outside the object.

Beware! If you’re used to Java or Python, it’s very easy to think of the syntax in line 33 as *assignment to an attribute or instance variable*, but it is just a method call, and in fact could be written as `beautiful.send('title=', 'La vita e bella')`. Furthermore, note that any instance variable that has not previously been assigned to will silently evaluate to `nil`.

<https://gist.github.com/8d6e400be9b2c2b73dd9635070114cb6>

```

1 # Time#now, Time#+ and Time#- represent time as 'seconds since 1/1/70'
2 class Fixnum
3   def seconds ; self ; end
4   def minutes ; self * 60 ; end
5   def hours   ; self * 60 * 60 ; end
6   def ago    ; Time.now - self ; end
7   def from_now ; Time.now + self ; end
8 end
9 Time.now           # => 2018-11-22 16:58:04 +0100
10 5.minutes.ago     # => 2018-11-22 16:53:12 +0100
11 5.minutes - 4.minutes # => 60
12 3.hours.from_now  # => 2018-11-22 19:58:45 +0100

```

Figure 2.6: By reopening Ruby’s core `Fixnum` class and adding six new instance methods to it, we get a beautiful syntax for time arithmetic. (Rails includes a more complete version of this facility.) Unix was invented in 1970, so its designers chose to represent time as the number of seconds since midnight (GMT) 1970-01-01, sometimes called the beginning of the *epoch*.

Summary:

- Everything in Ruby is an object, even primitive types like integers. Ruby objects have types, but the variables that refer to them don’t.
- Every operation is performed by calling a method on an object; the notation `a.b` means “call method `b` on object `a`.” Object `a` is said to be the *receiver*, and if it cannot handle the method call, it will pass the call to its superclass. This process is called *looking up a method* on a receiver.
- Every Ruby statement is an expression that has a well-defined value (which may be `nil`).
- Ruby has classes and single inheritance, with the usual instance and class methods and variables.
- Important Ruby idioms include the use of symbols, the use of keyword-based arguments to methods, and poetry mode, which allows omitting parentheses around method arguments and curly braces surrounding a hash when the resulting code is syntactically unambiguous.
- An idiomatic primitive type in Ruby is the symbol, an immutable string whose value is itself. Symbols are commonly used in Ruby to denote “specialness,” such as being one of a set of fixed choices like an enumeration, and to pass named (keyword) arguments to methods.
- Ruby has comprehensive **reflection**, allowing you to ask objects about themselves.
- `attr_accessor` is an example of metaprogramming: it creates new code at runtime, in this case getters and setters for an instance variable. This style of metaprogramming is extremely common in Ruby.



■ **Elaboration: In Ruby, all programming is metaprogramming**

`attr_accessor` is an example of **metaprogramming**—creating code at runtime that defines new methods—because `attr_accessor` is not built into the Ruby language, but instead is a regular method call that defines the getter and setter methods on the fly. That is, `attr_accessor :foo` defines instance methods `foo` and `foo=` that get and set the value of instance variable `@foo`. In fact, in a sense *all* Ruby programming is metaprogramming, since even the class definition in Figure 2.5 is not a declaration as it is in Java but actually code that is executed at runtime, creating a new object representing the `Movie` class and binding the name `Movie` to it. Going further, since defining a class happens at runtime, you can modify it later as well, as Figure 2.6 shows.

Self-Check 2.3.1. *What is the explicit-send equivalent of each of the following expressions: `a<b`, `a==b`, `x[0]`, `x[0]='foo'`.*

- ◊ `a.send(:<, b)`, `a.send(:==, b)`, `x.send(:[], 0)`, `x.send(:[]=, 0, 'foo')` ■

Self-Check 2.3.2. *Verify in an interactive Ruby interpreter that `5/4` gives 1, but `5/4.0` and `5.0/4` both give 1.25. Explain this behavior by identifying which class's / method is called in each case, and how you think it handles its argument.*

- ◊ In `5/4` and `5/4.0`, the `Integer` class's / instance method is called on the receiver `5`. That method performs integer division if its argument is also an integer, but if its argument is a float, it converts the receiver to a float and performs floating-point division. In `5.0/4`, the `Float` class's / method is called, which always performs floating-point division. ■

Self-Check 2.3.3. *Why is `movie.@year=1998` not a substitute for `movie.year=1998`?*

- ◊ The notation `a.b` always means “call method `b` on receiver `a`”, but `@year` is the name of an instance variable, whereas `year=` is the name of an instance method. ■

Self-Check 2.3.4. *Suppose we delete line 12 from Figure 2.5. What would be the result of executing `Movie.new('Inception', 2011).year`?*

- ◊ Ruby would complain that the `year` method is undefined. ■

Self-Check 2.3.5. *In Figure 2.6, is `Time.now` a class method or an instance method?*

- ◊ The fact that its receiver is a class name (`Time`) tells us it's a class method. ■

Self-Check 2.3.6. *Why does `5.superclass` result in an “undefined method” error? (Hint: consider the difference between calling `superclass` on `5` itself vs. calling it on the object returned by `5.class`.)*

- ◊ `superclass` is a method defined on classes. The object `5` is not itself a class, so you can't call `superclass` on it. ■

Self-Check 2.3.7. *Which of the following Ruby expressions are equal to each other: (a) `:foo` (b) `%q{foo}` (c) `%Q{foo}` (d) `'foo'`.`to_sym` (e) `:foo.to_s`*

- ◊ (a) and (d) are equal to each other; (b), (c), and (e) are equal to each other ■

Self-Check 2.3.8. *What is captured by `$1` when the string `25 to 1` is matched against each of the following regexps:*

- (a) `/(\d+)/$`
- (b) `/^|\d+([^\0-9]+)/`

- ◊ (a) the string “1” (b) the string “ to ” (including the leading and trailing spaces) ■

<https://gist.github.com/c0ec0b8c12c435990319416c056340a3>

```
1 link_to('Edit', {:controller => 'students', :action => 'edit'})
2 link_to 'Edit', :controller => 'students', :action => 'edit'
3 link_to 'Edit', controller: 'students', action: 'edit'
```

Figure 2.7: Three legal and equivalent calls to the method `link_to` (which we'll meet in Section 4.4) that takes one string argument and one hash argument. The first is fully parenthesized, the second omits the parentheses around the call arguments and the curly braces around the final hash argument, and the third uses the alternative (Ruby ≥ 2.0) syntax for the hash keys in the second argument.

Self-Check 2.3.9. Consider line 18 of Figure 2.5. Explain why the following would be an acceptable alternative way to define the `full_title` method, and the pros and cons compared to the way it appears in the figure:

```
def full_title ; "#title (#year)"; end
```

- ◊ This version calls the accessor methods `title` and `year` rather than accessing the instance variables directly. Doing so decouples the implementation of this method from the implementations of the underlying state of the movie (title and year). ■

2.4 Ruby Idioms: Poetry Mode, Blocks, Duck Typing

A **programming idiom** is a way of doing or expressing something that occurs frequently in code written by experienced users of a given programming language. While there may be other ways to accomplish the same task, the idiomatic way is the one that is most readily intention-revealing to other experienced users of the language. Your goal when learning a new language should be to learn to “think in” that language by understanding and using its idioms well, or in other words, to avoid the well-known pitfall that “you can write FORTRAN in any language”⁶. In this section we explore three key Ruby idioms: passing arguments to methods (“poetry mode” and named parameters), blocks, and duck typing.

Poetry mode and named parameters. Figures 2.7 and 2.8 show two pervasive idioms related to Ruby method calls. The first, *poetry mode*, allows omitting parentheses around the arguments to a method call when the parsing is unambiguous. In addition, when the *last* argument to a method call is a hash, the curly braces around the hash literal can be omitted.

In early versions of Ruby, hash arguments were often used to emulate the **named parameter** feature (also called *keyword arguments*) available in languages such as Python, C#, and others. For example, the documentation for the `link_to` method used in Figure 2.7 tells us that `:controller` and `:action` are just two of many possible additional (and optional) values that can be passed to the method as keys in a hash. True named parameters became available in Ruby 2.0, as Figure 2.8 shows; nonetheless, a great deal of Ruby code written prior to Ruby 2.0 still uses hashes to pass optional arguments or provide default values for arguments.

Blocks. Ruby uses the term *block* somewhat differently than other languages do. In Ruby, a block is just a method without a name, or an **anonymous lambda expression** in programming-language terminology. Like a regular named method, it has arguments and can use local variables.

As Figure 2.9 shows, one of the most common uses of blocks is to implement data structure traversal. The instance method `each`, available in all Ruby classes that are collection-like, takes a single argument consisting of a block (anonymous lambda) to which each member of the collection will be passed. `each` is an example of an **internal iterator**. Rubyists

<https://gist.github.com/4b32d51d724a86029604539dd465c7d6>

```

1 # Using 'named keyword' arguments
2 def greet(name, last_name: "", greeting: "Hi")
3   "#{greeting}, #{name} #{last_name}!"
4 end
5 greet("Dave")           # => "Hi, Dave! "
6 greet("Dave", last_name: "Fox") # => "Hi, Dave Fox!"
7 greet("Dave", greeting: "Yo")  # => "Yo, Dave!"
8 greet("Dave", greeting: "Hey", last_name: "Patterson")
9   # => "Hey, Dave Patterson!" - order of keyword args irrelevant
10 greet(greeting: "Yo")      # ArgumentError, since first arg is
    required

```

Figure 2.8: The use of *keyword arguments* or named parameters allows you to define methods in which some arguments are optional or assume default values. Named parameters can improve clarity for methods that take multiple arguments, though we will see in Chapter 9 that one should usually minimize the number of arguments a method accepts.

<https://gist.github.com/c3ea17d28f213c3e08e5503da3f210cf>

```

1 def print_movies(movie_list)
2   movie_list.each do |m|
3     puts "#{m.title} (rated: #{m.rating})"
4   end
5 end

```

Figure 2.9: `each` takes one argument—a block—and passes each element of the collection to the block in turn. A block is bracketed by `do` and `end`, and any arguments expected by the block are enclosed in |pipe symbols| after the `do`. Each time through the block, `m` is set to the next element of `movie_list`.

like to say that Ruby collections “manage their own traversal,” because it’s up to the receiver of `each` to decide how to implement that method to yield each collection element. (Indeed, in Figure 2.9, we can’t even tell what the underlying type of `movie_list` is.)

Internal iterators first appeared in the research language CLU, an early vehicle for research on data hiding that garnered a Turing Award for Barbara Liskov.

Figure 2.10 shows a simple example of such a collection operator, which can be used with any collection that implements `each` as a way of traversing itself. Note once again that we have no idea how the collection is implemented: all we need to know is that it implements the instance method `each` to enumerate its elements. Ruby provides a wide variety of such collection methods; Figure 2.11 lists some of the most useful. With some practice, you will automatically start to express operations on collections in terms of these functional idioms rather than in terms of imperative loops. Although Ruby allows `for i in collection`, `each` allows us to take better advantage of **duck typing**, which we’ll see shortly, to improve code reuse.



Duck Typing. You may be surprised to learn, though, that the collection methods summarized in Figure 2.11 (and several others not in the figure) aren’t part of Ruby’s `Array` class. In fact, they aren’t even part of any superclass from which `Array` and other collection types inherit. Instead, they take advantage of an even more powerful reuse mechanism: A **mix-in** is a named collection of related methods that can be added to any class fulfilling some “contract” with the mixed-in methods. A *module* is Ruby’s method for packaging together a group of methods as a mix-in. The Ruby statement `include ModuleName` inside a class definition mixes the instance methods, class methods, and variables of the module into that class. The collection methods in Figure 2.11 are defined in a module called `Enumerable` that is part of Ruby’s standard library and is mixed in to all of Ruby’s collection classes. As its documentation⁷ states, `Enumerable` requires the class mixing it in to provide an `each` method, since `Enumerable`’s collection methods are implemented in terms of `each`. It doesn’t matter what class you mix it into as long as that class defines the `each` instance method, and neither

<https://gist.github.com/3f068204f819ace57403adfd66652424>

```

1 # find largest element in a collection
2 def maximum(collection)
3   result = collection.first
4   collection.each do |item|
5     result = item if item > result
6   end
7   result
8 end
9 maximum([3,4,2,1])      # => 4
10 maximum(["a","x","b"]) # => "x"
11 max([RomanNumeral.new('XL'), RomanNumeral.new('LI')]) # => 'LI'
12
13 class RomanNumeral
14   include Comparable
15   def initialize(roman_numeral_string)
16     @orig_string = roman_numeral_string
17     @value = RomanNumeral.convert_from_roman(roman_numeral_string)
18   end
19   def <=(other)
20     @value <= other
21   end
22   def to_s
23     @orig_string
24   end
25   def self.convert_from_roman(str)
26     # ...code to convert Roman numerals from strings...
27   end
28 end

```

Figure 2.10: This example finds the maximum-valued element in any collection that responds to `each`, and is agnostic to the type(s) of the element(s) in the collection as long as they respond to `>`. It even works on Roman numerals if we have a `RomanNumeral` class that either defines `>` explicitly, or defines `<=>` and mixes in the `Comparable` module to define `<`, `>`, and so on.

the class nor the mix-in have to declare their intentions in advance. For example, the `each` method in Ruby’s `Array` class iterates over the array elements, whereas the `each` method in the `I/O` class iterates over the lines of a file or other I/O stream. Mix-ins thereby allow reusing whole collections of behaviors across classes that are otherwise unrelated.

Similarly, a class that defines the “spaceship operator” `<=>`, which returns `-1, 0, 1` depending on whether its second argument is less than, equal to, or greater than its first argument, can mix in the `Comparable` module, which defines `<`, `<=`, `>`, `>=`, `==`, and `between?` in terms of `<=>`. For example, the `Time` class defines `<=>` and mixes in `Comparable`, allowing you to write `Time.now.between?(Time.parse("19:00"), Time.parse("23:15"))`.

The term “duck typing” is a popular description of this capability, because “if something looks like a duck and quacks like a duck, it might as well be a duck.” From `Enumerable`’s point of view, if a class has an `each` method, it might as well be a collection, thus allowing `Enumerable` to provide other methods implemented in terms of `each`. When Ruby programmers say that some class “quacks like an `Array`,” they usually mean that it’s not necessarily an `Array` nor a descendant of `Array`, but it responds to most of the same methods as `Array` and can therefore be used wherever an `Array` would be used.



Method	Block?	Returns a <i>new</i> collection containing...
<code>c.map</code>	1	elements obtained by applying block to each element of <code>c</code>
<code>c.select</code>	1	Subset of <code>c</code> for which block evaluates to true
<code>c.reject</code>	1	Subset of <code>c</code> obtained by removing elements for which block evaluates to true
<code>c.uniq</code>		all elements of <code>c</code> with duplicates removed
<code>c.reverse</code>		elements of <code>c</code> in reverse order
<code>c.compact</code>		all non- <code>nil</code> elements of <code>c</code>
<code>c.flatten</code>		elements of <code>c</code> and any of its sub-arrays, recursively flattened to contain only non-array elements
<code>c.partition</code>	1	Two collections, the first containing elements of <code>c</code> for which the block evaluates to true, and the second containing those for which it evaluates to false
<code>c.sort</code>	2	Elements of <code>c</code> sorted according to a block that takes 2 arguments and returns -1 if the first element should be sorted earlier, +1 if the second element should be sorted earlier, and 0 if the two elements can be sorted in either order.
The following methods require the <i>collection elements</i> to respond to <code><=></code> ; see Section 2.4.		
<code>c.sort</code>		If <code>sort</code> is called <i>without</i> a block, the elements are sorted according to how they respond to <code><=></code> .
<code>c.sort_by</code>	1	Applies the block to each element of <code>c</code> and sorts the result. For example, <code>movies.sort_by { m m.title }</code> sorts <code>Movie</code> objects according to how their titles respond to <code><=></code> .
<code>c.max, c.min</code>		Largest or smallest element in the collection

Figure 2.11: Some common Ruby methods on collections. For those that expect a block, the “Block” column shows the number of arguments expected by the block; if blank, the method doesn’t expect a block. For example, a call to `sort`, whose block expects 2 arguments, might look like: `c.sort { |a,b| a <=> b }`. These methods all return a new object rather than modifying the receiver, but some methods also have a *destructive* variant ending in `!`, for example `sort!`, that modify the receiver in place (and also return the new value). Use destructive methods with extreme care, if at all.

Summary

- Poetry mode allows omitting parentheses around method call arguments, and omitting curly braces around a hash literal when it is the last argument to a method call. Hash arguments were previously used to emulate named parameters or keyword arguments, but this practice is now discouraged since Ruby supports true named parameters starting with version 2.0.
- Ruby borrows great ideas from ***functional programming***, especially the use of *blocks*—parameterized chunks of code called ***lambda expressions*** that carry their scope around with them, making them ***closures***.
- Because of Ruby’s ***dynamic typing***, calling a method on an object is legal as long as the receiver responds to the method, regardless of the receiver’s class, a behavior sometimes called ***duck typing***.
- A ***mix-in*** takes advantage of duck typing by allowing a set of related behaviors to be added to any class that satisfies the mix-in’s contract; for example, `Enumerable` just requires the including class to respond to `each`. A module is mixed into a class by putting `include ModuleName` after the `class ClassName` statement.
- Unlike interfaces in Java, mix-ins require no formal declaration. But because Ruby doesn’t have static types, it’s your responsibility to ensure that the class including the mix-in satisfies the conditions stated in the mix-in’s documentation, or you will get a runtime error.

■ Elaboration: Blocks Are Closures

The combination of blocks and iterators like `each`, which is how most Ruby operations on collections are expressed, is a technique Ruby borrows from ***functional programming***. A Ruby block is a ***closure***: whenever the block executes, it can “see” the entire lexical scope available at the place where the block appears in the program text. In other words, it’s as if the presence of the block creates an instant snapshot of the scope, which can be reconstituted later whenever the block executes. This fact is exploited by many Rails features that improve DRYness, including view rendering (which we’ll see in Section 4.4) and model validations and controller filters (Section 5.1), because they allow separating the definition of *what* is to occur from *when* in time and *where* in the structure of the application it occurs.



Self-Check 2.4.1. Write one line of Ruby that checks whether a string `s` is a palindrome, that is, it reads the same backwards as forwards. **Hint:** Use the methods in Figure 2.11, and don’t forget that upper vs. lowercase shouldn’t matter: `ReDivider` is a palindrome.

◊ `s.downcase == s.downcase.reverse`

You might think you could say `s.reverse=~Regexp.new(s)`, but that would fail if `s` happens to contain regexp metacharacters such as `$`. ■

Self-Check 2.4.2. Suppose you mix `Enumerable` into a class `Foo` that does not provide the `each` method. What error will be raised when you call `Foo.new.map { |elt| puts elt }`?

◊ The `map` method in `Enumerable` will attempt to call `each` on its receiver, but since the new `Foo` object doesn’t define `each`, Ruby will raise an Undefined Method error. ■

```
https://gist.github.com/2b1976e79ef3c4b6bd6bb6b8e6dd54ab
1 require "stripe"          # makes gem's definitions available within this file
2 Stripe.api_key = "sk_test_4eC39HqLyjWDarjtT1zdp7dc"
3 Stripe::Charge.create({
4   :amount => 2000,
5   :currency => "usd",
6   :source => "tok_mastercard", # obtained with Stripe.js
7   :description => "Charge for jenny.rosen@example.com"
8 })
```

Figure 2.12: To use a gem, you must install it either directly (as in `gem install 'stripe'`) or preferably via the Bundler tool discussed next. `gem help` gives brief help for the command-line tool that manages installed gems manually; by default it installs gems from the RubyGems website¹⁰, the definitive gem repository.

Self-Check 2.4.3. Which statement is correct and why: (a) `include 'enumerable'` (b) `include Enumerable`

- ◊ (b) is correct, since `include` expects the name of a module, which (like a class name) is a constant rather than a string. ■

2.5 CHIPS: Ruby Intro



CHIPS 2.5: Ruby Intro

<https://github.com/saasbook/hw-ruby-intro>

Write and run Ruby code that exercises the basics of control flow, classes and inheritance, accessor methods, regular expression and string manipulation, symbols, and common uses of blocks such as iterators and collection idioms. Learn your way around the interactive Ruby interpreter `irb` and the debugger `byebug`. Familiarize yourself with RSpec, a tool for creating automated tests, by writing code to get our provided tests to pass.

2.6 Gems and Bundler: Library Management in Ruby

 **Libraries.** The Ruby standard library⁸ includes a large number of useful classes covering file and network input/output, time and date manipulation, manipulating strings and collections, and more.

An external library is packaged as a Ruby *gem*, a collection of classes with well-defined interfaces. Gems can be as simple as augmenting existing classes with a few utility functions, or as complex as an entire framework: Rails itself is distributed as a gem that depends on several other gems. Similar to Python `import`, `require` makes a gem's classes and functions available within a file of Ruby code, as Figure 2.12 shows.

Where Ruby really shines, though, is in managing dependencies among gems. GitLab, a popular open-source application written in Rails, relies on around 400 gems. Since some of those rely in turn on other gems, all in all GitLab depends on over 800 gems, many of which are constantly evolving. It's therefore critical to specify which version(s) of libraries an app has been developed and tested with, so that when the app is deployed or distributed, it behaves the same way in every environment in which it's run.

To manage complex dependencies, we need a dependency manager or package manager, such as `pip` for Python, `npm` for Node.js, or Apache Maven for Java. Ruby's package man-



<code>gem install name [-v version]</code>	Install version <i>version</i> (default: latest) of gem named <i>name</i> . The default location for downloading gems is <code>rubygems.org</code> .
<code>bundle install [--without env]</code>	If <code>Gemfile</code> has not changed, inspect <code>Gemfile.lock</code> and ensure that the correct version(s) of gem(s) specified there are installed. If <code>Gemfile</code> has changed, recompute the dependency graph to regenerate <code>Gemfile.lock</code> , then ensure correct gems are installed. If <code>--without</code> is given, do not try to install gems associated with environment <i>env</i> .
<code>bundle update gemnames</code>	Force Bundler to update <i>gemnames</i> to their latest major versions, and recompute dependencies. Passing <code>--all</code> for <i>gemnames</i> forces updating all gems, ignoring and regenerating <code>Gemfile.lock</code> .
<code>bundle exec command</code>	Execute <i>command</i> in the context of the current bundle, that is, make sure the correct version(s) of gem(s) are loaded and active before <i>command</i> runs. For example, if <i>command</i> relies a particular version of some gem, but you have other version(s) of that gem also installed, without <code>bundle exec</code> the wrong version of the gem may be active when <i>command</i> is run.
<code>bundle help</code>	Show more detailed help; also see Bundler's website ¹¹ .

Figure 2.13: Frequently used commands for working with gems and Bundler. We will learn about Bundler “environments” in Section 4.1.

ager, Bundler, is itself a gem. Once Bundler is installed with `gem install bundler`, you should allow it to do your dependency management.

To use Bundler, a Ruby project should have a file called `Gemfile` in its top-level directory that records the dependencies of the app on particular libraries. Bundler reads this file and tries to compute a set of library version(s) that respects all the constraints in the file. For example, if the app depends on version ≥ 3.0 and ≤ 4.0 for library X, but the app also depends on library Y which requires version 3.5 of library X, then version 3.5 of library X will be installed. Bundler can also detect when it’s impossible to satisfy all the constraints. In general, when you start a new Ruby project you immediately create a `Gemfile` for it, and when you download someone else’s Ruby project to work on, you first run `bundle install` in the project’s main directory to allow Bundler to locate and download all the necessary libraries.

Bundler then arranges to install all needed gems with their proper versions, and records the results in `Gemfile.lock`. Both `Gemfile` and `Gemfile.lock` should be stored as part of the codebase, since the latter records which versions of which libraries were *actually* used in development, whereas `Gemfile` just specifies constraints on which versions *could* be compatibly used.

Increasingly, library version numbers follow semantic versioning¹², not just for Ruby gems but in other languages as well. The usual arrangement is for a version number to be formatted as `major.minor.patch`, where each field is an integer, such as 2.3.1. Changes in the value of `patch` are usually minor, backwards-compatible bug fixes, including security patches, that do not change the semantics or functionality of the gem. Changes in `minor` usually indicate that functionality has been *added* in a backward-compatible manner. Changes in `major` signal that the Application Programming Interface (API)—the way you call the library’s functions—has changed in a way that may break compatibility with previous versions.

Since breaking compatibility is a major decision that may affect thousands of apps using

a library, a common practice is for such changes to first appear as **deprecation** warnings in a new minor version or patch version. Such warnings typically manifest as messages emitted at build time or run time to the effect of “Warning: this feature will work differently [or be dropped] in the next major release of this library.” As a general rule, deprecation warnings become errors when the major version changes. The prudent developer faced with a deprecation warning will therefore read the documentation and determine if there is a way to change the *current* code so that it uses the soon-to-be-new version of the feature or of the feature’s API.

Indeed, when upgrading to a new major version, a best practice is to first incrementally upgrade so that you can identify and address deprecations before the major version change. For example, suppose your app uses version 2.7.3 of the Foobaz gem, but the latest version is 3.1.5, and you haven’t been keeping up:

1. First, update to the latest whose major version is still 2—let’s say that turns out to be 2.8.1.
2. Identify and address any deprecation warnings generated by that upgrade.
3. Now upgrade to the *first* release whose major version is 3. In all likelihood this is 3.0.0, but might be different. Ensure all works well with the new major version.
4. Next, upgrade to the latest *minor* version—in our example, probably 3.1.0. Ensure all works well.
5. Finally, upgrade to 3.1.5, the latest patch release.

Bundler uses `~>2` to mean “last release whose the major version is 2”, or `~>2.4` to mean “last release whose major and minor version is 2.4”. So `2.4.6` would satisfy both constraints, whereas `2.5` would satisfy the first but not the second.

Of course, in an ideal world, the developer has been gradually updating the gem over time, so it is not necessary to do all these steps at once.

In Chapter 8, we will have a lot to say about “ensuring all works well,” as this is one of the key roles of having a solid test suite.

Summary of libraries and dependency management in Ruby:

- Besides a language's standard library, which usually ships with the language distribution, most languages enjoy vast external libraries of contributed code. In Ruby, external libraries are distributed as Ruby gems.
- Keeping track of which version(s) of which libraries an app depends on is critical. The Bundler tool largely automates this by allowing the developer to express constraints on library versions and then figuring out a set of versions that meet all the constraints.
- Most libraries are semantically versioned as `major.minor.patch`, where patches fix bugs, minor releases add new functionality, and major releases overhaul the library in ways that may break compatibility with previous major releases.
- To prepare developers for such breaking changes, updated patch or minor versions will often display deprecation messages warning the developer about the use of a feature that is about to change incompatibly. Frequently upgrading maximizes the likelihood you can see and act on deprecations before they become errors; the Gemfile you prepare for Bundler specifies what "safe" upgrades are permitted.

2.7 Fallacies and Pitfalls

***Pitfall: Always watching the driver while pair programming.***

If one member of the pair has much more experience, the temptation is to let the more senior member do all the driving, with the more junior member becoming essentially the permanent observer. This relationship is not healthy, and will likely lead to disengagement by the junior member.

***Pitfall: Blindly following “cookbook” tutorials when learning a new language.***

Computer science legend and Turing Award winner Donald Knuth, who literally wrote the book(s) on the foundations of theoretical computer science, says that writing code is “harder than anything else I’ve ever had to do.” And Peter Norvig, Google’s Director of Research, has eloquently said¹³ that there is no shortcut around such a challenge: it requires deep study and lots of ongoing practice. Following a step-by-step tutorial without an understanding of the underlying mechanisms being explained will put something on the screen quickly, but you won’t understand how it got there nor be able to replicate this success with your own apps.

***Pitfall: Forgetting that the compiler won’t save you.***

In strongly-typed or statically-typed languages, the compiler can usually detect if a variable of one type is erroneously being assigned a value of an incompatible type, for example, writing `x=foo()` where `foo` returns a numeric value but `x` has been declared as a string variable. In weakly-typed or dynamically-typed languages (Ruby is both), there are no such “compile-time” checks—instead you’ll get a runtime error. So you must be that much more

careful in your testing and in the design of your code. Chapter 8 will introduce techniques for ensuring your code is well tested. The debate over the relative merits of static vs. dynamic typing is a long-running “holy war”¹⁴ among programmers that we won’t wade into here.



Pitfall: Writing Python in Ruby.

It takes some mileage to learn a new language’s idioms and how it fundamentally differs from other languages. Common examples for Python programmers new to Ruby include:

- Reading an expression such as `person.age` as “the age attribute of the person object” rather than “call the instance method `age` on the object `person`.”
- Thinking that `person.age=40` is an *assignment to an attribute* when in fact it is a method call. In fact, the `age=` method called on `person` is passed the argument (40), and can *do whatever it wants*. Ruby code often uses this mechanism as a way to provide syntactic sugar for “assignments” that cause side effects.
- Forgetting that any instance variable that has not previously been assigned will silently evaluate to `nil`.
- Thinking of `attr_accessor` as a declaration of attributes. This shortcut and related ones save you work *if* you want to make an attribute publicly readable or writable. But you don’t need to “declare” an attribute in any way at all (the existence of the instance variable is sufficient) and in all likelihood some attributes *shouldn’t* be publicly visible. Resist the temptation to use `attr_accessor` as if you were writing attribute declarations in Java.
- Writing explicit for-loops rather than using an iterator such as `each` and the collection methods that exploit it via mix-ins such as `Enumerable`. Use functional idioms like `select`, `map`, `any?`, `all?`, and so on.
- Using `lowerCamelCase` rather than `snake_case` to name variables. It seems trivial, but experienced programmer find it jarring to read code that violates the typographical conventions of a language, just as experienced musicians wince when they hear a note played out of tune. If in doubt, find other Ruby code with an example of what you want to do, and emulate it.



Pitfall: Thinking of symbols and strings as interchangeable.

While many Rails methods are explicitly constructed to accept either a string or a symbol, the two are not in general interchangeable. A method expecting a string may throw an error if given a symbol, or depending on the method, it may simply fail. For example, `['foo','bar'].include?('foo')` is truthy, whereas `['foo','bar'].include?(:foo)` is falsy.



Pitfall: Naming a local variable when you meant a local method.

Suppose class C defines a method `x=`. In an instance method of C, writing `x=3` will not have the desired effect of calling the `x=` method with the argument 3; rather, it will set a local variable `x` to 3, which is probably not what you wanted. To get the desired effect, write `self.x=3`, which makes the method call explicit.



Pitfall: Confusing require with include.

`require` loads an arbitrary Ruby file (typically the main file for some gem), whereas `include` mixes in a module. In both cases, Ruby has its own rules for locating the files containing the code; the Ruby documentation describes the use of `$LOAD_PATH`, but you should rarely if ever need to manipulate it directly if you use Rails as your framework and Bundler to manage your gems.

2.8 Concluding Remarks: How (Not) To Learn a Language By Googling

Your authors will allow themselves a literary flourish and frame the advice in this section with two quotes. The first is from Tom Knight, one of the principal designers of Lisp machines—research computers designed at MIT to optimize running programs in the Lisp programming language.

A novice was trying to fix a broken Lisp machine by turning the power off and on. Knight, seeing what the student was doing, spoke sternly: “You cannot fix a machine by just power-cycling it with no understanding of what is going wrong.” Knight turned the machine off and on. The machine worked.

—“AI Koans” section of The New Hacker’s Dictionary

To us, the above quote captures the perils of blindly copy-pasting code without understanding how it works: the code may appear to work initially, but if you don’t know why, or if it breaks something else, you may get into trouble in the future, and you certainly won’t learn much. While programmer Q&A sites such as StackOverflow¹⁵ are invaluable for both asking questions and discovering code snippets to perform specific tasks, your strategy should be to find a general *pattern* matching what you’re trying to do, then once you fully understand how the found example works, *adapt* it for your specific situation. In other words, search for *how, not what*.

The second quote is from the author of *The Cathedral and the Bazaar* (Raymond 2001), an early exposition of the potential advantages of open-source development:

Ugly programs are like ugly suspension bridges: they’re much more liable to collapse than pretty ones, because the way humans (especially engineer-humans) perceive beauty is intimately related to our ability to process and understand complexity. A language that makes it hard to write elegant code makes it hard to write good code.

—Eric S. Raymond

Learning to use a new language and making the most of its idioms is a vital skill for software professionals. These are not easy tasks, but we hope that focusing on unique and beautiful features in our exposition of Ruby and JavaScript will evoke intellectual curiosity rather than groans of resignation, and that you will come to appreciate the value of wielding a variety of specialized tools and choosing the most productive one for each new job.

If this is your first exposure to Ruby, then functional programming and blocks in Ruby and closures in JavaScript may take some getting used to. But as with any language, not learning to use the idioms properly may result in missing the opportunity to use a mechanism in the new language that might provide a more beautiful solution.

Our advice is therefore to persevere in a new language until you’re comfortable with its idioms. Resist the temptation to “transliterate” your code from other languages without first



considering whether there's a more idiomatic way to express what you need in the target language.

If you want to expand your “programming language cross training” program, we recommend *Seven Languages In Seven Weeks* (Tate 2010), which introduces the reader to a set of languages that invite radically different ways of thinking about expressing programming tasks.

Finally, we recommend these resources for more detail on Ruby itself. Again, we assume that Ruby is not your first language and that this book and course are not your first exposure to programming, so we omit references aimed at beginners.

- Programming Ruby¹⁶ and *The Ruby Programming Language* (Flanagan and Matsumoto 2008), co-authored by Ruby inventor Yukihiro “Matz” Matsumoto, are definitive references for Ruby.
- The online documentation for Ruby¹⁷ gives details on the language, its classes, and its standard libraries. A few of the most useful classes include `IO` (file and network I/O, including CSV files), `Set` (collection operations such as set difference, set intersection, and so on), and `Time` (the standard class for representing times, which we recommend over `Date` even if you’re representing only dates without times). These are reference materials, not a tutorial.
- *Learning Ruby* (Fitzgerald 2007) takes a more tutorial-style approach to learning the language.
- *The Ruby Way* is an encyclopedic reference to both Ruby itself and how to use it idiomatically to solve many practical programming problems.
- *Ruby Best Practices* (Brown 2009) focuses on how to make the best of Ruby’s “power tools” like blocks, modules/duck-typing, metaprogramming, and so on. If you want to write Ruby like a Rubyist, this is a great read.
- Many newcomers to Ruby have trouble with `yield`, which has no equivalent in Java, C or C++ (although recent versions of Python and JavaScript do have similar mechanisms). The ***coroutines*** article on Wikipedia gives good examples of the general coroutine mechanism that `yield` supports.

A. Begel and N. Nagappan. Pair programming: What’s in it for me? In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 120–128, Kaiserslautern, Germany, October 2008.

G. T. Brown. *Ruby Best Practices*. O'Reilly Media, 2009. ISBN 0596523009.

A. Cockburn and L. Williams. The costs and benefits of pair programming. *Extreme Programming Examined*, pages 223–248, 2001.

M. J. Fitzgerald. *Learning Ruby*. O'Reilly Media, 2007. ISBN 0596529864.

D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly Media, 2008. ISBN 0596516177.

J. Hannay, T. Dyba, E. Arisholm, and D. Sjoberg. The effectiveness of pair programming: A meta-analysis. *Information and Software Technology*, 51(7):1110–1122, July 2009.

L. Hoffmann. Q&a: Big challenge. *Communications of the ACM (CACM)*, 56(9):112–ff, Sept. 2013.

J. Moore. ipad 2 as a remote presence device? *Pivotal Labs*, 2011. URL <http://pivotallabs.com/blabs/categories/pair-programming>.

E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, Inc., 2001.

F. J. Rodríguez, K. M. Price, and K. E. Boyer. Exploring the pair programming process: Characteristics of effective collaboration. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 507–512, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346986. doi: 10.1145/3017680.3017748. URL <https://doi.org/10.1145/3017680.3017748>.

B. Tate. *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages (Pragmatic Programmers)*. Pragmatic Bookshelf, 2010. ISBN 193435659X. URL <https://www.amazon.com/Seven-Languages-Weeks-Programming-Programmers/dp/193435659X>.

Notes

¹<https://rubular.com>

²<http://dilbert.com/strips/comic/2003-01-09/>

³<http://dilbert.com/strips/comic/2003-01-11/>

⁴<https://ace.c9.io>

⁵<https://sublimetext.com>

⁶<https://queue.acm.org/detail.cfm?id=1039535>

⁷<http://ruby-doc.org/core-2.5.1/Enumerable.html>

⁸<https://ruby-doc.org/stdlib>

⁹<https://rubygems.org>

¹⁰<https://rubygems.org>

¹¹<https://bundler.io>

¹²<https://semver.org/spec/v2.0.0.html>

¹³<http://norvig.com/21-days.html>

¹⁴<https://wiki.c2.com/?HolyWar>

¹⁵<https://stackoverflow.com>

¹⁶<http://ruby-doc.org/docs/ProgrammingRuby>

¹⁷<http://ruby-doc.org/>

3

SaaS Application Architecture: Microservices, APIs, and REST

Dennis Ritchie (1941–2011) and Ken Thompson (1943–) were co-recipients of the 1983 Turing Award for fundamental contributions to operating systems design in general and the invention of Unix in particular.



I think the major good idea in Unix was its clean and simple interface: open, close, read, and write.

—*Unix and Beyond: An Interview With Ken Thompson*, IEEE Computer 32(5), May 1999

3.1	The Web's Client–Server Architecture	74
3.2	SaaS Communication Uses HTTP Routes	76
3.3	CHIPS: HTTP and URIs	80
3.4	From Web Sites to Microservices: Service-Oriented Architecture	80
3.5	RESTful APIs: Everything is a Resource	84
3.6	RESTful URIs, API Calls, and JSON	89
3.7	CHIPS: Create and Deploy a Simple SaaS App	92
3.8	Fallacies and Pitfalls	92
3.9	Concluding Remarks: Continuity From CGI to SOA	94

Prerequisites and Concepts

Concepts:

- SaaS apps follow the *client-server* pattern, in which a client makes requests and a server responds to the requests of many clients.
- Fundamental to the Web's architecture are the Hypertext Transfer Protocol (HTTP), a request-reply protocol over which Web content is delivered, and Universal Resource Identifiers (URIs), which name a particular *resource* on the Web and may specify parameters (options) for accessing it. A basic building block of SaaS is an HTTP *route*, which combines a *method* such as GET or POST with a URI.
- Over time, the Web shifted from a collection of static resources to a collection of programs (services) that could be remotely accessed via either a Web browser or another SaaS app. This shift towards *service-oriented architecture* laid the groundwork for the explosion of *microservices*.
- An Application Programming Interface (API) for a microservice is a formal description of the operations the microservice can do. The “API first” design stance suggests that even when creating a large SaaS app, we get a more modular design by thinking about the app in terms of its smaller components and the APIs by which they communicate.
- Since most mobile apps are just SaaS clients that access a SaaS app in the same way a microservice would, the API-first design stance espoused in this chapter is an ideal complement to the mobile-first design stance in Chapter 1.

3.1 The Web’s Client–Server Architecture

Every time you use a Web browser to visit a site, or use a mobile app that also makes use of the cloud (such as when a weather app downloads the latest weather forecasts), you are using a Software-as-a-Service (SaaS) *client* to make one or more *requests* of a SaaS *server*. SaaS based on Web protocols is the most widely deployed example of a **client-server architecture**: clients are programs whose specialty is asking servers for information and (usually) allowing the user to interact with that information, and servers are programs whose specialty is efficiently serving large numbers of clients simultaneously.

Modern SaaS clients can take many forms. Whether you visit Google Maps using a browser on your PC, a browser on a smartphone, or a smartphone native app, you’re using a SaaS client. And while the clients differ in how they present and let you interact with Google Maps since each is specialized to its task, all three are communicating with the same Google Maps SaaS service.

In contrast to the client software, which is typically a discrete app running on a single device such as a PC or smartphone, the “server” is in fact typically a collection of computers running multiple different software components (which we will meet in due time) that together comprise the functionality of the actual site. The way these components are distributed over one or many computers depends on the type of hosting environment and the number of users the app must serve. In any case, “the server” appears as a single logical entity to the client, which can remain blissfully unaware of the server’s deployment topology. Indeed, you will deploy on your own computer a “mini-server” with just enough functionality to let one user at a time (you, the developer) interact with your SaaS app during development and testing.

Distinguishing clients from servers allows each type of program to be highly specialized to its task: the client can have a responsive and appealing user interface, while the server concentrates on efficiently serving many clients simultaneously. Client-server is therefore our first example of a **design pattern**—a reusable structure, behavior, strategy, or technique that captures a proven solution to a collection of similar problems by *separating the things that change from those that stay the same*. In the case of client-server architectures, what stays the same is the separation of concerns between the client and the server, despite changes across implementations of clients and servers.

Of course, client-server isn’t the only architectural pattern found in Internet-based services. In the **peer-to-peer architecture**, used in BitTorrent, every participant is both a client and a server—anyone can ask anyone else for information. In such a system where a single program must behave as both client and server, it’s harder to specialize the program to do either job really well. But in the early days of computing, client-server architectures made particularly good sense because client hardware needed to be less expensive than server hardware, so that one could deploy large numbers of clients served by one or a few very expensive servers. Today, with falling hardware costs leading to powerful smartphones and Web browsers that support animation and 3D effects, a better characterization might be that clients and servers are comparably complex but continue to be specialized for their very different roles. Indeed, we will see those distinct roles reflected in the design patterns that appear in client frameworks (Angular, React, and so on) vs. those that appear in server frameworks (Rails, Django, Node, and so on). Even terms such as “client push” reflect the built-in assumption that clients are distinct from servers.

Although client-server systems long predate the emergence of SaaS, the Web, or even the

Year	System	Client	Server	Protocol(s)
1960	Sabre, airline reservations system for American Airlines	Custom electromechanical terminals installed at travel agencies	Two IBM 7090 mainframes	Custom FM -based protocol over leased telephone lines
1971	FTP (File Transfer Protocol), which allowed clients to download files from servers	Originally, command-line client <code>ftp</code> ; today, command-line clients (<code>cURL</code> , <code>NcFTP</code> , <code>WinSCP</code>), GUI apps (<code>Cyberduck</code> , <code>Fetch</code>), and all Web browsers	Various server software packages, including Unix <code>ftpd</code> , <code>FileZilla</code> , <code>Vsftpd</code>	ASCII-based FTP protocol over TCP/IP
1983	Novell NetWare, which allowed PCs running the CP/M or MS-DOS operating systems to share files on a server	Custom client software compatible with MS-DOS	Custom Novell file server appliance based on Motorola 68000 microprocessor	Custom protocols over custom PC-compatible network interface
1984	POP (Post Office Protocol), which allowed separation of email clients from servers	Various PC apps, including Eudora, Thunderbird, Apple Mail, Microsoft Outlook, Elm, Pine, Eureka	Various server software packages, including Apache James, Nginx, Eudora, Qpopper	ASCII-based POP protocol over TCP/IP; largely superseded by IMAP
1990	World Wide Web	Various PC apps, including NCSA Mosaic, Netscape Navigator, Microsoft Internet Explorer, Mozilla Firefox, Google Chrome	Various server software packages, including Apache Httpd, Microsoft Internet Information Server, Nginx	ASCII-based HyperText Transfer Protocol (HTTP) over TCP/IP

Figure 3.1: The Web inherits a long and rich history of computer-based client-server systems. While all of these examples arguably reflect the idea of software as a service, the term as used today refers to client-server systems built using HTTP and other Web standards and protocols.

Internet, because of the Web’s ubiquity we will use the term “SaaS” (software as a service) to mean “client-server systems built to operate using the open standards of the World Wide Web,” that is, in which Web services are accessed using the protocols and data formats described in this chapter, with Web sites accessed via browsers or via mobile apps being the most common examples.

Summary:

- SaaS Web apps are examples of the ***client-server architectural pattern***, in which client software is typically specialized for interacting with the user and sending requests to the server on the user’s behalf, and the server software is specialized for handling large volumes of such requests.
- The Web’s heritage as a fundamentally client-server architecture is ubiquitous throughout the software stacks, protocols, and terminology we will encounter throughout the rest of this book.
- Because Web apps use open standards that anyone can implement royalty-free, in contrast to proprietary standards used by older client-server apps, the Web browser has become the “universal client.”
- An alternative to client-server is peer-to-peer, in which all entities act as both clients and servers. While arguably more flexible, this architecture makes it difficult to specialize the software to do either job really well.

Self-Check 3.1.1. *What is the primary difference between the roles of clients and servers in SaaS?*

- ◊ A SaaS client is optimized for allowing the user to interact with information, whereas a SaaS server is optimized for serving many clients simultaneously. ■

3.2 SaaS Communication Uses HTTP Routes

How do SaaS clients and SaaS servers communicate? A ***network protocol*** is a set of communication rules on which agents participating in a network agree. The fundamental protocol linking all computers on the Internet is **TCP/IP**, the venerable ***Transmission Control Protocol/Internet Protocol***. TCP/IP allows a pair of communicating agents to exchange ordered sequences of bytes in both directions simultaneously (***full duplex***), analogous to a telephone conversation in which both parties can speak and listen at the same time. If a program at one end of the TCP/IP connection (say, the client) emits a string to the connection, the server will receive that exact string, and vice versa. TCP/IP doesn’t distinguish the roles of the agents on either side of the connection—it doesn’t care if one is the server and one is the client, or if they are peers in a peer-to-peer network—and it doesn’t place any restrictions on what strings are communicated. It is up to the individual programs communicating over a TCP/IP connection to determine the rules of communication. As we will see, in the case of Web browsers and servers, those rules are defined by **HTTP**, the HyperText Transfer Protocol.

How do computers contact each other in a TCP/IP-based network? Each computer is assigned an **IP address** consisting of four bytes separated by dots, such as 128.32.244.172.

Most of the time we don't use IP addresses directly—another Internet service called **Domain Name System (DNS)**, which has its own protocol also based on TCP/IP, is automatically invoked to map **hostnames** like `www.eecs.berkeley.edu` to IP addresses. When you type a site name such as `www.eecs.berkeley.edu` into your browser's address bar, the browser automatically contacts a DNS server to translate that name into an IP address, in this case 128.32.244.172. For reasons related to the design and history of IP, the IP address 127.0.0.1 and the hostname `localhost` always refer to the very computer on which the app is running. This capability lets you develop and test Web apps by connecting to the server running on your own computer: from the client's point of view such a server functions identically to one in the cloud backed by thousands of computers. That is, a TCP/IP based server program provides the same *abstraction* to the client regardless of where the server software is running and how it is distributed over one or many computers.

Because multiple TCP/IP-based programs can be running on the same computer simultaneously—for example, a Web server and an email server—the IP address isn't sufficient to distinguish them. Therefore, establishing a TCP/IP connection also requires a **port number** from 1 to 65535 to indicate which program on the server is the intended communication partner. Some program must be *listening* to that port number on the server in order to accept connections. The default ports for production Web servers are 80 for HTTP and 443 for HTTPS. The latter stands for Secure HTTP, which uses **public-key cryptography** to **encrypt** HTTP communication and protect it from eavesdroppers, as Chapter 12 describes. When you start up a development server (to test your app) in your own development environment, which port it “listens” on may be determined by the IDE you use, the framework you use (Rails uses port 3000, for example), or the manner in which you start the server.

The HTTP **protocol**—the rules for communication to make a request and receive a response—are well-circumscribed and may be summarized as follows:

1. The client initiates a TCP/IP connection to a server by specifying the IP address and port number (usually 80). If the computer at that IP address does not have an HTTP server process listening on the specified port, the client immediately experiences an error, which most browsers report as “This site can't be reached” or “Connection refused.”
2. Otherwise, if the connection succeeds, the client immediately sends an HTTP *request* describing its intention to perform some operation on a *resource*. A resource is any entity that the server app manipulates—a Web page, an image, and a form submission that creates a new user account are all examples of resources.
3. The server delivers an HTTP *response* either satisfying the client's request or reporting any errors that prevented the request from succeeding. The response may also include information in the form of an **HTTP cookie** that allows the server to correctly identify this same client on future interactions.

What does the client's HTTP request in step 2 look like? An HTTP request consists of a *route*, zero or more *headers*, and possibly a *request body*, all of which are just strings sent over the TCP/IP connection. As Figure 3.2 shows, an **HTTP route** consists of an HTTP *method*—usually, one of GET, POST, PUT, PATCH, or DELETE—plus a **URI**, or **Uniform Resource Identifier**. You are familiar with URIs as the strings usually beginning with `http://` that you type into a browser's address bar. Importantly, though, it is the *combination* of the HTTP

Octet is sometimes used in the networking literature to refer to a group of 8 bits—a legacy from the era before the IBM System/360 standardized the 8-bit byte.

65535 ($2^{16} - 1$) is the largest unsigned value that will fit in the 16 bits originally allocated to the port number in IP's original design.

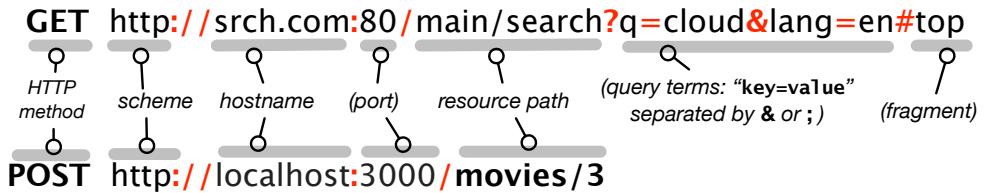


Figure 3.2: An **HTTP route** consists of an **HTTP method** plus a **URI**, and expresses the client's desire to perform some operation on the resource named by the URI. A **full URI** begins with a scheme such as `http` or `https`, which describes what protocol is to be used to access the resource, and includes the above components. Optional components are in parentheses; if the port number is omitted, it defaults to 80 for HTTP or 443 for HTTPS. A **partial URI** omits any or all of the leftmost components, in which case those components are filled in or *resolved* relative to a **base URI** determined by the specific application. Best practice is to use full URIs.

URI or URL? URIs are sometimes referred to as URLs, or Uniform Resource Locators. Despite subtle technical distinctions, for our purposes the terms can be used interchangeably. We use URI because it is more general and matches the terminology used by most libraries.

method and URI that defines a route: the same URI with different HTTP methods can have different meanings to a SaaS app. We will have much more to say about the semantics of routes later in the chapter, but in general, GET typically means “deliver a copy of the resource to the client without modifying the resource or causing any side effects,” whereas POST, PUT, PATCH, DELETE are typically used to perform an operation that creates, modifies, or deletes a resource. When you visit a URI by typing it into a browser’s address bar, your browser performs a GET to that URI; when you submit a fill-in form, the browser may perform either a GET or a POST to a specified URI, depending on how the page is authored. For historical reasons, most browsers don’t generate PUT, PATCH, or DELETE directly; we will return to their use shortly.

You’ll get hands-on practice with HTTP in an upcoming CHIPS exercise, but this background will serve to orient you in advance. HTTP’s simplicity derives from two characteristics. First, HTTP is a **request-response** protocol: every HTTP interaction begins with the client making a request, to which the server delivers a response (unless the server crashes or the network becomes unavailable, of course!) The HTTP request must include the route and HTTP protocol version, and usually also includes some request headers that provide information about the client. The HTTP protocol version tells the server which HTTP features the client can understand, so that (for example) servers can avoid using newer HTTP features if the client only understands an older protocol version. The server reply must include the HTTP version and 3-digit status code indicating the result of the requested operation; any text following the status code on the first response line is optional and ignored, but is often used to provide a human-readable version of the status. The response also includes headers describing the rest of the response data, followed by a blank line and then the response payload itself.

Second, HTTP is a **stateless protocol**: every HTTP request is independent of and unrelated to all previous requests. If this is so, how can a web site keep track of information such as whether you have logged in? HTTP provides a mechanism called **cookies** for this purpose. The first time a client makes a request from a particular server, the server can include in the response a `Set-Cookie`: header, containing a chunk of information that the server can use to identify this client on future HTTP requests. *It is the client’s responsibility* to store that information and pass it back via a `Cookie`: header on every subsequent request to that same server. Analogously to a coat-check token, a cookie should be both tamper-evident and opaque to (not interpretable by) the client, but if presented later to the server, is sufficient to identify the client, thereby enabling the concept of a continuing **session** between the server

and that client. Stateless protocols therefore simplify server design at the expense of more complex application design, but happily, successful frameworks such as Rails shield you from much of this complexity.

Summary

- Web browsers and servers communicate using the **HyperText Transfer Protocol**. HTTP relies on **TCP/IP** (Transmission Control Protocol/Internet Protocol) to reliably exchange an ordered sequence of bytes.
- Each computer connected to a TCP/IP network has an **IP address** such as 128.32.244.172, although the **Domain Name System** (DNS) allows the use of human-friendly names instead. The special name localhost refers to the local computer and resolves to the special IP address 127.0.0.1.
- Each application running on a particular computer must “listen” on a distinct **TCP port**, numbered from 1 to 65535 ($2^{16} - 1$). Port 80 is used by HTTP (Web) servers.
- A **Uniform Resource Identifier** (URI) names a resource available on the Internet. The interpretation of the resource name varies from application to application.
- An HTTP route consists of *both* an HTTP method (such as GET or POST) and a URI. The same URI with different methods results in different routes, which may or may not behave the same way in a particular SaaS app.
- HTTP is a stateless protocol in that every request is independent of every other request, even from the same user. **HTTP cookies** allow the association of HTTP requests from the same user. It’s the browser’s responsibility to accept a cookie from an HTTP server and ensure that the cookie is included with future requests sent to that server.

Cookie in hacker jargon has long meant¹ “an uninterpretable blob of data that must be presented later to identify yourself or achieve some task.”

■ Elaboration: Multi-homing, IPv6

We have drastically simplified many aspects of TCP/IP, including **multi-homed** devices and the slow phase-out of the current version of IP (IPv4) in favor of version 6 (IPv6), which uses a different format for addresses. However, since SaaS app writers rarely have to deal directly with IP addresses, these simplifications don’t materially alter our explanations.

Self-Check 3.2.1. Is DNS a client–server protocol? Why or why not?

- ◊ Yes. DNS clients only ask for lookup services (**DNS resolution**). DNS servers provide the responses, though they may consult other servers (in effect temporarily acting as clients) as part of doing so. ■

Self-Check 3.2.2. Can you make a TCP connection without specifying a port number, and if so, what happens?

- ◊ All TCP connections must specify a port number. However, specific types of clients (Web browsers, email readers, and so on) have the knowledge built into them of the **default** port numbers for those services, so end users of such clients rarely have to know this information.

■

Self-Check 3.2.3. *True or false: HTTP as a protocol has no concept of a “session” consisting of a sequence of related HTTP requests to the same site.*

- ◊ True. HTTP is stateless, with every request being completely independent of all other requests from the same client. Therefore a mechanism such as HTTP Cookies must be used to create the abstraction of a session. ■

Self-Check 3.2.4. *Many HTTP servers rely on using HTTP cookies to identify a client on repeated requests to the same site, for example, to track information such as whether that user has logged in. What happens if you completely disable cookies in your browser and try to visit such a site?*

- ◊ Try it and see. Use a search engine to find instructions on how to (temporarily) disable cookies entirely in your browser, and try to log in to a site where you have an account. Don’t forget to re-enable cookies when you finish your experiment. ■

3.3 CHIPS: HTTP and URIs



CHIPS 3.3: HTTP and URIs

<https://github.com/saasbook/hw-http-intro>

Construct URIs and make direct HTTP requests using command-line power tools that all SaaS developers should know. Examine and understand HTTP request and response headers, error codes, and cookies.

3.4 From Web Sites to Microservices: Service-Oriented Architecture

Nobody should start to undertake a large project. You start with a small trivial project, and you should never expect it to get large. If you do, you’ll just overdesign and generally think it is more important than it likely is at that stage. Or worse, you might be scared away by the sheer size of the work you envision. So start small, and think about the details. Don’t think about some big picture and fancy design. If it doesn’t solve some fairly immediate need, it’s almost certainly over-designed. And don’t expect people to jump in and help you. That’s not how these things work. You need to get something half-way useful first, and then others will say “hey, that almost works for me,” and they’ll get involved in the project.

—Linus Torvalds, interviewed by Preston St. Pierre in Linux Times, Oct 25, 2004.

When the Web began in 1990, HTTP servers existed primarily to serve static content (originally the text and images of scientific papers) that browsers would display. Each time the browser made another HTTP request, the server would deliver a new web page for the browser to show. But the emergence of SaaS around 1995 soon shifted the functionality of servers: rather than simply returning copies of static Web pages, servers would now run a program, and create HTML pages “on the fly” that implemented that program’s user interface. However, it was still the case that every new HTTP request resulted in the browser loading and displaying a new page. The next evolutionary step was the appearance of **AJAX**, or **Asynchronous JavaScript And XML**. If a browser supported AJAX, pages could include code written in the JavaScript language, and that code could make subsequent HTTP requests to the

server *without* causing a page reload. In response to those requests, the server would return not an HTML page, but a data structure in either XML or JSON format, both of which we'll meet shortly. The JavaScript code running in the browser would use that data to determine how to change the appearance or behavior of the displayed page, all without causing a page reload.

AJAX marked a turning point in the relationship between a Web site and a client: instead of receiving HTML pages and functioning primarily as a display engine, the client was essentially calling a function on the remote server and expecting to get some data back, just as if the client was calling a library function. This shift in perspective invited a new way of looking at the Web: as a set of independent services that could be composed to produce larger sites—a so-called ***Service Oriented Architecture (SOA)***.

Internet Explorer 5 was the first browser to support AJAX, in 1999. Google Maps, launched in 2005, was a dramatic demonstration of using AJAX to build truly interactive Web apps.

SOA had long suffered from lack of clarity and direction.... SOA could in fact die—not due to a lack of substance or potential, but simply due to a seemingly endless proliferation of misinformation and confusion.

—Thomas Erl, *About the SOA Manifesto*, 2010

SOA as an architectural pattern may have been a new perspective on structuring the Web, but it was certainly *not* a new idea. Despite initial skepticism about whether SOA was more than just a marketing “buzzword,” one very prominent company was internally (and, at the time, silently) making SOA quite concrete. The e-commerce giant Amazon.com launched its retail site in 1995 powered by a ***monolithic application***—a single large application that handled all aspects of the site. According to the blog of former Amazonian Steve Yegge², in 2002 the CEO and founder of Amazon mandated a change to what we would today call SOA. Yegge claims that Jeff Bezos broadcast an email to all employees along the following lines (we are paraphrasing the main points of Yegge’s description for conciseness):

All teams responsible for different subsystems of Amazon.com will henceforth expose their subsystem’s data and functionality through service interfaces only. No subsystem is to be allowed direct access to the data “owned” by another subsystem; the only access will be through an interface that exposes specific operations on the data. Furthermore, every such interface must be designed so that someday it can be exposed to outside developers, not just used within Amazon.com itself.

In this decree, Bezos captures the critical distinction of SOA: *the only way one service can name or access another service’s data is to request specific operations on that data through an external interface that provides those operations.* As an example, suppose we wanted to create a simple bookstore service where users can post reviews of books they’ve bought and maintain a profile of their reading interests. We’d need three subsystems: book reviews, user profiles, and buying. The left side of Figure 3.3 shows the silo version, similar to how Amazon.com worked in 1995. Each subsystem can internally share access to data directly in different subsystems. For example, the reviews subsystem can get user profile info out of the users subsystem. The only externally visible interface is “the bookstore.”

In other words, you as an independent web developer cannot access Amazon’s User Profile service to manage users profiles for your own e-commerce site; it’s for Amazon’s internal use only. In contrast, the right side of Figure 3.3 shows the SOA version of the bookstore service, in which all subsystems are separate and independent. Even though all are inside the

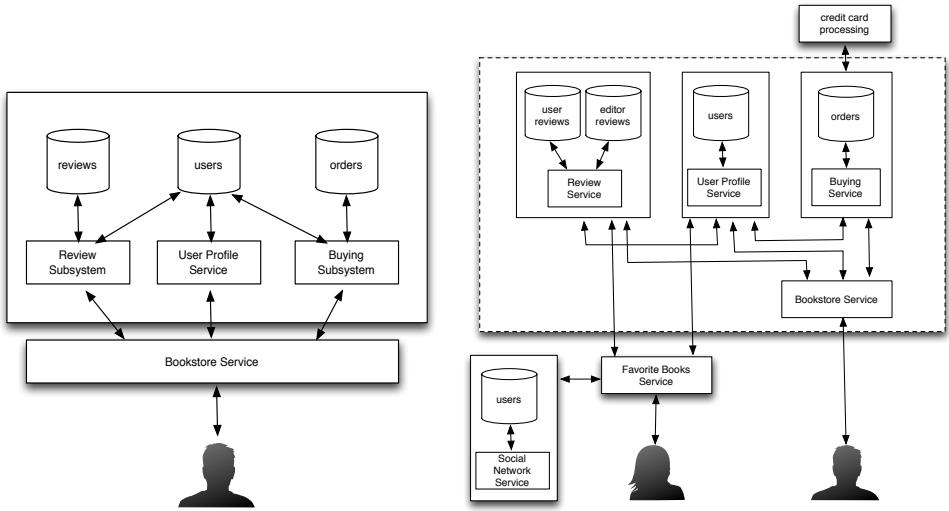


Figure 3.3: Left: Silo version of a fictitious bookstore service, with all subsystems behind a single API. Right: SOA version of a fictitious bookstore service, where all three subsystems are independent and available via APIs.

“boundary” of the bookstore, which is shown as a dotted rectangle, the subsystems interact with each other *as if* they were separate. For example, if the reviews subsystem wants to update a user’s profile to indicate that the user has written a review, the reviews subsystem can’t reach directly into the users database. Instead, it has to ask the users **service** to update the user’s information, via whatever interface is provided for that purpose. If no such operation is provided, the reviews team must negotiate with the user database team to get them to expose the necessary operation.

A **microservice**, in contrast, is a standalone service that performs just one type of task, *and* is specifically designed to be accessible from and incorporated into the functionality of any other outside service (perhaps for a fee). Though there is no hard-and-fast criterion distinguishing the scope of a microservice from that of a service, the prefix *micro* is intended to promote an “extreme SOA” design stance in which each microservice is responsible for a single narrowly-defined function. For example, Google Maps feels like a single app when you use it in a browser, but the different “clusters” of related functionality used by that app—drawing the map, computing driving directions, **geocoding** addresses, and so on—appear as distinct microservices to the underlying JavaScript code that implements the app. A rough rule of thumb is that the scope of a microservice should correspond to a set of closely related operations on a very tightly integrated set of resources. Notwithstanding, since there is no hard distinction between a service and a microservice, we will use the term *service* throughout. We will adopt the following rough definition, found in Nadareishvili et al. 2016: “A (micro)service is an independently deployable component of bounded scope that supports interoperability through message-based communication.” Based on this definition, we can see that the monolithic bookstore fails to be “micro” not just because of its size, but because its components are not independently deployable since they share access to databases.

As Figure 3.4 shows, there are both pros and cons to preferring a service-oriented architecture over a monolithic one. In short, we can think of microservices as a manifestation

Pro	Con
Reusability: Others can recombine existing services to create new apps, as in Figure 3.3, and each microservice can be implemented using the most appropriate language or framework, since its implementation is completely hidden behind its API.	Performance: each invocation of a service involves the higher cost of wading through the deeper software stack of a network interface, so there is a possible performance penalty to SOA.
Easier testing: a microservice does only one thing, so testing each microservice is easier.	Managing partial failure: a monolithic system is either working or not, but in an SOA, some services may fail while others are working, making dependability more challenging.
More Agile-friendly: Chapter 1 reveals that Agile works best with small-to-medium projects and teams. SOA allows large services to be created by composing smaller ones, each of which can be built and operated by a small Agile team.	More development work: you must design and implement an interface for each service component, rather than a single interface for the entire site. Fortunately, an approach called REST, which we describe next, simplifies this task.
“You build it, you run it” (as Amazon Web Services CTO Werner Vogels has said): the same tightly-knit team is responsible for developing, testing, and operating the microservice, allowing the microservice to be improved more quickly in response to customer requests.	Developers must learn about operations, and vice versa; hence “dev/ops.” This is a reality of modern SaaS to which we return in Chapter 12.

Figure 3.4: Each benefit of SOA in the left-hand column comes at a cost, as shown in the right-hand column. Nonetheless, in practice the benefits of SOA seem to outweigh the costs, if SOA is done well.

of extreme programming (XP) as applied to service-oriented architecture: If it's good for services to be independently evolvable, make each one as compact as possible to maximize that independence. Microservices may also arise by design or by splitting up an existing large service, as occurred with Twitter. Around 2013, their monolithic Rails application, which they humorously called "the MonoRail" internally, was split up into a large number of microservices, most of which were written in Java, Scala, or Clojure.

Summary of Service-Oriented Architecture and Microservices

- Although the term was nearly lost in a sea of confusion, ***Service Oriented Architecture (SOA)*** just means an approach to software development in which subsystems can only access each others' data via external interfaces.
- From 1990 to 2010, the Web underwent a transformation from serving static content (Web 1.0) to serving dynamically-generated user interfaces (SaaS) to allowing ongoing interactions after the initial page load (AJAX) to architecting large apps by composing independent services (SOA).
- Although there is no bright line separating a service from a microservice, a microservice should perform a related set of operations on a well-circumscribed set of resources, should be independently deployed and operated (usually by the same team that builds it), and should be designed to be readily incorporated with other external services.

■ *Elaboration: Microservices and the Unix philosophy*

The influence of the Unix operating system is pervasive throughout software engineering, due in part to its careful design choices. In particular, the "Unix philosophy" promotes the building of simple components that perform just one task and are easily composable, in that the output from any component should constitute legal input to any other. Microservices can be seen as the triumph of the Unix philosophy in the world of SaaS: a microservice should do just one thing very well, and make the fewest possible assumptions about how it will be integrated into a larger SOA.

Self-Check 3.4.1. *Another take on SOA is that it is just a common sense approach to improving programmer productivity. Which productivity mechanism does SOA best exemplify: Clarity via conciseness, Synthesis, Reuse, or Automation and Tools?*

◊ Reuse! The purpose of making internal interfaces visible is so that programmers can stand on the shoulders of others. ■

3.5 RESTful APIs: Everything is a Resource

An API that isn't comprehensible isn't usable.

—James Gosling, inventor of Java

SaaS microservices using HTTP are just the latest manifestation of ***remote procedure call***, an idea with a long history (Birrell and Nelson 1984).

The premise of service-oriented architecture is that each service provides a well-defined set of operations on one or a few related types of resources—analogous to a library for a programming language. In other words, clients need a way to name the server function to

	Python program (caller) calls a method in a Python package or library (callee)	SaaS client (caller) invokes SaaS service (callee)
1. How does the caller identify the callee?	Same computer and same process as caller; <code>import</code> makes a particular named library available to the caller, as in <code>import numpy</code>	An endpoint is a logical address that clients contact in order to use the service. In our case, it usually takes the form of a “base URI,” that is, a URI prefix (including the microservice’s hostname and port number if necessary) that is common to all API calls made through that endpoint
2. Which operation is called?	Method named in code, e.g. <code>numpy.array(...)</code>	Operation named in path portion of URI
3. How does the caller pass required and optional parameters to the callee?	Passed as arguments to method call, e.g. <code>numpy.array([1,2,3])</code>	May be passed as part of URI path, as key-value pairs in URI query string, or as a data payload in JSON or XML format in the request body
4. How does the caller receive a return value?	Returned from method call and usually assigned to a variable, e.g. <code>n=numpy.array([1,2,3])</code>	Service typically returns a data structure in JSON or XML format
5. How does the callee signal an error?	Callee may return a “sentinel” error value (such as <code>None</code> in Python), or raise an exception	Service returns an appropriate HTTP status code to indicate error type, and usually provides an error message as part of the returned data structure

Figure 3.5: APIs describe any caller–callee contract, whether calling a service in an SOA or a Python library function. API documentation specifies which operations are available, how URLs should be constructed to invoke those operations, how errors are reported, and what other operational requirements must be met (for example, limiting the number of calls made per day to a microservice, or including an account identifier or password with each API call).

be called, pass arguments to it, consume return values, detect and handle server exceptions (errors in execution), and so on, just as when an application calls a library function, but all subject to the constraints of using HTTP for communication. The term **API**, or Application Programming Interface, refers to the “contract” between a caller and callee, whether these are a program calling a library function or a SaaS client invoking a service on a SaaS server, as Figure 3.5 shows.

Unfortunately, rows 2 and 3 in the figure are problematic, because HTTP does not prescribe a way to “name a remote function” or “pass parameters” since those tasks were never part of its original design. In particular, the HTTP and URI specifications offer no conventions regarding the *semantics* (implied meaning) of how URIs are constructed or how these tasks should occur. There was early and widespread recognition that standardizing the conventions for such communication would enable the creation of an ecosystem in which *any* client, not just a Web browser, could make use of a given server in different ways.

In most of the microservices world, these conventions are articulated by REST, short for REpresentational State Transfer. In 2000, computer scientist Roy Fielding proposed REST in his Ph.D. dissertation as a way of mapping requests to actions that is particularly well suited to a service-oriented architecture. REST is not a standard, but a design stance regarding how a service should be constructed, and by extension, what its API should look like. Fielding’s idea was to represent the various entities manipulated by a Web app as **resources** (hence *representational*), and to construct routes so that any HTTP request would contain all the information necessary to identify both a resource and the action to be performed on it, which

might cause a change of state in one or more resources (hence *state transfer*). An API that adheres to Fielding’s guidelines is said to be RESTful, and the routes (HTTP method plus URI) defined by the API to invoke particular actions are said to be RESTful routes.

Although simple to explain, REST is an unexpectedly powerful principle for simplifying and organizing SaaS applications, because it makes the app designer think carefully about how each type of entity manipulated by the app can be represented as a resource, what operations can be done to that resource, and what conditions or assumptions must hold in order for a request for such an operation to be self-contained. For any RESTful API operation, it should be straightforward to answer the following questions:

1. What is the primary resource affected by the operation?
2. What is the operation to be done on that resource? What are the possible results? What are the possible side effects, if any?
3. What other data is necessary to complete the operation, if any, and how is it specified?

For example, consider the answers to the above questions in the case of posting a review for the movie “2001: A Space Odyssey,” a classic favorite of one of your authors:

1. The primary resource is the new review for the specific movie “2001: A Space Odyssey,” which might include (for example) a numerical rating and a few lines of text.
2. The operation is to create a new review using that information. One possible result is success, with the side effect that a new review is created. The other possible result is that creating the review fails for a variety of possible reasons (perhaps this client is not authorized to post reviews, or the database is full, or no more reviews are allowed for this movie), in which case there are no side effects.
3. Besides the review itself, the additional necessary data is some identification of which movie the review is intended to be linked to. As we will see, this identifier will likely be passed as part of the route, either as a component in the path portion of the URI or as a parameter in the query-string portion of the URI. Also, if the review app allows optionally associating a reviewer name or reviewer ID with the review, an identifier representing the reviewer may similarly be necessary.

The API documentation should describe what operations are available and how the required and optional arguments should be provided for each operation. As we’ll learn in Chapter 4, Rails and other frameworks have built-in support for easily defining RESTful routes.

In its purest form, a RESTful API defines up to five operations on a resource, captured by the acronym CRUDI: Create a new instance of a resource, Read (retrieve) a copy of a resource, Update (make changes to) a resource, Delete a resource, and list an Index of all available resources of a given type, possibly filtered by particular criteria. Many APIs go further and define additional operations specific to the resource types used by that service. Nearly always, each resource is given a unique ID—usually a number—that will serve as the “permanent handle” to that resource and is never reused even if the resource is deleted. A common usage pattern for RESTful APIs is to return a list of resources (with their IDs) corresponding to a search operation; the client can then retrieve the desired resources one by

one via their IDs. Consistent with Section 3.2, RESTful routes whose actions have no side effects typically use GET, while those with side effects use POST, PUT, or PATCH.

We can now describe concretely how RESTful service APIs address the requirements of rows 1–3 of Figure 3.5. (You’re strongly encouraged to consult the API documentation at <https://developers.themoviedb.org/3/> as you read the rest of this section.) Recall Figure 3.2, which shows the various parts of a URI. RESTful APIs observe several conventions for how those parts of the URI are constructed when making an API call:

- The hostname component of the URI tells us which server provides the service: <https://api.themoviedb.org>.
- In addition, most servers providing RESTful APIs specify a **base URI**, or common URI prefix that should be prepended to all API calls. You can see from the API documentation that all of the URIs begin with <https://api.themoviedb.org/4/>; this base URI or hostname-plus-prefix is sometimes referred to as the API **endpoint**, or logical address that clients contact in order to use the service.
- In this case, the API documentation tells us that the component 3 of the endpoint name refers to the version number of the API. Making the version number part of the URI allows the API to evolve while preserving compatibility with older clients. You may also see variants such as <https://themoviedb.org/api/v4/>, <https://api.themoviedb.org/v4/>, and others.
- The URI path components *following* the prefix specify the operation to be performed and the resource on which to perform it. API documentation frequently uses a colon (:) or curly braces to indicate a URI component corresponding to a resource ID, so the API documentation might state that the route GET /movie/{movie_id} or GET /movie/:movie_id requests detailed information for the specific movie whose numeric ID is substituted for {movie_id} or :movie_id in the URI.

In the next section we describe exactly how the data associated with these requests is formatted and how errors are handled—rows 4 and 5 of Figure 3.5.

A common though not universal feature of RESTful APIs is that the structure of the URI path itself reveals information about the relationships among resource types. For example, the TMDb API route GET /movie/{movie_id}/reviews, retrieves all the reviews for a particular movie—effectively, the Index operation on reviews, constrained to a particular movie. The structure of the URI suggests that the same movie can have many associated reviews (a so-called “has-many” relationship, which we’ll meet in Chapter 5). Similarly, a hypothetical route such as GET /movies/5/reviews/22, to request the content of review ID 22 associated with movie ID 5, might seem redundant since the review ID must be unique anyway; but the route structure reveals an otherwise non-obvious relationship. Not all RESTful sites follow this practice, though: the TMDb route for a particular review is simply GET /reviews/{review_id}, and some TMDb routes use multiple path components (terms separated by slashes) to express different sub-operations on a resource rather than relationships among resource types.

You can verify by reading the TMDb API docs that the call for retrieving all reviews for a movie actually returns some of the content of each review. In a “purer” RESTful API, such an Index call might just return a list of review IDs, and the client could then retrieve the contents of individual reviews by their review ID. It’s possible that the TMDb API sought to

Singular or plural? Some styles of REST API, including that used by Rails, use plural resource names when there can be more than one resource of that type, as in GET /movies/35, and singular names when there’s exactly one such resource, as in GET /homepage.

	Non-RESTful site URI	RESTful site URI
1. Login to site	POST /login/dave	POST /login/dave
2. Welcome page	GET /welcome	GET /user/301/welcome
3. Add item ID 427 to cart	POST /add/427	POST /user/301/add/427
4. View cart	GET /cart	GET /user/301/cart
5. Checkout	POST /checkout	POST /user/301/checkout

Figure 3.6: Non-RESTful requests and routes are those that depend on the results of previous requests but don't make those dependencies visible or explicit as part of the current request. In a Service-Oriented Architecture, a client of the RESTful site could immediately request to view the cart (line 6), but a client of the non-RESTful site would first have to perform lines 3–5 to set up the implicit information on which line 6 depends.

make things more efficient so that enough information is returned for each review that the client could decide which reviews were worth getting more details on. But if a movie had thousands of reviews, returning review data rather than just review IDs for all those reviews might become unwieldy.

RESTfulness may seem an obvious design choice, but until Fielding crisply characterized the REST philosophy and began promulgating it, many Web apps were designed non-RESTfully. Figure 3.6 shows how a hypothetical non-RESTful e-commerce site might implement the functionality of allowing a user to login, adding a specific item to his shopping cart, and proceeding to checkout. For the hypothetical non-RESTful site, every step after the login (step 1) relies on implicit information: step 3 assumes the site “remembers” who the currently-logged-in user is to show them the welcome page, and steps 4–5 assume the site “remembers” who has been adding items to their cart for checkout. In contrast, each URI for the RESTful site contains enough information to satisfy the request without relying on such implicit information: after Dave logs in, the fact that his user ID is 301 is present in every request, and his cart is identified explicitly by his user ID rather than implicitly based on the notion of a currently-logged-in user.

Summary

- To treat one or more SaaS apps as “services” that can accept remote procedure calls on behalf of a client, we need to be able to identify the service, identify which operation (which function) is to be called, pass data to and receive data from the service, and handle errors.
- While there is no enforced standard regarding the mapping between HTTP routes and API operations on a service, REST (REpresentational State Transfer) has emerged as a simple, consistent way to do so that works well with Web technologies.
- The key idea of REST is to represent each type of thing managed by the service as a resource, and provide a limited set of operations (typically Create, Read, Update, Delete, and Index) that can be performed on a resource. The corresponding RESTful request includes all the information necessary to complete the specified action on that resource.

■ Elaboration: Command-Query Separation

Noted software engineering researcher Bertrand Meyer has long advocated (Meyer 1997) **command-query separation**: a given method or operation should either be data-altering (command) or read-only (query) but not both. REST respects this principle by not only separating these operations but, in the “pure REST” formulation, giving each operation only a single responsibility: a given API call either returns data—an individual resource, or a possibly filtered collection of resources of one particular type—or it creates, updates, or deletes a single resource of a particular type.

Self-Check 3.5.1. Which of these routes for updating the information of movie ID 35 follow good HTTP and REST practices: (a) POST /movie/35, (b) POST /movies/35, (c) PUT /movie/35, (d) PUT /movies/35, (e) GET /movie/35, (f) GET /movies/35,

- ◊ All except (e) and (f) follow defensible practices. Whether to use singular or plural is a matter of style and convention, but GET should not be used for routes whose actions have side effects. ■

3.6 RESTful URIs, API Calls, and JSON

In considering how to treat a collection of SOA servers as a fabric for programming, we’ve addressed rows 1–3 of Figure 3.5. We next describe how data is passed to or received from such services, and some operational considerations such as authorization (is the client allowed to make this API call on this resource?) and how errors are handled.

At the highest level, there are three ways to pass parameters *from* an HTTP client *to* a service: in the URI, in the request body (for POST or PUT requests), and rarely, as the value of an HTTP header.

When the number of parameters is small, and in particular when the parameters are simple types such as strings or numbers, they can often be passed as parameters embedded in the URI, as Figure 3.2 showed: param1=value1¶m2=value2&...¶mN=valueN. This situation is typical for GET requests, where we’re usually asking for data based on an ID and perhaps some optional parameters. For example, verify using the TMDb API documentation that the route GET /search/movies?query=Batman+Returns will search TMDb for a movie whose title matches the query string “Batman Returns”.

When the data to be passed is more complex, or when the API operation involves a state-changing HTTP method such as POST or PUT, the data is sent as part of the request body, as browsers do when submitting the values entered on a fill-in form. (Recall that GET requests have no request body.) How is this data presented to the server? While there are many choices, there is no question that the SOA community has rapidly converged on **JSON** (pronounced “JAY-sahn”), or JavaScript Object Notation, as the common interchange format. JSON is so called because its syntax resembles, though is not identical to, the syntax of a JavaScript object literal—a set of unordered key/value pairs, like a Ruby hash, Python dict, or Java HashMap. In JSON, each key (or “slot,” as we’ll learn in Chapter 6) must be a double-quoted string, and its value may be a simple type (string, numeric, true, false, null), a linear array each of whose elements can be any of these, or another object whose slots are constrained to these same types. The JSON web site⁴ shows some simple examples, and because of JSON’s popularity as the default data format for SOA, virtually every modern language comes with libraries to both generate and parse JSON. Whitespace (spaces,

Like JavaScript itself, the JSON standard is stewarded by ECMA, the European Computer Manufacturers Association.

```
https://gist.github.com/c428409170320d32ba60934e1d29b190
1 # set endpoint for TMDb API
2 export BASE=https://api.themoviedb.org/v3
3 # set our API key for use in other calls
4 export KEY="my API key here"
5 # Search for a movie by keywords
6 curl "$BASE/search/movie?api_key=$KEY&query=Batman+Returns"
7 # For better legibility, pipe the output to json_pp:
8 curl "$BASE/search/movie?api_key=$KEY&query=Batman+Returns" | json_pp
9 # Start a new guest session
10 curl "$BASE/authentication/guest_session/new" | json_pp
11 # capture the guest session ID from Curl's output:
12 export SESSION=e91f07cca8166b7b1e707d8a826e8a38
13 # Create a file containing the JSON object for rating a movie:
14 echo '{ "rating": 6.5 }' > myrating.json
15 # Use Curl to POST a movie rating request using the file's contents:
16 curl -X POST -H "Content-Type: application/json" -d @myrating.json \
    "$BASE/movie/364/rating?api_key=$KEY&guest_session_id=$SESSION"
```

Figure 3.7: Try the lines of this shell script one at a time, replacing the value of KEY with your TMDb API key. After line 8 you'll have to visually parse out the correct movie ID from the JSON response, and after line 10 the correct guest session ID to use in line 12. If your system doesn't have json_pp installed, you can omit it.

tabs, newlines) is optional in JSON, and most servers return whitespace-free JSON. Unix command-line tools such as `json_pp` and browser extensions like JSONView restore spacing and indentation to make JSON more readable. Note that calls that require sending JSON data may *also* allow (or require) sending some parameter values encoded in the URI; you must check the API documentation for details.



HTTP headers are sometimes used to pass very specialized types of parameters. For example, some APIs require you to add the HTTP header `Content-Type: application/json` to a request that will be accompanied by a JSON payload, while others don't. Finally, nearly all APIs require authorization—the client must prove it has the right to make each API call. While authorization schemes vary, the most common is to include a client-specific API key with each request. API keys are usually requested manually and may be free or paid (TMDb's are free), and the service may impose limits such as the number of calls made per day. Depending on the API, the key may be sent as an argument in the URI (as with TMDb), as the value of an `Authorization:` header, or either.

Chapter 12 explains why HTTPS makes it OK to transmit what amounts to a password as part of a Web request.

Scopes let a caller specify, at token request time, what kinds of operations it wants to do using the API. The GitHub API⁵ supports a variety of scopes; simpler APIs such as TMDb's usually don't support scopes.

Putting this all together, Figure 3.7 shows the use of the `curl` tool to do a sequence of RESTful API requests—all but the last are GETs—exercising the TMDb API. Note in particular that the API key is a required URI parameter for every request, and verify against the API documentation the correct format for the object in line 14 representing the desired rating you wish to submit for a movie.

What if an error occurs? Recall from Section 3.2 that every HTTP response begins with a 3-digit status code; these are catalogued and maintained⁶ by the World Wide Web Consortium. Services use status codes to indicate various types of errors:

- 2xx codes indicate success. For example, code 200 (“OK”) would be the usual success status for a GET, whereas code 201 (“Created”) would be more typical for a POST that creates a new resource.
- 3xx codes indicate the client must take further action to complete the request—that is, a redirect. Perhaps the requested resource has moved to a different URI, which would be specified in the response body.

- 4xx codes indicate that the service encountered an application error processing the request. 400 means the request was malformed, but other codes for well-formed requests include 401 (Unauthorized), 402 (Payment required), and others.
- 5xx codes indicate a problem with the service infrastructure itself—an error that prevented the remote call from even completing, such as the server encountering an internal error so severe that it is too broken to even explain what went wrong.

In case of an error (any status other than 2xx), the response body usually contains a message explaining what went wrong. Depending on the API, the response body will consist of either just this string, or more commonly, a JSON object with a single string-valued slot named `message` or `error` or something similar.

Summary

- One way to apply the RESTful design stance to HTTP routes is to use the HTTP method (GET, POST, and so on) and the URI path to encode the resource and operation to be performed. For GET requests, optional arguments can be encoded in the URI itself (`?param1=value1&...¶mN=valueN`). For POST or PUT requests, typically used for form submission, both the form's field values and additional optional arguments can be transmitted as part of the request body.
- Requests to a service may require that the client present some credentials to prove it is authorized to use the service. Many schemes exist, but among the simplest is HTTP Basic Authentication, in which a username and password (possibly collected from the user by the browser's UI) are embedded in the HTTP headers. The use of Secure HTTP (HTTPS), which we describe in Chapter 12, ensures that eavesdroppers cannot read this sensitive data.
- JSON (JavaScript Object Notation), based on JavaScript language syntax, has become the most popular data format for sending data to and receiving data from SaaS services.
- No one “legislated” that REST would defeat dozens of competing proposals to become the preferred way to design services. Instead, REST became widely adopted because it was simple to understand and implement, well matched to the underlying Web protocols, and unencumbered by intellectual property restrictions.

■ ***Elaboration: Why did REST win?***

The road to today’s microservices is littered with acronyms of proposed conventions and standards for interoperation that never fully caught on: ***SOAP***, ***WSDL***, ***UDDI***, ***XML-RPC***, ***DCOM***, ***Jini***, and ***CORBA***, to name just a few. Since no single entity controls the Internet and gets to “pick the winner,” a winner usually emerges by rough consensus for practical reasons: it is easy for developers to understand and use (especially to get simple common cases working quickly), it is well matched to the underlying technology stack (in this case the Web’s protocols and standards, especially HTTP), and it is not subject to a costly or restrictive developer license. REST meets these criteria—it is simple enough to be described in this one textbook section—but it achieves the goal of being a good match for HTTP by cheating: it is a *retrospective codification* of the practices that were *observed* to work well as the Web went through its growing pains, such as an emphasis on stateless design, a scheme for constructing URIs that plays nicely with Web caching (Chapter 12), no reliance on an implicit notion of a session, and so on. Most of the competing protocols ignored one or more of these lessons and so failed to hit the “sweet spot.”

Self-Check 3.6.1. *You try an API call on TMDb and the status code of the response is 400. Assuming TMDb adheres to the official W3C semantics of the status codes, which of the following could be the reason for the error: (a) your request was malformed so could not be attempted; (b) you forgot to include your API key; (c) your request and API key were well-formed, but you are attempting an operation that you’re not authorized to do.*

- ◊ (a) is most likely. 401 Unauthorized would be more likely for the other two cases. ■

3.7 CHIPS: Create and Deploy a Simple SaaS App



CHIPS 3.7: Create and Deploy a Simple SaaS app

<https://github.com/saasbook/hw-sinatra-saas-hangperson>

Create and deploy a SaaS app that plays the “Hangman” game using Ruby and the simple Sinatra SaaS framework. Design how game actions map to HTTP routes, how game state is represented, how cookies are used to manage that state, and how to detect and prevent cheating (that is, realizing that you cannot trust any HTTP client).

3.8 Fallacies and Pitfalls



Fallacy: Splitting a large “monolithic” service into many microservices decreases complexity.

The complexity of a system’s application logic does not disappear from splitting it or designing it to be service-oriented; the complexity is simply distributed among the microservices, and in particular, how the interaction among those microservices is managed. The two main structural patterns for coordination among microservices are orchestration (the composition layer has a higher level of complexity and holds more of the application’s logic) and choreography (the services interact among themselves without a separate composition

layer). Finally, splitting a large service into microservices requires thinking about module boundaries just as one would in designing a large service in the first place.



Fallacy: Publishing my API makes my service RESTful (or makes it a microservice, or SOA-friendly, and so on).

A published API just means that external calls are allowed; the API's understandability and usability will determine whether the API or service is widely adopted, since the API defines the boundary negotiation for what the service will do (expected output) and how the caller will access it (expected input). REST is not the only good way to design an API, but there are certainly many bad ways to design an API, and carefully following REST makes it less likely you'll accidentally choose one of those ways.



Fallacy: I haven't published an API, so clients cannot call my app.

Every publicly-accessible website already has a de facto HTTP API, because URIs can be constructed to interact with the site. Of course, such an "accidental" API will rarely adhere to good design practices; a client wishing to use it might have to (for example) pick apart a returned HTML page to extract the information it wants, a process sometimes called HTML scraping. Nonetheless, if your app is publicly available, it has an API whether you intended it or not. If you do intend for your app to be used programmatically as a service or microservice, you should design and expose an API according to the guidelines in this chapter.



Pitfall: Thinking in terms of URIs, user actions, and views instead of thinking in terms of resources.

Consider a sequence of steps for a hypothetical e-commerce site: in step 1, a user visits a product page; in step 2, they add a product to a shopping cart; perhaps they repeat steps 1 and 2 to add multiple products; and in step 3, they pay for the product(s). If you design the business logic for such an app from a "Web-page-centric" point of view, you might be tempted to simply keep track of which step the user is on (say, as part of the session or by including the step number as a parameter in a URL) and include logic that dispatches to the appropriate internal action for each step. But such a design approach doesn't force you to think about important questions such as: If the user leaves the site and returns later, how can we ensure their cart contents haven't changed? If the user abandons the order without paying, at what point do we decide to delete it? If the payment step fails, can we easily direct the user to retry only that step without going through all the previous steps again? In contrast, a RESTful API design approach would begin by asking: What kind of resource is an order, and what operations are available on it? What kind of resource is a product, and what operations are available on it? What can go wrong during each type of operation? How is each kind of resource stored on the server, and under what conditions can it be deleted? How does the client refer to a particular resource that was created earlier, even if the user has been away from the keyboard for a long time? Thoughtful answers to such questions result in a clean service design that is suitable for use both as a standalone (micro)service and as the back end of a browser-based experience.



Pitfall: Poor design of API resulting from misunderstanding of the domain or of the customers' use cases.

We noted that the TMDb API call for "get reviews associated with this movie" actually returns the reviews' contents, not just their IDs. This design makes sense if most requests of

this type are likely to inspect the content of most of the reviews. But if the more common use case was (for example) to allow an end user to display details of reviews with particular characteristics such as numerical rating, a leaner API might allow specifying those options to constrain the result.

In general, insufficient understanding of how customers will use your API may lead to inefficient resource representation internally, but as we will see, the good news is that the internal representation of resources can be changed later as long as the details of that representation haven't leaked into the API.

3.9 Concluding Remarks: Continuity From CGI to SOA

Because an early use of the Web was to serve static files stored in a file system, early HTTP servers such as Apache could be configured to expose a subdirectory of the file system as a browseable tree of files, so early URIs typically mimicked the hierarchical structure of a file system. For example, the URI `http://www.cs.berkeley.edu/reports/1997/daedalus.ps` very likely referred to the actual file `daedalus.ps` located in subdirectory `reports/1997/` somewhere on the computer whose hostname is `www.cs.berkeley.edu`.

In 1993 the **Common Gateway Interface** or CGI protocol marked the emergence of SaaS. A CGI-capable Web server could interpret certain URIs not as the name of a local file, but as a directive to run a program and send its output back to the client. While no conventions were proposed for how to construct such URIs, a common one was to place all such “CGI programs” under a single subdirectory, often called `cgi-bin`, and use a combination of URI path components and parameters in the query string to “pass arguments” to the program to be run. The CGI program had to emit a complete well-formed HTTP response, including appropriate HTTP response headers and a content payload such as an HTML page. As SaaS became an increasingly common way to deploy Web sites, **application server** frameworks began to emerge that automatically took care of some of this “plumbing,” such as building an HTTP response or handling common HTTP errors, freeing the application writer to focus only on the content.

bin is short for **binary (executable)** almost everywhere in computing, even though later CGI programs were not binary files but scripts in languages like Perl or Python.

The rapid rise in popularity of the “microservices with RESTful APIs” model has led to a renewed focus on API design and tools to support it. Joshua Bloch’s article *How To Design a Good API And Why It Matters* (Bloch 2006), and the accompanying technical talk given at Google⁷, provide a good overview of how to think about API design. Bloch compares API design decisions to language design decisions, which have been debated for decades, and offers a concise operational definition of an API as “the methods of operation by which components in a system use one another.” (Another talk⁸ by the same author provides a wry and opinionated history of APIs since 1950.) Furthermore, since the API is the visible “contract” with callers of the service, formally documenting the API itself has become increasingly important, along with ensuring that as the service evolves, the API documentation stays current. The OpenAPI (formerly Swagger) tools⁹ include an API editor for designing APIs with the OpenAPI specification, a code generator to generate server and client stubs for using an API, and tools to automatically extract and publish documentation with “live” API exercisers from an OpenAPI description.

A recent alternative to purely-procedural REST APIs is **GraphQL**, which is based on describing data structures rather than procedure calls. In a RESTful API, the server decides what operations to expose and what data structures are required to invoke them. In contrast, a GraphQL client defines the data structures it needs, and the same structures are returned from

the server. The richness and complexity of GraphQL may not be worthwhile for simpler APIs, but it is an interesting emerging alternative to REST for data-intensive services.

Lastly, it is worth remembering that such APIs are really just the latest manifestation of a key factor in the Web’s success: separating the things that change from those that stay the same. TCP/IP, HTTP, and HTML have all gone through several major revisions, but all include ways to detect which version is in use, so a client can tell if it’s talking to an older server (or vice versa) and adjust its behavior accordingly. Today, APIs allow separation of interface from implementation at the level of entire services. Although dealing with multiple protocol and language versions puts an additional burden on browsers and other clients, it has led to a remarkable result: A Web page created in 2019, using a markup language based on 1960s technology, can be retrieved using network protocols developed in 1969 and displayed by a browser first created in 1992. Separating the things that change from those that stay the same is part of the path to creating long-lasting software.

Tim Berners-Lee, a computer scientist at CERN¹⁰, led the development of HTTP and HTML in 1990. All open Web standards, including these, are now stewarded by the nonprofit vendor-neutral World Wide Web Consortium (W3C)¹¹.

- A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, Feb. 1984. ISSN 0734-2071. doi: 10.1145/2080.357392. URL <http://doi.acm.org/10.1145/2080.357392>.
- J. Bloch. How to design a good api and why it matters. In *Proc. 21st ACM SIGPLAN Conference (OOPSLA)*, pages 506–507, Portland, Oregon, 2006. URL <http://portal.acm.org/citation.cfm?id=1176617.1176622>.
- B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997.
- I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. *Microservice Architecture*. O’Reilly Media, Sebastopol, CA, 2016.

Notes

¹<http://www.catb.org/~esr/jargon/html/M/magic-cookie.html>

²<https://gist.github.com/chitchcock/1281611>

³<https://developers.themoviedb.org>

⁴<https://json.org/example.html>

⁵<https://docs.github.com/en/developers/apps/scopes-for-oauth-apps>

⁶<https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

⁷<https://www.youtube.com/watch?v=heh40eB9A-c>

⁸<https://www.youtube.com/watch?v=ege-kub1qtk>

⁹<https://swagger.io/tools>

¹⁰<http://info.cern.ch>

¹¹<http://w3.org>

4

SaaS Framework: Rails as a Model–View–Controller Framework

Alan Perlis (1922–1990) was the first recipient of the Turing Award (1966), conferred for his influence on advanced programming languages and compilers. In 1958 he helped design ALGOL, which has influenced virtually every imperative programming language including C and Java. To avoid FORTRAN's syntactic and semantic problems, ALGOL was the first language described in terms of a formal grammar, the *Backus-Naur form* (named for Turing Award winner John Backus and his colleague Peter Naur).



In programming, everything we do is a special case of something more general—and often we know it too quickly.

—Alan Perlis

4.1	The Model–View–Controller (MVC) Architecture	98
4.2	Rails Models: Databases and Active Record	100
4.3	CHIPS: ActiveRecord Basics	105
4.4	Routes, Controllers, and Views	105
4.5	CHIPS: Rails Routes	110
4.6	Forms	111
4.7	CHIPS: Hangperson on Rails	116
4.8	Debugging: When Things Go Wrong	116
4.9	CHIPS: Hello Rails	120
4.10	Fallacies and Pitfalls	120
4.11	Concluding Remarks: Rails as a Service Framework	121

Prerequisites and Concepts

Like most modern SaaS frameworks, Rails captures two decades' worth of developer experience by encapsulating common operations (so SaaS app writers don't have to handle them) and by exposing proven SaaS design patterns (so SaaS app writers can easily apply them).

Prerequisites:

You should be familiar with basic operations in SQL (the Structured Query Language), such as `INSERT`, `SELECT . . . WHERE`, `UPDATE`, and `DELETE`. An excellent free resource for learning SQL basics is the Khan Academy SQL tutorial¹: the sections on SQL Basics, More Advanced SQL Queries, and Modifying Databases with SQL will suffice for this chapter.

Concepts:

- A Rails app is best viewed as a collection of RESTful resources on which the app provides an interface for performing various operations.
- Well designed software systems reflect organization at multiple levels of granularity, often based on identifiable architectural patterns. A Rails app implements the server side of the client-server pattern introduced in Chapter 3. The structure of the app itself introduces another architectural pattern called model–view–controller, or MVC.
- In the Rails implementation of MVC, models—the main data managed by the app—are stored in a relational database using the *Active Record* design pattern. Views, which allow users to see and interact with the data, use the *Template View pattern* to create HTML or JSON representations of the app's resources (models). Controllers, which mediate interaction between the views and models, follow *Representational State Transfer* (REST), in which each controller action describes a single self-contained operation on one of the app's resources.
- Rails' mechanisms can be used to build both SaaS apps designed for use from a browser and microservices adapted to work in a service-oriented architecture.
- Rails uses Ruby's features of metaprogramming and introspection to provide *convention over configuration*: if you follow certain conventions regarding the naming of classes, variables, and files, nearly all manual configuration can be avoided, making apps smaller and simpler to understand and maintain.
- Unlike debugging PaaS, debugging SaaS requires understanding the different places something could go wrong during the flow of a SaaS request, and how to surface that information to the developer.

4.1 The Model–View–Controller (MVC) Architecture

All well-written and nontrivially-sized applications reflect some macroarchitectural organization, that is, they can be thought of as ensembles of large communicating subsystems. Put another way, while we established in Section 3.1 that SaaS apps follow a client–server architecture, we have said nothing about the organization of the server application. In this section we use an architectural pattern called **Model–View–Controller** (usually shortened to MVC) to do so.

An application organized according to MVC consists of three main types of code. Models are concerned with the data manipulated by the application: how to store it, how to operate on it, and how to change it. An MVC app typically has a model for each type of entity manipulated by the app. For example, for a movie database app in which moviegoers (users) can write reviews, the entities would include (at least) movies, moviegoers, and reviews. (Towards the end of this chapter, a CHIPS exercise will introduce such an app, *RottenPotatoes*, which we'll use throughout the rest of the book.)

Views are presented to the user and contain information about the models with which users can interact. The views serve as the interface between the system's users and its data; for example, in *RottenPotatoes* you can list movies and add new movies by clicking on links or buttons in the views. There is only one kind of model in *Rotten Potatoes*, but it is associated with a variety of views: one view lists all the movies, another view shows the details of a particular movie, and yet other views appear when creating new movies or editing existing ones.

Finally, controllers mediate the interaction in both directions: when a user interacts with a view (for example, by clicking something on a Web page or submitting a form), a specific controller *action* corresponding to that user activity is invoked. Each controller corresponds to one model, and in Rails, each controller action is handled by a particular Ruby method within that controller. The controller can ask the model to retrieve or modify information; depending on the results of doing this, the controller decides what view will be presented next to the user, and supplies that view with any necessary information. Since *RottenPotatoes* has only one model (*Movies*), it also has only one controller, the *Movies* controller. The actions defined in that controller can handle each type of user interaction with any *Movie* view (clicking on links or buttons, for example) and contain the necessary logic to obtain Model data to *render* any of the *Movie* views.

Given that SaaS apps have always been view-centric and have always relied on a persistence tier, Rails' choice of MVC as the underlying architecture might seem like an obvious fit, but there are caveats. Technically, while a View in the MVC sense means “any logic needed to display something,” a Rails view is really a special case called a *template* or *template view*, in which static markup interspersed with variable substitution is processed by a generic logic engine. As Figure 4.1 shows, other patterns are possible for view-oriented frameworks. Model–View–Presenter can be thought of as an embellishment of Model–View–Controller, and Model–View–ViewModel provides two-way interaction between the model and the view so that updates to the view automatically occur in the model, in contrast to MVC in which the focus is on reflecting model changes in the view.

As a software engineer who is experienced at learning new stacks, you will be in a position to learn by asking other experienced colleagues to give you a “developer’s-eye view” of a new language or framework. If you asked an experienced Rails developer to concisely describe the framework to you, you might get something like the following description.

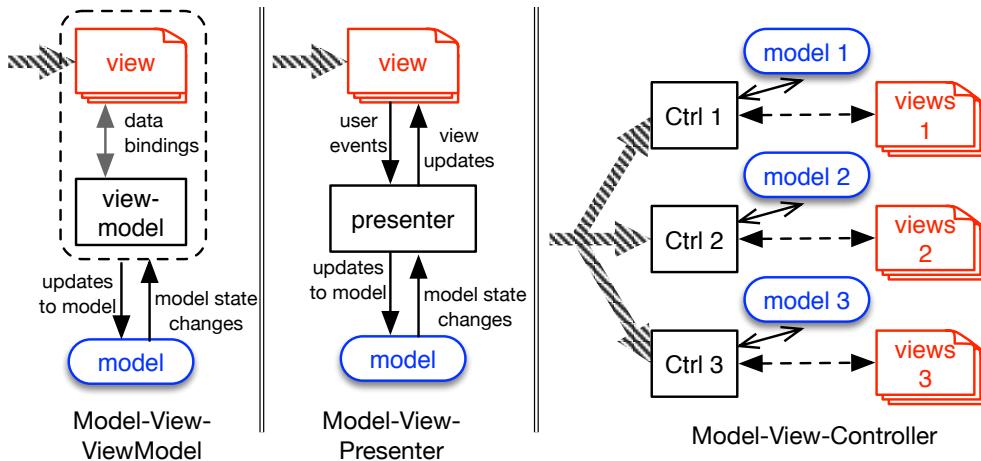


Figure 4.1: Three architectural patterns for Web apps that include a GUI. In all cases, the Model contains the main business logic. Model-View-Controller (MVC, right), which Rails implements, makes sense when the views (HTML pages) are passive and all user interaction (clicks, form field events, and so on) must be handled by the controller, which in Rails is part of the server code. But modern Rails apps are no longer “pure” MVC: Model-View-Presenter (MVP, center) concentrates all UI-handling for the view in a Presenter module, which more accurately describes rich Web apps in which client-side JavaScript explicitly handles many UI interactions. Model-View-ViewModel (MVVM, left), also called Model-View-Binder and originally invented for developing GUI applications with Microsoft .NET, goes a step further and reifies two-way binding between a view and its data: UI interactions on the view automatically affect the data, and updating the data automatically updates the view. JavaScript frameworks such as Angular and Vue largely follow this pattern.

- Rails is designed to support apps that follow the Model–View–Controller pattern. The framework provides powerful base classes from which your app’s models, views, and controllers inherit.
- Each Rails model is a resource type whose instances are rows in a particular table of a relational database. The database-stored models are exposed to Ruby code via a design pattern known as Active Record (Section 4.2), in which each type of model behaves more or less like a data structure whose fields (attributes) are semi-automatically serialized to the database.
- A Rails app is best viewed as a collection of RESTful resources, each consisting of its own model, controller, and set of views. Resources may have relationships to each other; for example, in a movie-reviewing application, we might say that Movie and Review are each a type of resource, that a single Movie can have many Reviews, and that any given Review belongs to some Movie. **Foreign keys** in the database tables (Section 5.4) capture such relationships.
- Because Rails is a server-side framework, it needs a way to map an HTTP route (Section 3.2) to code in the app that performs the correct action. The Rails routing subsystem (Section 4.4) provides a flexible way to map routes to Ruby methods located in Rails controllers. You can define routes any way you like, but if you choose to use some “standard” routes based on RESTful conventions, most of the routing is set up for you automatically.
- Rails was originally designed for apps whose client was a Web browser, so its view subsystem is designed around generating HTML pages. But it is equally easy to generate

(for example) JSON data structures to return via a RESTful API.

- Rails embodies strong opinions about many mechanical details of your app’s implementation, such as how classes and files are named and where they are stored. If you follow these opinions, Rails uses ***convention over configuration*** to save you a lot of work. For example, rather than explicitly specifying a mapping from a model class to the name of its corresponding database table or the filenames and class names of its associated controller and view files, Rails infers all these things based on simple naming rules.



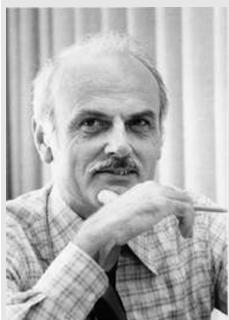
Summary

- The ***Model-View-Controller*** or MVC design pattern is one of a family of patterns for structuring interactive applications. MVC distinguishes *models* that implement business logic, *views* that present information to the user and allow the user to interact with the app, and *controllers* that mediate the interaction between views and models.
- In MVC SaaS apps such as those built using Rails, every user action that can be performed on a web page—clicking a link or button, submitting a fill-in form, or using drag-and-drop—is eventually handled by some controller action, which will consult the model(s) as needed to obtain information and generate a view in response.
- Rails heavily uses convention over configuration: if you agree to follow certain rules about naming files and classes, you are freed from having to specify explicitly which file or class supports the functionality of each model, view, or controller.

Self-Check 4.1.1. *In which element of the MVC model is the app code “farthest away” from the user? Briefly explain your answer.*

◊ The model code is “farthest away” from the user. Users interact directly with views (which should have little to no code) and code in controllers handles users’ requests for interaction, but model code is invoked only by the controller when needed. ■

Edgar F. “Ted” Codd (1923–2003) received the 1981 Turing Award for inventing the ***relational algebra*** formalism underlying relational databases.



4.2 Rails Models: Databases and Active Record

Every nontrivial application needs to store and manipulate persistent data. For many SaaS applications, the two key requirements may be expressed as follows:

1. The app must be able to store different types of data items, or *entities*, in which all instances of a particular type of entity share a common set of *attributes*. For example, in RottenPotatoes, the attributes for a movie entity might include title, release date, MPAA rating, and so on. All movies have the same attributes, though the attribute *values* are different for each movie.
2. The app must be able to express relationships among different kinds of entities. Returning to RottenPotatoes, two other entities might be movie reviews and moviegoers. A movie has many reviews and a moviegoer has many reviews, though any single review is associated with exactly one movie and one moviegoer.

id	title	rating	release_date	description
1	Hamilton	PG-13	2020-07-03	The groundbreaking American musical...
2	Casablanca	PG	1942-11-26	Casablanca is a...

Figure 4.2: A possible RDBMS table for storing movie information. The `id` column gives each row's *primary key*—a permanent and unique identifier that is never reused, even if that row is deleted. Most databases can be configured to assign primary keys automatically in various ways; Rails uses the very common convention of assigning integers in increasing order as new rows are created.

The above two requirements are so common in business that *Relational database management systems* (RDBMSs) evolved in the early 1970s as elegant structured-storage systems whose design was based on a formalism for representing such structure and relationships. An RDBMS stores a collection of *tables*, each of which stores entities with a common set of *attributes*. One row in the table corresponds to one entity, and the columns in that row correspond to the attribute values for that entity. The `movies` table for RottenPotatoes would include columns for `title`, `rating`, `release_date`, and `description`, and the rows of the table look like Figure 4.2.

In Rails, data takes the form of a set of resources stored in a relational database. Amazingly, you don't need to know much about how RDBMSs work to get started with Rails, though understanding their basic operation becomes more important as your apps begin to comprise multiple types of resources with relationships among them.

Therefore, the key questions to address in order to understand the role of the database in the Rails Model–View–Controller architecture are as follows:

1. What is the correspondence between how an instance of a resource (say, the information about a specific movie) is stored in the database and how it is represented in the programming language used by the framework (in this case, Ruby)?
2. What software mechanisms mediate between those two representations, and what programming abstractions do those mechanisms expose?

In our case, the answer is that Rails implements the **Active Record architectural pattern**. In this pattern, a Rails model is a class backed by a specific table of an RDBMS. An instance of the class (for example, the entry for a single movie) corresponds to a single row in that table. The model has built-in behaviors that directly operate on the database:

- Create a new row in the table (representing a new object),
- Read an existing row into a single object instance,
- Update an existing row with new attribute values from a modified object instance,
- Delete a row (destroying the object's data forever).

Active Record refers to the pattern itself;
ActiveRecord refers to the code module that instantiates the pattern in the Rails framework.

This collection of four commands is often abbreviated **CRUD**. The combination of table name and `id` uniquely identifies a model stored in the database, and as we will see, is therefore how objects are usually referenced in RESTful routes in Rails apps.

Unlike some other SaaS frameworks in which the abstraction exposed to the developer is the connection to the database itself, Active Record gives each model the knowledge of how to create, read, update, and delete instances of itself in the database (CRUD). That is, all of the logic for “talking to” the database, and (critically) for how to marshal and unmarshal

(serialize or deserialize) attributes, is implicitly included in each model. Rails accomplishes this by providing a class `ActiveRecord::Base` from which your models will inherit. In OOP terms, Create and Read are class methods, since they define actions on the *collection* of model instances as a whole, whereas Update and Delete are instance methods, since they define actions on a specific model instance.

Remarkably, as Figure 4.3 shows, simply defining a class that descends from Rails' `ActiveRecord::Base` class provides all the necessary machinery to “connect” the model to the database. Specifically:

- The directory `app/models` is expected to contain one Ruby code file per model. The file name is determined by converting the model's class name to `lower_snake_case`, so the file `app/models/movie.rb` is expected to define the class `Movie`.
- The database table name is determined by converting the model's class name to `lower_snake_case` and pluralizing it. For example, instances of model `AccountCode` would be stored in table `account_codes`.
- The attributes of the model, and their types (string, integer, date, and so on), are inferred from the names and types of the table's columns.
- The model automatically has class (static) methods `new` and `create`, among others, that expect a hash of arguments whose keys match those attribute names and whose values supply the attribute values for a movie instance to be created in memory (`new`) or immediately persisted in the database (`create`).

In fact, just about the only thing this class definition *doesn't* do is create the actual table in the database; we must do that ourselves, by first telling Rails how to actually connect to the database, and then providing instructions for creating the necessary model table(s) in our schema. When a new Rails app is created from scratch, the automatically-generated file `config/database.yml` specifies how to connect to the database. By default, Rails apps are initially configured to use SQLite, a lightweight single-user RDBMS, but later we will see how to modify this file to connect to “industrial strength” database servers such as Postgres or MySQL. To create the actual table, we create and apply a *migration*—a Ruby script describing a set of changes to make to the database schema.

 Why use migrations rather than directly issuing SQL statements such as `create table`? There are many reasons, but as we will see, Rails defines three *environments* in which your app can run: development (when you're coding), production (the live app containing real customer data), and test (used only when running automated tests). Each environment gets its own completely separate database, but of course, the schemata of all three databases need to be kept in sync. It is much less error-prone to write a single migration script and run it against each environment than to ensure you issue the exact same set of SQL commands three times.

To create and apply a migration, you first give the command `rails generate migration name`, where `name` is some descriptive name for what the migration does; in this example, we might say `rails generate migration create_movies_table`. Rails will create a Ruby file whose name consists of your migration's name plus a timestamp. The file defines a migration class with your specified name that descends from `ActiveRecord::Migration` and has an empty `change` instance method. You fill in that method with the desired schema changes, save the file, and then run the command

YAML (“YAML Ain't Markup Language”) can express hierarchical data structures comparable to those covered by JSON.

<https://gist.github.com/0fa79aaa81f0ec133a38de8bf9a2150a>

```
1 class Movie < ActiveRecord::Base
2 end
```

<https://gist.github.com/9170d0cedfc2c7897f28feb134190ce2>

```
1 class CreateMovies < ActiveRecord::Migration
2   def change
3     create_table 'movies' do |t|
4       t.string 'title'
5       t.string 'rating'
6       t.text 'description'
7       t.datetime 'release_date'
8       # Add fields that let Rails automatically keep track
9       # of when movies are added or modified:
10      t.timestamps
11    end
12  end
13 end
```

<https://gist.github.com/aa0a53221b45188929d2c91dc98424a0>

```
1 # Seed the RottenPotatoes DB with some movies.
2 more_movies = [
3   { :title => 'Aladdin', :rating => 'G',
4    :release_date => '25-Nov-1992'},
5   { :title => 'When Harry Met Sally', :rating => 'R',
6    :release_date => '21-Jul-1989'},
7   { :title => 'The Help', :rating => 'PG-13',
8    :release_date => '10-Aug-2011'},
9   { :title => 'Raiders of the Lost Ark', :rating => 'PG',
10    :release_date => '12-Jun-1981'}
11 ]
12
13 more_movies.each do |movie|
14   Movie.create(movie)
15 end
```

Figure 4.3: Top: A minimal valid ActiveRecord class. Middle: A migration makes changes to the database schema, in this case to create the table that the model expects to find. Bottom: the `create` class method creates a movie instance in the database from a hash whose keys are the table's column names.

`rake db:migrate`, which invokes the Rails utility tool `rake` to run the task `db:migrate`. (`rake -T` shows a list of available tasks with brief descriptions.)

Notice that you don't have to specify the filename of which migration to apply: Rails tracks which migrations have been applied to which environments' databases. The `db:migrate` task examines the **environment variable** `RAILS_ENV` to determine which environment to apply the migration in, defaulting to `development` if not set, and then applies *all* pending migrations not yet applied to that database. Running `rake db:migrate` multiple times is harmless, since migrations already applied will simply be ignored on subsequent runs.

The next CHIPS exercise gives you some hands-on practice with how the Rails implementation of Active Record actually works.

Summary

- The Rails implementation of ActiveRecord uses convention over configuration to infer database table names from the names of model classes, and to infer the names and types of the columns (attributes) associated with a given kind of model.
- Basic Active Record support focuses on the CRUD actions: create, read, update, delete.
- Every model instance saved in the database receives an ID number unique within its table called the primary key, whose attribute name (and therefore column name in the table) is `id` and which is never “recycled” (even if the corresponding row is deleted). The combination of table name and `id` uniquely identifies a model instance stored in the database, and is therefore how objects are usually referenced in RESTful routes.
- Changing the database schema, including creating the tables needed by your models, is accomplished by creating and running migrations. Rails itself tracks which migrations have been applied in each of the three environments—development, production, and testing.

■ *Elaboration: Overriding convention over configuration*

Convention over configuration is great, but there are times you may need to override it. For example, if you're trying to integrate your Rails app with a non-Rails legacy app, the database tables may already have names that don't match the names of your models, or you may want friendlier attribute names than those given by taking the names of the table's columns. All of these defaults can be overridden at the expense of more code, as the ActiveRecord documentation describes. In this book we choose to reap the benefits of conciseness by sticking to the conventions.

Self-Check 4.2.1. *What do you think would happen if you tried to run the code in the top and bottom parts of Figure 4.3 without having created and run the migration in the middle part of the figure?*

◊ An error would occur upon the first call to any method in the `Movie` class that requires accessing the database, since it would be unable to find any table named `movies`. ■

Helper method	URI returned	RESTful route and action	
<code>movies_path</code>	<code>/movies</code>	<code>GET /movies</code>	index
<code>movies_path</code>	<code>/movies</code>	<code>POST /movies</code>	create
<code>new_movie_path</code>	<code>/movies/new</code>	<code>GET /movies/new</code>	new
<code>edit_movie_path(m)</code>	<code>/movies/:id/edit</code>	<code>GET /movies/:id/edit</code>	edit
<code>movie_path(m)</code>	<code>/movies/:id</code>	<code>GET /movies/:id</code>	show
<code>movie_path(m)</code>	<code>/movies/:id</code>	<code>PUT /movies/:id</code>	update
<code>movie_path(m)</code>	<code>/movies/:id</code>	<code>DELETE /movies/:id</code>	destroy

Figure 4.4: The set of RESTful routes generated by the single line resources '`'movies'`' in a Rails app's `routes.rb` file. You can display a table like this by running the command `rake routes` in the root directory of your app.

4.3 CHIPS: ActiveRecord Basics



CHIPS 4.3: ActiveRecord Basics

<https://github.com/saasbook/hw-activerecord-practice>

Write ActiveRecord operations to manipulate a database of fictional customers, as a way of learning ActiveRecord's basic features before using it in Rails apps.

4.4 Routes, Controllers, and Views

We've now been introduced to how Rails implements the models in MVC, but when users interact with a SaaS app via a browser, they're interacting with views and invoking controller actions, either by typing URIs into their browser (resulting in an HTTP GET) or interacting with page elements that generate GET requests (links) or POST requests (forms). In this section we take a tour through views and controllers to understand the lifecycle of such a request when it hits a Rails app. We first explore the controllers and views corresponding to REST actions that only read model data: Index and Read. In Section 4.6 we consider controllers and views corresponding to actions that modify data: Create, Update, and Delete.

As we know from Section 3.2, our app will receive a request in the form of an HTTP route. The first step in a Rails app is therefore to determine which code in the app should be invoked to handle that route. Rails provides a flexible routing subsystem that maps routes to specific Ruby methods in each controller using the contents of the file `config/routes.rb`. You can define any routes you like there, but if your app is RESTful (centered around CRUD requests against a set of resources) and you abide by convention over configuration, the single line `resources 'movies'` (in our case) defines a complete set of RESTful routes for a model (resource) called `Movies`, as Figure 4.4 shows.

Although "RESTful route and action" column in the table should look familiar from Section 3.2, we raise four questions about it:

1. The four CRUD actions plus the Index action should only need five routes; why are there seven?
2. Most Web browsers can only generate HTTP GET and POST requests; how can a browser generate a route such as `Update`, which uses HTTP PUT?



`resources :movies`
would also work: like table
and column names in
migrations, resource names
in the routes file may be
either strings or symbols.

3. Some routes such as `show` include a variable (parameter) as part of the route URI, and others such as `create` must also provide the attribute values of the entity to be created as parameters. How are these parameters and their values made available to the controller action?
4. Finally, what are the “route helper methods” referred to in the table and why are they needed?

The first question—why seven routes rather than five—is easy but subtle. We preview the answer here and will return to it when we discuss HTML forms in Section 4.6. A RESTful request to create a movie would typically include information about the movie itself—title, rating, and so on. But in a user-facing app, we need a way to collect that information interactively from the user, usually by displaying a form the user can fill in. Submitting the form would clearly correspond to the `create` action, but what route describes *displaying* the form? The Rails approach is to define a default RESTful route `new` that displays whatever is necessary to allow collecting information from the user in preparation for a `create` request. A similar argument applies to `update`, which requires a way to show the user an editable version of the *existing* resource so the user can make changes; this latter action is called `edit` in rails, and typically displays a form pre-populated with the existing resource’s attribute values.

Actually, most browsers also implement HEAD, which requests metadata about a resource, but we needn’t worry about that here.

Turning to the second question, for historical reasons Web browsers only implement GET (for following a link) and POST (for submitting forms). To compensate, Rails’ routing mechanism lets browsers use POST for requests that normally would require PUT or DELETE. Rails annotates the Web forms associated with such requests so that when the request is submitted, Rails *internally* changes the HTTP method “seen” by the controller to PUT or DELETE as appropriate. The result is that the Rails programmer can operate under the assumption that PUT and DELETE are actually supported, even though browsers don’t implement them. As a result, the *same set of routes* can handle either requests coming from a browser (that is, from a human being) or requests coming from another service in a SOA.

What about routes that include a parameter in the URI, such as `show`, or those that must also include parameters corresponding to attribute values for a resource, such as `create`? As we will see in the code examples in this section (and you will have an opportunity to experiment with in the next CHIPS), the Rails routing subsystem prepares a hash called `params[]` that is made available to the controller. With the above `routes.rb` file as part of an app, typing `rake routes` at the command line (within the root directory of your app) will list all the routes implied by that file, showing wildcard parameters with the colon notation introduced in Section 3.5. For example, the route for `show` will appear as `GET /movies/:id`, which tells us that `params[:id]` will hold the actual ID value parsed from the URI. Further, as we will see, Rails provides an easy way to generate an HTML form in which the form fields are named in such a way that another value in `params`, in this example `params[:movie]`, is itself a hash of key/value pairs corresponding to a `Movie` object’s attributes and their desired values. This mechanism sounds more confusing than it actually is, as the code examples below will show.

Finally, what are “route helpers”? By convention over configuration, the route URIs will match the resource name, but as we’ll see later, you can override this behavior. You might, for example, decide later that you’d rather have your routes built around `film` rather than `movie`. But then any view in your app that references the old-style `movie` route URIs—for example, the page that serves the form allowing users to edit a movie’s info—would have to be changed to `film`. This is the problem that route helpers solve: they decouple what the

route does (create, read, and so on) from the actual route URI. As the table suggests, the Ruby method `movies_path` will return the correct URI for the route “list all movies,” *even if* the URI text itself is changed later (or for “create new movie,” if POST is used as the route’s verb). Similarly `movie_path(23)` will always return the correct URI for “show movie ID 23” (or update, edit, or destroy movie ID 23, depending on which HTTP verb is used). The route helpers also make explicit what the route is supposed to do, improving readability.

What about the controller methods (called controller *actions* in Rails) that handle each RESTful operation? Once again, convention over configuration comes to the rescue. By default, the routes created by `resources 'movies'` will expect to find a file `controllers/movies_controller.rb` that defines a class `MoviesController` (which descends from the Rails-provided `ApplicationController`, just as models descend from `ActiveRecord::Base`). That class will be expected to define instance methods `index`, `new`, `create`, `show` (read), `edit`, `update`, and `destroy`, corresponding to the RESTful actions of Figure 4.4.



Each of these controller actions generally follows a similar pattern:

1. Collect the information accompanying the RESTful request: parameters, resource IDs in the URI, and so on
2. Determine what ActiveRecord operations are necessary to fulfill the request. For example, the `Index` action might just require retrieving a list of all movies from the `Movies` table; the `Update` action might require identifying a resource ID from the URI, parsing the contents of a form, and using the form data to update the movie with the given ID (primary key); and so on.
3. Set instance variables for any information that will need to be displayed in the view, such as information retrieved from the database.
4. Render a view that will be returned as the result of the overall request.

That leaves only the last bullet point: how does each controller action select a view, and how is the information generated in the controller action made available to that view?

You should no longer be surprised to hear that part of the answer lies once again in convention over configuration. Controller actions do not return a value; instead, when a controller action finishes executing, by default Rails will identify and render a view named `app/views/model-name/action.html.erb`, for example `app/views/movies/show.html.erb` for the `show` action in `MoviesController`. The Rails module that choreographs how views are handled is `ActionView::Base`. This view consists of HTML interspersed with Erc (Embedded Ruby) tags that allow the results of evaluating Ruby code to be interpolated into the HTML view. In particular, any instance variables set in the controller method become available in the view.



Warning!
Counter-intuitively, if there is
no controller action
matching a route but there is
a view, Rails will render that
view. See Fallacies & Pitfalls
for details.

Rereading the previous sentence should give you pause. Why would instance variables of one class (`MoviesController`) be accessible to an object of a completely different class (`ActionView::Base`), violating all OOP orthodoxy? The simple reason is that the designers of Rails thought it would make coding easier. What actually happens is that Rails creates an instance of `ActionView::Base` to handle rendering the view, and then uses Ruby’s metaprogramming facilities to “copy” all of the controller’s instance variables into that new object!

- The controller code is in class `MoviesController`, defined in `app/controllers/movies_controller.rb` (note that the model's class name is pluralized to form the controller file name.) Your app's controllers all inherit from your app's root controller `ApplicationController` (in `app/controllers/application_controller.rb`), which contains controller behaviors common to multiple controllers (we will meet some in Chapter 5) and in turn inherits from `ActionController::Base`.
- Each instance method of the controller is named using `lower_snake_case` according to the RESTful action it handles, plus the two “pseudo-actions” `new` and `edit`.
- The view template for each action is named the same as the controller method itself, so the view for Showing a movie would be in `app/views/movies/show.html.erb`. Strangely but conveniently, each view has access to the instance variables set in the controller action that caused the view to be rendered.

Sanitization To help thwart cross-site scripting and similar attacks described in Chapter 12, Erb sanitizes² Ruby output before interpolating it into the HTML.

There's one last thing to notice about these views: they aren't legal HTML! In particular, they lack an HTML DOCTYPE, the `<html>` element, and its children `<head>` and `<body>`. In fact, we need to put those elements in `views/application.html.erb`, which “wraps” all views by default, as Figure 4.6 shows.

Summary:

- Rails provides various helper methods that take advantage of the RESTful route URIs, including `link_to` for generating HTML links whose URIs refer to RESTful actions.
- Convention over configuration is used to determine the file names for controllers and views corresponding to a given model. If the RESTful route helpers are used, as in `resources :movies`, convention over configuration also maps RESTful action names to controller action (method) names.
- Although the most common way to finish a controller action is to render the view corresponding to that action, for some actions such as `create` it's more helpful to send the user back to a different view. Using `redirect_to` replaces the default view rendering with a redirection to a different action.
- Although redirection triggers the browser to start a brand-new HTTP request, the `flash` can be used to save a small amount of information that will be made available to that new request, for example, to display useful information to the user regarding the redirect.



■ Elaboration: Metaprogramming in Rails

So far we have seen two examples of how Rails combines convention over configuration with on-the-fly code generation. In ActiveRecord, getter and setter instance methods for ActiveRecord models are defined at runtime by inspecting the database schema. In the routing subsystem, route helpers such as `new_movie_path` are defined at runtime based on the contents of `config/routes.rb`. (While controller methods automatically knowing how to locate the default view to render is also an example of convention over configuration, there's no new code that needs to be generated at runtime to support it.)

<https://gist.github.com/f27917e167b6e7953fd4ee646f1313be>

```

1 # This file is app/controllers/movies_controller.rb
2 class MoviesController < ApplicationController
3   def index
4     @movies = Movie.all
5   end
6   def show
7     id = params[:id]           # retrieve movie ID from URI route
8     @movie = Movie.find(id)    # look up movie by unique ID
9   end
10 end

```

<https://gist.github.com/ef90ccfad3b471295ba0286434413121>

```

1 <h1>All Movies</h1>
2
3 <%= link_to 'Add Movie', new_movie_path, :class => 'btn btn-primary' %>
4
5 <div id="movies">
6   <div class="row">
7     <div class="col-8">Movie Title</div>
8     <div class="col-2">Rating</div>
9     <div class="col-2">Release Date</div>
10  </div>
11  <% @movies.each do |movie| %>
12    <div class="row">
13      <div class="col-8"> <%= link_to movie.title, movie_path(movie) %> </div>
14      <div class="col-2"> <%= movie.rating %></div>
15      <div class="col-2"> <%= movie.release_date.strftime('%F') %> </div>
16    </div>
17  <% end %>
18 </div>

```

<https://gist.github.com/e497af7a26ff6d79c80f0dc3cb65b>

```

1 <h1>Details about <%= @movie.title %></h1>
2
3 <div id="metadata">
4   <ul id="details">
5     <li> Rating: <%= @movie.rating %> </li>
6     <li> Released on: <%= @movie.release_date.strftime('%F') %> </li>
7   </ul>
8 </div>
9
10 <div id="description">
11   <h2>Description:</h2>
12   <p> <%= @movie.description %> </p>
13 </div>
14
15 <%= link_to 'Edit this movie', edit_movie_path(@movie), :class => 'btn' %>
16 <%= link_to 'Back to movie list', movies_path, :class => 'btn btn-primary' %>

```

Figure 4.5: The controller code and template markup to support the RESTful actions `index` and `show`. The `index` method just retrieves all movies, while the `show` method examines `params[]`, which has been prepared by the routing subsystem, to retrieve the desired movie's `id` field from the route URI. Controller instance variables defined in each action are available in the corresponding view. The `row` and `col-n` classes applied to the `div` elements in the movie list take advantage of the Bootstrap framework's 12-column grid system to display a tabular view.

<https://gist.github.com/d8c76aa83914c7f074eda9cb815da94e>

```

1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title> RottenPotatoes! </title>
5      <link rel="stylesheet" href="https://getbootstrap.com/docs/4.0/dist/css/
6          bootstrap.min.css">
7      <%= javascript_include_tag :application %>
8      <%= csrf_meta_tags %>
9    </head>
10   <body>
11     <div class="container">
12       <% if flash[:notice] %>
13         <div class="alert alert-info text-center"><%= flash[:notice] %></div>
14       <% elsif flash[:alert] %>
15         <div class="alert alert-danger text-center"><%= flash[:notice] %></div>
16       <% end %>
17       <%= yield %>
18     </div>
19   </body>
</html>
```

Figure 4.6: Our generic application template “wraps” every view. Rails provides a generic version of this file when you start a new app; in our version, we have added a line to use the Bootstrap CSS framework, and annotated the HTML elements in our individual views to use Bootstrap’s classes. Section 4.6 explains the meaning and purpose of lines 11–15.

Self-Check 4.4.1. *The route helpers for Show and Update take an argument, as in movie_path(@movie), but the route helpers for New and Create (new_movie_path and movies_path) do not. Why the difference?*

- ◊ The argument to the Show and Update route helpers is either an existing Movie instance or the ID (primary key) of an existing instance. Show and Update operate on existing movies, so they take an argument to identify which movie to operate on. New and Create operate on not-yet-existing movies. ■

Self-Check 4.4.2. *Why doesn’t the route helper movies_path for the Index action take an argument? (Hint: The reason is slightly different than the answer to the previous question!)*

- ◊ The Index action just shows a list of all the movies, so no argument is needed to distinguish which movie to operate on. ■

4.5 CHIPS: Rails Routes



CHIPS 4.5: Rails Routes

<https://github.com/saasbook/rails-routing-practice>

Specify routes in a Rails app and identify how parts of a route map to parameters and other data made available to Rails controllers, using the Rails routing practice app at <http://rails-routing-practice.saasbook.info>.

<https://gist.github.com/7af0ab0fe2db36f7723998e9d56ff637>

```

1 <h2>Create New Movie</h2>
2 <%= form_tag movies_path, :method => :post, :class => 'form' do %>
3   <%= label :movie, :title, 'Title', :class => 'form-control' %>
4   <%= text_field :movie, :title, :class => 'form-control' %>
5   <%= label :movie, :rating, 'Rating', :class => 'form-control' %>
6   <%= select :movie, :rating, ['G', 'PG', 'PG-13', 'R', 'NC-17'], :class => 'form-
control' %>
7   <%= label :movie, :release_date, 'Released On' %>
8   <%= date_select :movie, :release_date, :class => 'form-control' %>
9   <%= submit_tag 'Save Changes' %>
10 <% end %>
```

Figure 4.7: The form the user sees for creating and adding a new movie to RottenPotatoes. Rather than coding HTML form elements directly, we use Rails’ form helpers, which will name the form fields in such a way that the controller can parse them easily to populate a new `Movie` object. The `:class=>name` option sets the HTML class(es) of an element; we apply Bootstrap’s classes `form` and `form-control` to the form and its elements respectively.

4.6 Forms

So far we’ve looked at views that display data to the user, but not views that collect data *from* the user. The simplest mechanism for doing so consist of HTML forms. To create them, we address the following steps:

1. How do we display a fill-in form to the user?
2. How is the information filled in by the user made available to the controller action, so that it can be used in a `create` or `update` ActiveRecord call?

The first step is straightforward: As Section 4.4 described, Rails by default defines a `new` action (and route) for this purpose. Following the pattern seen so far and illustrated in Figure 4.4:

- The route `GET /movies/new` names the action, and the route helper `new_movie_path` will generate the URI portion of the route;
- The action will be handled by the method `MoviesController#new`;
- By default, the controller action will end by rendering a view `app/views/movies/_new.html.erb`.



It seems logical to assume that that view should contain an HTML form that submits to the `create` RESTful route. When that HTML form is submitted, a controller action will need to parse the form data and do something with it—in our case, use it to populate the attributes of a new `Movie` object. To do so, the controller action must have knowledge about how the HTML form fields are named, so that it can extract data from each field and use that data to populate attributes of an instance of the `Movie` model. Such a scenario—a form whose fields represent attributes of an ActiveRecord model object—is so common that Rails streamlines the process by using form tag helper methods³. These helpers are Ruby methods that generate HTML form tags whose names follow particular conventions that make them easy to parse by the controller action. Figure 4.7 shows an example; watch Screencast 4.6.1 for a description of what’s going on in it.

Screencast 4.6.1: Views with fill-in forms.

<http://youtu.be/2tvI8tPyy9c>

The `form_tag` method for generating a form requires a route to which the form should be submitted—that is, a URI and an HTTP verb. We use the RESTful URI helper and HTTP POST method to generate a route to the `create` action, as `rake routes` reminds us.

Specifically, the `form_tag` helper takes two arguments. The first is the URI to which the form should submit; in this case, the RESTful route helper (Figure 4.4) is used to generate that URI. The second argument is a hash of optional arguments, one of which may be the HTTP method that should be used to submit the form. POST is the default so we didn't need to specify it here; GET is acceptable if submitting the form doesn't change any application data. If you specify PUT, PATCH, or DELETE, as Section 4.4 described, the browser will still use POST to submit the form but Rails will “automagically” make it appear that the form was submitted using the method you specified.

Not all input field types are supported by the form tag helpers (in this case, the date fields aren't supported), and in some cases you need to generate forms whose fields don't necessarily correspond to the attributes of some ActiveRecord object. The Rails guide on form tag helpers⁴ and Rails API documentation⁵ describe the various form tag helper options in detail.

Note that just as with the RESTful route helpers (Figure 4.4), you are not *required* to use form tag helpers, but as we'll see next, given the common flow of submitting a form related to a model and using the form's information to create or update a model instance, using them actually saves you work and results in less code in your views.

To recap where we are, we created the `new` controller method that will render a view giving the user a form to fill in, placed that view in `new.html.erb`, and arranged to have the form submitted to the `create` controller method. All that remains is to have the `create` controller action parse the form information and use it to create a new movie in the database.

Recall from the examples in Section 4.2 that the `Movie.create` call takes a hash of attribute names and values to create a new movie instance in the database (in contrast to `Movie.new`, which creates an instance of the Ruby class `Movie` but does not save it in the database). The reason to use form tag helpers now becomes clear: if you look at the HTML page generated by the `new.html.erb` template, you'll see that the form fields created by the form tag helpers have names of the form '`movie[title]`', '`movie[rating]`', and so on. As a result, the value of `params['movie']` is *a hash of movie attribute names and values*, which we can pass along directly using `Movie.create!(params['movie'])`.

Or `params[:movie]`, since `params` is another example of a Rails hash-like object whose keys can be referenced as either strings or symbols.



It is worth reading the above paragraph again: the form tag helpers give names to the form fields that result in `params['movie']` containing exactly what needs to be passed to ActiveRecord's `create` or `update_attributes` methods. This streamlining is not only a great example of convention over configuration, but also an example of how a framework can simplify the “mechanical work” of making the models, views, and controllers in a SaaS app work smoothly together.

We must, however, attend to an important detail before our controller action will work. “Mass assignment” of a whole set of attributes, as occurs when we pass the hash `params['movie']` to an ActiveRecord call, is a mechanism that could be used by a malicious attacker to set model attributes that shouldn't be changeable by regular users⁶. As Figure 4.8 shows, Rails requires us to declare in the controller which elements of `params` are *required* to be present (if any) for a given action, and critically, which elements are *permitted*

<https://gist.github.com/2156d1ba12108d25463fb9d8563b13d5>

```

1 class MoviesController < ApplicationController
2   def create
3     params.require(:movie)
4     params[:movie].permit(:title,:rating,:release_date)
5     # shortcut: params.require(:movie).permit(:title,:rating,:release_date)
6     # rest of code...
7   end
8 end

```

Figure 4.8: Rails’ “strong parameters” mechanism requires us to declare which values from `params` are required to be present and which values are permitted to be used to update the model. `require` and `permit` operate on the `params` hash or any of its sub-hashes, as the Rails documentation⁸ explains. In Section 5.1 we introduce before-filters, which can be used to DRY out this code rather than repeating it in any controller action that might try to create or modify a model instance.

to be assigned to model attributes. This mechanism follows the **principle of least privilege** in computer security, a topic to which we return in Section 12.9 when discussing how to defend customer data.

The screencast shows how mass-assignment works in practice, and also shows the helpful technique of using debug breakpoints to provide a detailed look “under the hood” during execution of a controller action.

Screencast 4.6.2: The Create action.

<http://youtu.be/SJ2wnTxPXC4>

Inside the `create` controller action, we placed a debug breakpoint to inspect what’s going on, and used a subset of the debugger commands in Figure 4.10 to inspect the `params` hash. In particular, because our form’s field names all looked like `movie[...]`, `params['movie']` is itself a hash with the various movie fields, ready for assigning to a new `Movie` object.



At this point, the controller action has used `params['movies']` to create a new movie record, ostensibly successfully. But what should we display when the `create` action completes? Strict convention over configuration suggests a view `app/views/movies/create.html.erb` that simply confirms the movie was created, and provides a link or other mechanism to go back to the list of movies, but it seems clumsy to have a separate view just to do that. One alternative would be to streamline the user experience by just sending the user back to the newly-updated list of all movies (Index action), but arrange to display a confirmation message somewhere on the page that the movie was added successfully.

Since we already have an action and view to handle Index, the DRY way to proceed is to have the controller action issue an **HTTP redirect**, telling the Web browser to start an entirely new request for the Index action. But this approach presents a small problem. Since HTTP is stateless, when this new Index request is routed by Rails to our controller’s `index` method, all of the variables associated with the prior `create` request are gone. That is a problem if we want to display a friendly message at the top of the movie list, since at the moment of the `index` call, we no longer know the name or ID of the previously-created movie.



To address this common scenario, the `flash[]` is a special object available in a controller that quacks like a hash, but whose contents persist only from the current request to the next. In other words, if we put something into `flash[]` during the current controller action, we can access it during the *very next* action, but not during any subsequent actions. The entire hash is persisted, but by convention, `flash[:notice]` is used for informational messages



<https://gist.github.com/579c3846f57f8133cd6e1953ef259c74>

```

1 class MoviesController < ApplicationController
2   # 'index' and 'show' methods from Section 4.4 omitted for clarity
3   def new
4     @movie = Movie.new
5   end
6   def create
7     if (@movie = Movie.create(movie_params))
8       redirect_to movies_path, :notice => "#{@movie.title} created."
9     else
10      flash[:alert] = "Movie #{@movie.title} could not be created: " +
11        @movie.errors.full_messages.join(",")
12      render 'new'
13    end
14  end
15  def edit
16    @movie = Movie.find params[:id]
17  end
18  def update
19    @movie = Movie.find params[:id]
20    if (@movie.update_attributes(movie_params))
21      redirect_to movie_path(@movie), :notice => "#{@movie.title} updated."
22    else
23      flash[:alert] = "#{@movie.title} could not be updated: " +
24        @movie.errors.full_messages.join(",")
25      render 'edit'
26    end
27  end
28  def destroy
29    @movie = Movie.find(params[:id])
30    @movie.destroy
31    redirect_to movies_path, :notice => "#{@movie.title} deleted."
32  end
33  private
34  def movie_params
35    params.require(:movie)
36    params[:movie].permit(:title,:rating,:release_date)
37  end
38 end

```

Figure 4.9: Putting together the main ideas of this section, here is a simple controller that handles the CRUD actions for the `Movie` model. The `create` and `update` actions make use of the fact that Rails form helpers arrange to populate `params['movies']` with a hash of attribute values ready to pass to Active Record, but the private `movie_params` method tells Rails which `params` are required and which are “safe” to pass along to Active Record.

and `flash[:alert]` is used for messages about things going wrong. Using the `flash` in conjunction with a redirect is so common that the Rails `redirect_to` method provides a special syntax for it, as Figure 4.9 shows. In fact, the `flash` is just a special case of the more general `session[]`, whose contents persist “forever” across requests from the same browser (until you clear it out manually).

But which view(s) should attempt to display the contents of the `flash`? In this example, we chose to redirect the user to the movies listing, so perhaps we should add code to the Index view to display the message. But in the future we might decide to redirect the user someplace else instead, and in any case, the idea of displaying a confirmation message or warning message is so common that it makes sense to factor it out rather than putting it into one specific view.

Recall that `app/views/layouts/application.html.erb` is the template used to “wrap” all views by default. This is a good candidate for displaying `flash` messages since any pending messages will be displayed no matter what view is rendered, as lines 11–15 of Figure 4.6 show.

Summary

- When creating a form, you specify the controller action that will receive the form submission by passing `form_tag` the appropriate RESTful URI and HTTP method (as displayed by `rake routes`). It's convenient, but not required, to use RESTful URI helpers like `movies_path` and `edit_movie_path` rather than creating the URIs manually.
- When the form is submitted, the controller action can inspect `params[]`, whose keys are the form field names and whose values are the user-supplied contents of the fields. If you use Rails form helpers, the field names are chosen such that `params[:model]` is a hash whose keys and values are the user-supplied values for an instance of `model`.
- When creating or updating a model object, for user friendliness it's common to simply `redirect_to` a view such as `index`, rather than rendering a dedicated view.
- When finishing a controller action using a redirect, `flash[:notice]` or `flash[:alert]` can be set to a message that will persist until the next request. It's conventional to modify the application layout to display such messages, so they get displayed no matter where the redirect points.

■ Elaboration: How does form submission using PUT work?

What exactly happens when you specify `:method=>:put` in the options to `form_tag`? Rails actually arranges to insert a “hidden field” in your form whose value serves as a cue to the routing and dispatching system that the form “should have been” submitted via HTTP PUT rather than POST. The dispatching system then modifies the HTTP request object seen by the routing system to make it appear that PUT was used, and removes the hidden form field from the actual form contents the controller will see. At that point, the Rails routing subsystem can do its job and route the request as if PUT had been used. This scheme may seem convoluted, but it allows the same set of routes and actions to be used to handle both user-generated requests (from a browser) and programmatic RESTful requests made to an API in a service-oriented architecture.

Self-Check 4.6.1. *Why does the form for creating a new movie submit to the `create` method rather than the `new` method?*

- ◊ Creating a new record requires two interactions. The first one, `new`, loads the form. The second one, `create`, causes the actual creation of the new record based on values in the filled-in form. ■

Self-Check 4.6.2. *Why must every controller action either render a view or perform a redirect?*

- ◊ HTTP is a request-reply protocol, so every action must generate a reply. One kind of reply is a view (Web page) but another kind is a redirect, which instructs the browser to issue a new request to a different URI. ■

Self-Check 4.6.3. *Why does it make no sense to have both a render and a redirect (or two renders, or two redirects) along the same code path in a controller action?*

- ◊ Each request needs exactly one reply. Render and redirect are two different ways to reply to a request. ■

Self-Check 4.6.4. In line 2 of Figure 4.7, what would be the effect of changing `:method=>:post` to `:method=>:get` and why?

- ◊ The form submission would result in listing all movies rather than creating a new movie. The reason is that a route requires both a URI and a method: The `movies_path` helper with the GET method would route to the `index` action, whereas the `movies_path` helper with the POST method routes to the `create` action. ■

Self-Check 4.6.5. Given that submitting the form shown in Figure 4.7 will create a new movie, why is the view called `new.html.erb` rather than `create.html.erb`?

- ◊ A RESTful route and its view should name the resource being requested. In this case, the resource requested when the user *loads* this form is the form itself, that is, the ability to create a new movie; hence `new` is an appropriate name for this resource. The resource requested when the user *submits* the form, named by the route specified for form submission on line 3 of the figure, is the actual creation of the new movie. ■

4.7 CHIPS: Hangperson on Rails



CHIPS 4.7: Hangperson on Rails

<https://github.com/saasbook/hw-rails-hangperson>

You're already familiar with the Hangperson game logic and how the Sinatra framework supports the SaaS version of the app. In this assignment, we give you the complete code for a Rails version of the same app, using the unmodified game logic, so you can readily understand the differences between how Rails and Sinatra support SaaS apps.

4.8 Debugging: When Things Go Wrong

Debugging is an annoying but invaluable skill. The amazing sophistication of today's software stacks makes it possible to be highly productive, but with so many "moving parts," it also means that things inevitably go wrong, especially when learning new languages and tools. Errors might happen because you mistyped something, because of a change in your environment or configuration, or any number of other reasons.

The best way to debug, of course, is to stop a bug in its tracks and fix it before it manifests. Two fundamental suggestions can help. First, use good tools, including a text editor that supports automatic indentation and syntax highlighting. Syntax errors such as incomplete block structure, missing quotation marks, and so on become easy to catch. If your editor isn't so equipped, you can either write your code on stone tablets, or switch to a more productive modern editor.

Second, use good colleagues! Many, many students taught by your authors report that they saved huge amounts of time by always pair programming (Section 2.2), because their partner caught a "silly" bug before it became a work stopper. If you're not pairing but you're in an "open seating" configuration, or have instant messaging enabled using a tool such as Slack, put the message out there. Try explaining your code line by line to a colleague, or if

Test-driven development (Chapter 8) requires the same skills as debugging but can be much more efficient at preventing bugs before they are hard to track down, and leaves you with better-designed code besides.



no colleague is available, to a pet or even an inanimate object. The act of trying to explain in detail may lead you to discover the flaw in your own reasoning.

But neither good tools nor good colleagues can substitute for your own thinking and debugging skills—after all, it's your code! Debugging usually involves a gradient of activities you can think of as a rising TIDE:

1. Trace the source of the error as closely as possible by really examining the error message(s) and log file(s) closely.
2. Instrument the code near the error, such as by inserting statements to print intermediate values of variables, to see where things go off the rails (so to speak).
3. Debug interactively using the facilities provided in your language or framework, such as the byebug gem for Ruby, so you can inspect and modify application state around the bug.
4. Explore question boards such as StackOverflow for similar bugs, and if all else fails, post a question there and ask for help.

We consider each of these in turn.

Trace. If the bug causes your app to crash or report an error message, take the time to really read the error message. Ruby's error messages can look disconcertingly long, but a long error message is your friend because it gives the **backtrace** showing not only the method where the error occurred, but also its caller, its caller's caller, and so on. Don't throw up your hands when you see a long error message; use the information to understand both the proximate cause of the error (the problem that "stopped the show") and the possible paths towards the root cause of the error. What is the last line of *your* code implicated in the backtrace? What happened before then?

This step can be challenging if the bug occurs in code that you blindly cut-and-pasted with no understanding of how it works or what assumptions it makes.

For example, a particularly common proximate cause of Ruby errors is **Undefined method 'foobar' for nil:NilClass**. (`NilClass` is a special class whose only instance is the constant `nil`.) This error occurs when some computation fails and returns `nil` instead of the object you expected, but you forgot to check for this error and subsequently tried to call a method on what you assumed was a valid object. If the computation occurred in another method "upstream," the backtrace can help you figure out where. In SaaS apps, this confusion can be compounded if the failed computation happens in the controller action but the invalid object is referenced in the view, as in this example:

<https://gist.github.com/ca1c2cb80a75a77a2d64adc1e059bb71>

```

1 # in controller action:
2 def show
3   @movie = Movie.where(:id => params[:id]) # what if this movie not in DB?
4   # BUG: we should check @movie for validity here!
5 end
6
7 # later, in the view - this will fail since @movie is nil
8 <h1> <%= @movie.title %> </h1>

```

Rubber duck debugging comes from an anecdote in the book *The Pragmatic Programmer: From Journeyman to Master* (Hunt and Thomas 1999) in which a programmer would debug their code by forcing themselves to explain it to a rubber duck they carried around.



An amusing perspective on the perils of blind "shotgun problem solving" is the Jargon File's hacker koan "Tom Knight and the Lisp Machine."⁹

If you're not sure you even understand the error message, use a search engine to look up key words or key phrases in the error message. You can also search sites like StackOverflow¹⁰, which specialize in helping out developers and allow you to vote for the most helpful answers to particular questions so that they eventually percolate to the top of the answer list.

n[ext]	execute next line
s[tep]	steps into blocks or methods
f[inish]	finish current method call and return
ps <i>expr</i>	print and evaluate <i>expr</i> , which can be anything that's in scope within the current stack frame, and can be used to set variables that are in scope, as in eval x=5
up	go up the call stack, to caller's stack frame
down	go down the call stack, to callee's stack frame
where	display where you are in the call stack
b[reak] <i>file:num</i>	set a breakpoint at line <i>num</i> of <i>file</i> (current file if <i>file</i> : omitted)
b <i>method</i>	set a breakpoint when <i>method</i> called
c[ontinue]	continue execution until next breakpoint
q[uit]	quit program

Figure 4.10: Command summary of the interactive Ruby debugger, byebug.

Lastly, don't forget that the log file `development.log` (or `production.log` in the production environment, or `test.log` during a testing run) contains information such as what specific HTTP request was received, what controller action was invoked, whether the action succeeded or failed or redirected, what database queries if any were performed, and so on. With so many moving parts, there is a lot of "invisible" machinery that gets invoked before your controller code is even reached, and problems that happen there can be hard to find.

Instrument. **Instrumentation** consists of extra statements you insert to record values of important variables at various points during program execution. There are various places you can instrument a Rails SaaS app:

- Display a detailed description of an object in a view. For example, try inserting `<%= debug(@movie)%>` or `<%= @movie.inspect%>` in any view to insert the result of the corresponding Ruby expression into the view itself.
- “Stop the show” inside a controller method by raising an exception whose message is a representation of the value you want to inspect, for example, `raise params.inspect` to see the detailed value of the `params` hash inside a controller method. Rails will display the exception message as the Web page resulting from the request.
- Use `Rails.logger.debug(message)` in a model or controller to emit *message* to the log.

Debug interactively. The `byebug` gem is already installed via the default Rails Gemfile. To use the debugger in a Rails app, insert the statement `byebug` at the point in your code where you want to stop the program. When you hit that statement, the terminal window where you started the server will give you a debugger prompt.

Explore question boards. Many bugs are not unique to a specific app but have happened to others in the past. Before you think about posting a question to a board such as StackOverflow, remember that it can take hours or days to get a response (if you get one), so 15 minutes spent carefully searching previous posts could save you a lot of time.

If that approach fails and you decide to post a question, remember that everyone else on StackOverflow is as busy as you, so write your question in a way that makes it easy for a knowledgeable person to find and answer. Choose your keywords specifically and carefully:

printf debugging is an old name for this technique, from the C library function that prints a string on the terminal.

`rails` is extremely common, but the conjunction `rails controller redirect` narrows the topic down significantly. In the question post itself, be as specific as possible about what went wrong, what your environment is, and how to reproduce the problem. The ideal question post will express as concisely and specifically as possible (a) what your code is supposed to do, (b) the result you expected, (c) the result you actually observe. Here are some examples and anti-examples:

- **Vague:** “The `sinatra` gem doesn’t work on my system.” There’s not enough information here for anyone to help you.
- **Better, but annoying:** “The `sinatra` gem doesn’t work on my system. Attached is the 85-line error message.” Other developers are just as busy as you and probably won’t take the time to extract relevant facts from a long trace.
- **Best:** Look at the actual transcript¹¹ of this question on StackOverflow. At 6:02pm, the developer provided specific information, such as the name and version of their operating system, the specific commands they successfully ran, and the unexpected error that resulted. Other helpful voices chimed in asking for specific additional information, and by 7:10pm, two of the answers had identified the problem.

While it’s impressive that this developer got their answer in just over an hour, it means they also lost an hour of coding time, which is why you should post a question only after you’ve exhausted the other alternatives.

Summary

- Use a language-aware editor with syntax highlighting and automatic indentation to help find syntax errors.
- Instrument your app by inserting the output of `debug` or `inspect` into views, or by making them the argument of `raise`, which will cause a runtime exception that will display `message` as a Web page.
- To debug using the interactive debugger, make sure your app’s Gemfile includes `byebug` and place the statement `byebug` at the point in your code where you want to break.

Self-Check 4.8.1. *Why can’t you just use `print` or `puts` to display messages to help debug your SaaS app?*

- ◊ Unlike command-line apps, SaaS apps aren’t attached to a terminal window, so there’s no obvious place for the output of a `print` statement to go. ■

Self-Check 4.8.2. *Which of the debugging methods described in this section are appropriate for collecting instrumentation or diagnostic information once your app is deployed and in production?*

- ◊ Only the `logger` method is appropriate, since the other two methods (“stopping the show” in a controller or inserting diagnostic information into views) would interfere with the usage of real customers if used on a production app. ■

4.9 CHIPS: Hello Rails



CHIPS 4.9: Hello Rails

<https://github.com/saasbook/hw-hello-rails>

Build and deploy a simple app based on Rails.

4.10 Fallacies and Pitfalls



Pitfall: Route with a matching view but no controller action.

If you specify a route (say `GET /movies`) that lacks a corresponding controller action (`MoviesController#index` in this case) but *does* have a matching view (`app/views/movies/index.html.erb` in this case), *Rails will behave as if the controller action exists but has an empty method definition*. That is, it will render the view, but any instance variables used in the view will be `nil`. If the matching view template *does not* exist, Rails will signal an error when the route is used. This inconsistency is non-intuitive, so be careful when you set up a route that you immediately add *both* the controller action and the view template, or else add neither.



Pitfall: Modifying the database manually rather than using migrations, or managing gems manually rather than using Bundler.

Especially if you've come from other SaaS frameworks, it may be tempting to use the SQLite command line or a GUI database console to manually add or change database tables or to install libraries. **Don't do it.** If you modify the database manually, you'll have no consistent way to reproduce these steps in the future (for example at deployment time) and no way to roll back the changes in an orderly way. Also, since migrations and Gemfiles are just files that become part of your project, you can keep them under version control and see the entire history of your changes.



Pitfall: Bulky controller actions or views.

Because controller actions are the first place in your app's code to be called when a user request arrives, it's remarkably easy for them to slowly absorb logic that really belongs somewhere else, such as the model or a service object. Similarly, it's easy for code to creep into views—most commonly, a view may find itself calling a model method such as `Movie.all`, rather than having the controller method set up a variable such as `@movies=Movie.all` and having the view just use `@movies`. Besides violating MVC, coupling views to models can interfere with caching, which we'll explore in Chapter 5. The view should focus on displaying content and facilitating user input, and the controller should focus on mediating between the view and the model and set up any necessary variables to keep code from leaking into the view.



Pitfall: Overstuffing the session[] hash.

You should minimize what you put in the `session[]` for two reasons. First, with the default Rails configuration, the session is packed into a cookie (Section 3.2), which the HTTP

specification limits to 4 KiB in size. Second, and more importantly, bulky sessions are a warning that your app's actions aren't very self-contained and therefore probably not RESTful, and may be difficult to use as part of a Service-Oriented Architecture. Although nothing stops you from assigning arbitrary objects to the session, you should keep just the `ids` of necessary objects in the session and keep the objects themselves in model tables in the database.



Pitfall: Adopting a framework without a good reason.

Software breeds complexity. The more numerous and complex the libraries on which your app relies, the more you'll have to learn to write the app properly; the more resource-intensive (thus possibly slower) the app will run; the harder it will be to maintain; and perhaps most importantly, the more exposure surface it will have for being attacked. Use the simplest framework that will solve most of your problems.

The answer to the question “Should my app use framework X” should be NO by default, unless you can come up with good technical reasons why it should. Some examples of good reasons: the framework encapsulates a solution to a hard technical problem, as Stripe does for credit card processing; the framework provides low-level machinery to instantiate an architecture that the app follows closely, as Rails does for Model–View–Controller; or there are technological constraints requiring you to use the framework (“all our apps are built on framework X and that’s all we know how to maintain”).

Bad reasons include: “It’s the hot new framework”; “All the popular apps use it”; “I will be more marketable if I learn this framework, and this app is a good excuse to learn it”. The last one is doubly false: the most sought-after software engineers are those who can *learn* new frameworks quickly, and at any rate, a new customer-facing app is not an excuse to learn a framework but an artifact that your customers will rely on, and that whoever comes after you will have to maintain (if you’re lucky; otherwise it will just die).

4.11 Concluding Remarks: Rails as a Service Framework

The introduction to Rails in this chapter may seem to introduce a lot of very general machinery to handle a fairly simple and specific task: implementing a Web-based UI to CRUD actions. However, we will see in Chapter 5 that this solid groundwork will position us to appreciate the more advanced mechanisms that will let you truly DRY out and beautify your Rails apps. For example, adapting your app into an API-based service or microservice is much easier if the app mostly follows the structure of a collection of resources supporting the CRUD actions and perhaps some others. Controller actions such as `index` and `show` would need few changes to emit a JSON representation of an object instead of displaying an HTML representation. The new action wouldn’t need to be adapted at all: it’s only there so that the human user can fill in the values that will be used for `create`, so an API wouldn’t need to provide any version of it. Similarly, in an API-based service, `create` and `update` could simply return a JSON representation of the created or updated object, or even the created object’s ID, rather than redirecting back to some other view.

Thus, as with many tools we will use in this book, the initial learning curve to do a simple task may seem a bit steep, but you will quickly reap the rewards by using this strong foundation to add new functionality and features quickly and concisely.

That said, Rails is an opinionated framework, and over the years it has been critiqued for



being too large, too complex, and employing too much “magic” to make things work, such as the trick of how instance variables set by controller actions are available to the view, despite the view and controller not sharing any ancestor class and not really being “instances” of anything in any meaningful way. Over the years, modules have been extracted from Rails, more attention has been given to making controllers API-friendly, and many parts of ActiveRecord have been extracted into separate modules that can be mixed in or used without ActiveRecord and even without Rails. In fact, Rails itself was originally extracted from a standalone app written by the consulting group 37signals.

To understand the architecture of a software system is to understand its organizing principles. In Chapter 3 we identified the client-server pattern and the REST API pattern as dominant characteristics of SaaS. In this chapter we identified Model–View–Controller and Active Record as architectural patterns within the boundary of a software artifact. Such patterns are a powerful way to manage complexity in large software systems; we will have much more to say about them in Chapter 11. For now, we observe that by choosing to build a SaaS app, we have predetermined the use of some patterns and excluded others. By choosing to use Web standards, we have predetermined a client-server system; by choosing cloud computing, we have predetermined the three-tier architecture (which Chapter 12 discusses further) to permit horizontal scaling. Model–View–Controller is not predetermined, but we choose it because it is a good fit for Web apps that are view-centric and have historically relied on a persistence tier, notwithstanding other possible patterns such as those in Figure 4.1. REST is not predetermined, but we choose it because it simplifies integration into a Service-Oriented Architecture and can be readily applied to the CRUD operations, which are so common in MVC apps. Active Record is perhaps more controversial—as we will see in Sections 5.8 and 12.7, its powerful facilities simplify apps considerably, but misusing those facilities can lead to scalability and performance problems that are less likely to occur with simpler persistence models.

A good framework should also be closely wedded to the language in which it’s implemented: the framework should not only provide a well-defined abstraction for the application’s architecture, but also use the language’s features to support those abstractions. For example, Rails uses Ruby’s introspection and metaprogramming to provide an elegant implementation of both the Model–View–Controller application architecture and the Active Record design pattern for storing model data. And not all server-side or client-side stacks are application frameworks: while Node.js provides some low-level abstractions for **event-driven programming** (about which we will have much more to say in Chapter 6), it provides no abstractions for any particular architectural pattern on which one might want to base an application.

If we were building a SaaS app in 1995, none of the above would have been obvious because practitioners had not accumulated enough examples of successful SaaS apps to “extract” successful patterns into frameworks like Rails, software components like Apache, and middleware like Rack. By following the successful footsteps of software architects before us, we can take advantage of their ability to *separate the things that change from those that stay the same* across many examples of SaaS and provide tools, frameworks, and design principles that support building things this way. As we mentioned earlier, this separation is key to enabling reuse.

A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999. ISBN 020161622X.

Notes

- ¹<https://www.khanacademy.org/computing/computer-programming/sql>
- ²http://en.wikipedia.org/wiki/HTML_sanitization
- ³https://guides.rubyonrails.org/form_helpers.html
- ⁴https://guides.rubyonrails.org/form_helpers.html
- ⁵<https://apidock.com/rails>
- ⁶<http://homakov.blogspot.com/2012/03/how-to.html>
- ⁷<http://api.rubyonrails.org/classes/ActionController/Parameters.html>
- ⁸<http://api.rubyonrails.org/classes/ActionController/Parameters.html>
- ⁹<http://catb.org/jargon/html/koans.html>
- ¹⁰<http://stackoverflow.com>
- ¹¹<http://stackoverflow.com/questions/2945228/i-see-gem-in-gem-list-but-have-no-such-file-to-load>

5

SaaS Framework: Advanced Programming Abstractions for SaaS

Kristen Nygaard (left, 1926–2002) and Ole-Johan Dahl (right, 1931–2002) shared the 2001 Turing Award for inventing fundamental OO concepts including objects, classes, and inheritance, and demonstrating them in Simula, the ancestor of every object-oriented language.



Programming is understanding.

—Kristen Nygaard

5.1	DRYing Out MVC: Partials, Validations and Filters	126
5.2	Single Sign-On and Third-Party Authentication	131
5.3	CHIPS: Rails Intro	136
5.4	Associations and Foreign Keys	136
5.5	Through-Associations	140
5.6	RESTful Routes for Associations	143
5.7	CHIPS: Associations	146
5.8	Other Types of Code	147
5.9	Fallacies and Pitfalls	149
5.10	Concluding Remarks: Languages, Productivity, and Beauty	149

Prerequisites and Concepts

This chapter covers advanced features of Rails that you can use to make your code more DRY and concise, including how to reuse entire external services such as Twitter to integrate with your apps.

Prerequisites:

You should be familiar as a Web user with the “Log in using...” *single sign-on* flow supported by many apps, such as “Log in with Google” or “Log in with GitHub.”

You should have a basic understanding of join operations in relational databases, and in particular on how tables are joined using foreign keys. We review this material only briefly before showing how Rails uses it to provide flexible mechanisms for expressing associations among ActiveRecord models in a SaaS app. To brush up on this material, complete the “Relational queries in SQL” section of the Khan Academy SQL tutorial¹.

Concepts:

- Rails mechanisms such as controller filters, model lifecycle hooks, and model validations provide a limited form of *aspect-oriented programming*, which allows code about crosscutting concerns to be centralized in a single place and automatically called when needed.
- Single sign-on (SSO), such as “Log in using your GitHub account,” lets a user identify themselves to service B by their credentials on service A, without revealing those credentials to service B. Using SSO relieves your app of having to manage passwords and provides some convenience to the user, though at the expense of sacrificing some privacy.
- ActiveRecord *associations* use metaprogramming and reflection to map relationships among resources in your app, such as “belongs to” or “has many”, to queries that mirror those relationships in the app’s database. From ActiveRecord all the way through the routing system, Rails provides comprehensive facilities for managing such relationships.
- ActiveRecord *scopes* are composable “filters” you can define on your model data, enabling DRY reuse of model logic.
- Some important pieces of code in your app don’t really belong in a model, view, or controller. We introduce some other kinds of code that play important roles and suggest where to put it and how to structure it.

```
https://gist.github.com/be580b19585cd2e77d6ce7fcacfd5508
```

```

1 | <!-- ...other code from index.html.erb here... -->
2 | <div class="row bg-dark text-white">
3 |   <div class="col-6 text-center">Title and More Info</div>
4 |   <div class="col-2 text-center">Rating</div>
5 |   <div class="col-4 text-center">Release Date</div>
6 | </div>
7 | <%= render partial: 'movie', collection: @movies %>
```

```
https://gist.github.com/8e8dba742ccb8de1850456bc498f5f65
```

```

1 | <div class="row">
2 |   <div class="col-8"> <%= link_to movie.title, movie_path(movie) %> </div>
3 |   <div class="col-2"> <%= movie.rating %> </div>
4 |   <div class="col-2"> <%= movie.release_date.strftime('%F') %> </div>
5 | </div>
```

Figure 5.1: (Top) Main view that uses a partial for each row of the movies table; (Bottom) Partial containing the code to render one row. To leverage convention over configuration, we name it `_movie.html.erb`: Rails uses the filename (without the underscore) to set a *local* variable (`movie`) to each item of the `@movies` collection in turn.

5.1 DRYing Out MVC: Partials, Validations and Filters



As Section 6.7 explains, the partial is also the basic unit of view updating for JavaScript-enabled pages.



One of the core tenets of Rails is DRY—Don’t Repeat Yourself. In this section we introduce three mechanisms Rails provides to help you DRY out your code: model validations, view partials, and controller filters.

We start with views. A *partial* is Rails’ name for a reusable chunk of a view. When similar content must appear in different views, putting that content in a partial and “including” it in the separate files helps DRY out repetition. Our simple app already presents one opportunity: the Index (list all movies) view includes a chunk of HTML that is repeated for each movie in the list. We can factor out that code into a partial, and include it by reference, as Figure 5.1 shows.

Partials rely heavily on convention over configuration. Their names must begin with an underscore (we used `_movie.html.erb`) which is *absent from* the code that references the partial. A partial may be in a different directory than the view that uses it, in which case a path such as `'layouts/footer'` would cause Rails to look for `app/views/layouts/_footer.html.erb`. A partial can access all the same instance variables as the view that includes it, but partials that may be used from different views usually do not reference controller instance variables, since those may be set differently (or not at all) by different controller actions. A particularly nice use of a partial is to render a table or other collection in which all elements are the same, as Figure 5.1 demonstrates.

Partials are simple and straightforward, but the mechanisms provided by Rails for DRYing out models and controllers are more subtle and sophisticated. It’s common in SaaS apps to want to enforce certain validity constraints on a given type of model object or constraints on when certain actions can be performed. For example, when a new movie is added to RottenPotatoes, we may want to check that the title isn’t blank, that the release year is a valid date, and that the rating is one of the allowed ratings. (You may think there’s no way for the user to specify an invalid rating if they’re choosing it from a dropdown menu, but the request might be constructed by a malicious user or a **bot**.) With SaaS, you can’t trust anyone: the server must *always* check its inputs rather than trust them, or risk attack by methods we’ll see in Chapter 12.

As another example, perhaps we want to allow any user to add new movies, but only

<https://gist.github.com/89addee067f097c7167e38e2c40a555aa>

```

1 class Movie < ActiveRecord::Base
2   def self.all_ratings ; %w[G PG PG-13 R NC-17] ; end # shortcut: array of
3   strings
4   validates :title, :presence => true
5   validates :release_date, :presence => true
6   validate :released_1930_or_later # uses custom validator below
7   validates :rating, :inclusion => {:in => Movie.all_ratings},
8   :unless => :grandfathered?
9   def released_1930_or_later
10    errors.add(:release_date, 'must be 1930 or later') if
11    release_date && release_date < Date.parse('1 Jan 1930')
12  end
13  @@grandfathered_date = Date.parse('1 Nov 1968')
14  def grandfathered?
15    release_date && release_date < @@grandfathered_date
16  end
17 # try in console:
18 m = Movie.new(:title => '', :rating => 'RG', :release_date => '1929-01-01')
19 # force validation checks to be performed:
20 m.valid? # => false
21 m.errors[:title] # => ["can't be blank"]
22 m.errors[:rating] # => [] - validation skipped for grandfathered movies
23 m.errors[:release_date] # => ["must be 1930 or later"]
24 m.errors.full_messages # => ["Title can't be blank", "Release date
25 must be 1930 or later"]

```

Figure 5.2: Lines 3–5 use predefined validation behaviors in `ActiveModel::Validations::ClassMethods`³. Lines 6–15 show how you can create your own validation methods, which receive the object to be validated as an argument and add error messages describing any problems. Note that we first validate the presence of `release_date`, otherwise the comparisons in lines 10 and 14 could fail if `release_date` is nil.

allow special “admin” users to delete movies. Both examples involve specifying constraints on entities or actions, and although there might be many places in an app where such constraints should be considered, the DRY philosophy urges us to centralize them in *one* place. Rails provides two analogous facilities for doing this: validations for models and filters for controllers.

Model validations, like migrations, are expressed in a mini-DSL embedded in Ruby, as Figure 5.2 shows. Validation checks are triggered when you call the instance method `valid?` or when you try to save the model to the database (which calls `valid?` before doing so). Any validation errors are recorded in the `ActiveModel::Errors`⁴ object associated with each model; this object is returned by the instance method `errors`. As line 7 shows, validations can be conditional: the movie’s rating is validated *unless* the movie was released before the ratings system went into effect (in the USA, 1 November 1968).

We can now understand lines 10–12 and 23–25 from Figure 4.9 in the last chapter. When creating or updating a movie fails (as indicated by a falsy return value from `create` or `update_attributes`), we set `flash[:alert]` to an error message informed by the contents of the movie errors object. We then `render` (*not* redirect to) the form that brought us here, with `@movie` still holding the values the user entered the first time, so the form will be prepopulated with those values. A redirect would start an entirely new request cycle, and `@movie` would not be preserved.

In fact, validations are just a special case of a more general mechanism, Active Record lifecycle callbacks⁷, which allow you to provide methods that “intercept” a model object at various relevant points in its lifecycle. Figure 5.3 shows what callbacks are available; Figure 5.4 illustrates how to use this mechanism to “canonicalize” (standardize the format of)

Validations store error messages but do not actually raise an error; this is another example of **command-query separation**, explained at the end of Section 3.5.

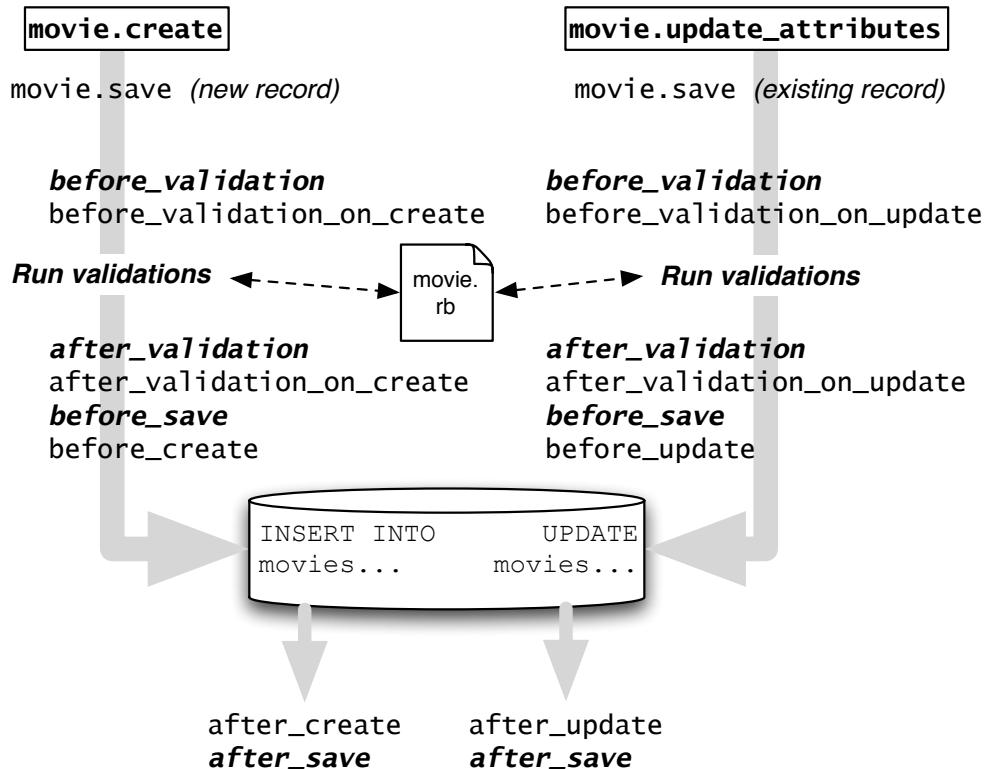


Figure 5.3: The various points at which you can “hook into” the lifecycle of an ActiveRecord model object. All ActiveRecord operations that modify the database (`update`, `create`, and so on) all eventually call `save`, so a `before_save` callback can intercept every change to the database. See this Rails Guide⁶ for additional details and examples.

<https://gist.github.com/9e961598f5b8f69c2c37dbe5c6c3a917>

```

1  class Movie < ActiveRecord::Base
2    before_save :capitalize_title
3    def capitalize_title
4      self.title = self.title.split(/\s+/).map(&:downcase).
5        map(&:capitalize).join(' ')
6    end
7  end
8  # now try in console:
9  m = Movie.create!(:title => 'STAR  wars', :release_date => '27-5-1977', :
10   rating => 'PG')
m.title # => "Star Wars"

```

Figure 5.4: This `before_save` hook capitalizes each word of a movie title, downcases the rest of the word, and compresses multiple spaces between words to a single space, turning `STAR wars` into `Star Wars`. Coincidentally, Rails’ `ActiveSupport::Inflector#titleize` provides this functionality.

<https://gist.github.com/d9d5cb0fa2f9c4343b21b18bfda7ccf1>

```

1 class ApplicationController < ActionController::Base
2   before_filter :set_current_user
3   protected # prevents method from being invoked by a route
4   def set_current_user
5     # we exploit the fact that the below query may return nil
6     @current_user ||= Moviegoer.where(:id => session[:user_id])
7     redirect_to login_path and return unless @current_user
8   end
9 end

```

Figure 5.5: If there is a logged-in user, the redirect will *not* occur, and the controller instance variable `@current_user` will be available to the action and views. Otherwise, a redirect will occur to `login_path`, which is assumed to correspond to a route that takes the user to a login page, as Section 5.2 explains. (`and` is just like `&&` but has lower precedence, thus it parses as `((redirect_to login_path) and (return)) unless...`)

certain model fields before the model is saved. We will see another use of lifecycle callbacks when we discuss the Observer design pattern in Section 11.7 and caching in Section 12.6.

Analogous to a validation is a controller filter—a method that checks whether certain conditions are true before an action is run, or sets up common conditions that many actions rely on. If the conditions are not fulfilled, the filter can choose to “stop the show” by rendering a view template or redirecting to another action. If the filter allows the action to proceed, it will be the action’s responsibility to provide a response, as usual.

As an example, an extremely common use of filters is to enforce the requirement that a user be logged in before certain actions can be performed. Assume for the moment that we have verified the identity of some user and stored her primary key (ID) in `session[:user_id]` to remember the fact that she has logged in. Figure 5.5 shows a filter that enforces that a valid user is logged in. In Section 5.2 we will show how to combine this filter with the other “moving parts” involved in dealing with logged-in users.

Filters normally apply to all actions in the controller, but as the documentation on filters states, `:only` or `:except` can be used to restrict a filter to guarding only certain actions. You can define multiple filters: they are run in the order in which they are declared. You can also define after-filters, which run after certain actions are completed, and around-filters, which contain code to run before and after, as you might do for auditing or timing.

Summary of DRYing out MVC in Rails:

- Partials allow you to reuse chunks of views across different templates, collecting common view elements in a single place.
- Validations let you collect constraints on a model in a single place. Validations are checked anytime the database is about to be modified; failing validation is one of the ways that non-dangerous `save` and `update_attributes` can fail.
- The `errors` field of a model, an `ActiveRecord::Errors` object, records errors that occurred during validation, but it is up to you to take action if `model.errors.empty?` is not true.
- Controller filters let you collect conditions affecting many controller actions in a single place, or set up instance variables used by many actions in a single place, by defining a method that runs before those actions.

■ Elaboration: Aspect-oriented programming

Aspect-oriented programming (AOP) is a programming methodology for DRYing out code by separating *crosscutting concerns* such as model validations and controller filters from the main code of the actions to which the concerns apply. In our case, we specify model validations declaratively in one place, rather than invoking them explicitly at each **join point** in the code where we'd want to perform a validity check. A set of join points is collectively called a **pointcut**, and the code to be inserted at each join point (such as a validation in our example) is called **advice**. Rather than supporting fully general AOP, which would allow you to specify arbitrary pointcuts along with what advice applies to each, Rails defines pointcuts for model validations and controller filters.

A critique of AOP is that the source code can no longer be read in linear order. For example, when a before-filter prevents a controller action from proceeding, the problem can be hard to track down, especially for someone unfamiliar with Rails who doesn't realize the filter method isn't even being called explicitly but is an advice method triggered by a particular join point. A response to the critique is that if AOP is applied sparingly and tastefully, and all developers understand and agree on the pointcuts, it can improve DRYness and modularity. Validations and filters are the Rails designers' attempt to identify this beneficial middle ground.

Self-Check 5.1.1. *Why didn't the Rails designers choose to trigger validation when you first instantiate a movie using `Movie.new`, rather than waiting until you try to persist the object?*

◊ As you're filling in the attributes of the new object, it might be in a temporarily invalid state, so triggering validation at that time might make it difficult to manipulate the object. Persisting the object tells Rails "I believe this object is ready to be saved." ■

Self-Check 5.1.2. *In line 5 of Figure 5.2, why can't we write `validate released_1930_or_later`, that is, why must the argument to `validate` be either a symbol or a string?*

◊ If the argument is just the "bare" name of the method, Ruby will try to evaluate it at the moment it executes `validate`, which isn't what we want—we want `released_1930_or_later` to be called at the time any validation is to occur. ■

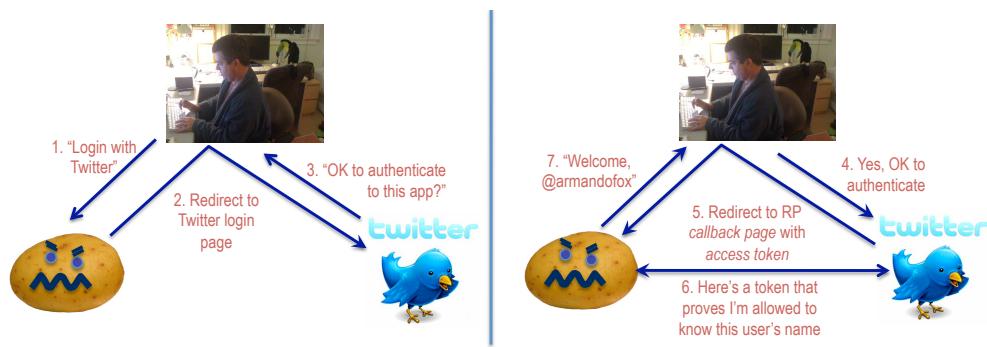


Figure 5.6: Third-party authentication enables SSO by allowing a SaaS app to request that the user authenticate himself via a third-party provider. Once the user has done so, the provider sends a token to the requesting app proving that the user authenticated themselves correctly and possibly encoding additional privileges the user grants to the requesting app. The flow shown is a simplified version of OAuth, an evolving (and mildly controversial) open standard for authentication and authorization used by Twitter, Facebook, Microsoft, Google, Netflix, and many others. Twitter logo and image copyright 2012 Twitter Inc., used for instructional purposes only.

5.2 Single Sign-On and Third-Party Authentication

One way to be more DRY and productive is to avoid implementing functionality that you can instead reuse from other services. One example of this today is **authentication**—the process by which an entity or **principal** proves that it is who it claims to be. In SaaS, end users and servers are two common types of principals that may need to authenticate themselves. Typically, a user proves their identity by supplying a username and password that (presumably) nobody else knows, and a server proves its identity with a **server certificate** (discussed in Chapter 12) whose **integrity** can be verified using cryptography.

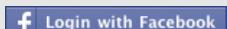
In the early days of SaaS, users had to establish separate usernames and passwords for each site. Today, an increasingly common scenario is **single sign-on** (SSO), in which the credentials established for one site (the *provider*) can be used to sign in to other sites that are administratively unrelated to it. Clearly, SSO is central to the usefulness of service-oriented architecture: It would be difficult for services to work together on your behalf if each had its own separate authentication scheme. Given the prevalence and increasing importance of SSO, our view is that new SaaS apps should use it rather than “rolling their own” authentication.

However, SSO presents the dilemma that while you may be happy to use your credentials on site A to login to site B, you usually don’t want to reveal those credentials to site B. (Imagine that site A is your financial institution and site B is a foreign company from whom you want to buy something.) Figure 5.6 shows how *third-party authentication* solves this problem using RottenPotatoes and Twitter as an example. First, the app requesting authentication (RottenPotatoes) creates a request to an authentication provider on which the user already has an account, in this case Twitter. The request often includes information about what privileges the app wants on the provider, for example, to be able to tweet as this user or learn who the user’s followers are.

A typical SSO process is illustrated by the OAuth⁸ protocol, which begins with a link or button the user must click. That link takes the user to a login page served securely by *the provider*. The user is then given the chance to login to the provider and decide what privileges to grant the requesting app. Critically, this interaction takes place entirely between the user and the provider: the requesting app has no access to any part of this interaction.

Authorization refers to whether a principal is allowed to do something. Although separate from authentication, the two are often conflated because many standards handle both.

Facebook was an early example of SSO.



```
https://gist.github.com/533907757ee761982af7b3b00b3f517f
```

```
1 | rails generate model Moviegoer name:string provider:string uid:string
```

```
https://gist.github.com/897d2dace6a3ccb68859778f5c4ba032
```

```
1 | # Edit app/models/moviegoer.rb to look like this:
2 | class Moviegoer < ActiveRecord::Base
3 |   def self.create_with_omniauth(auth)
4 |     Moviegoer.create!(
5 |       :provider => auth["provider"],
6 |       :uid => auth["uid"],
7 |       :name => auth["info"]["name"])
8 |     end
9 |   end
```

Figure 5.7: Top (a): Type this command in a terminal to create a `moviegoers` model and migration, and run `rake db:migrate` to apply the migration. Bottom (b): Then edit the generated `app/models/moviegoer.rb` file to match this code, which the text explains.

Once authentication succeeds, the provider generates an HTTP POST to a particular route on the requesting app. This post request contains an **access token**—a string created using cryptographic techniques that can be passed back to the provider later, allowing the provider to verify that the token could only have been created as the result of a successful login process. At this point, the requesting app is able to do two things:

1. It can believe that the user has proven her identity to the provider, and optionally record the provider’s persistent user-ID (uid) for that user, usually provided as part of the access token. For example, Armando Fox’s uid on Twitter happens to be 318094297, though this information isn’t useful unless accompanied by an access token granting the right to obtain information about that uid.
2. It can use the token to request further information about the user from the provider, depending on what specific privileges were granted along with successful authentication. For example, a token from Facebook might indicate that the user gave permission for the app to learn who his friends are, but denied permission for the app to post on his Facebook wall.

Happily, adding third-party authentication to Rails apps is straightforward. Of course, before we can enable a user to log in, we need to be able to represent users! So before continuing, create a basic model and migration following the instructions in Figure 5.7.

There are three aspects to managing third-party authentication in SaaS:

- 
1. How to authenticate the user via a third party authentication provider (“auth provider”) such as Google or GitHub
 2. How to remember that the user has logged in successfully
 3. How to link the user’s ID in our own app with that provider’s ID, so that we can recognize this user in the future

By far the simplest way to accomplish the first task in Rails is to use the excellent OmniAuth⁹ gem, which provides a uniform API to many different SSO providers, abstracting away the entire process in Figure 5.6. No matter which provider is used, OmniAuth arranges to send the user to the provider’s login page, handles the providers’ callbacks for successful or

<https://gist.github.com/e47a5afb49af3cef6cf24438e521c451>

```

1 get 'auth/:provider/callback' => 'sessions#create'
2 get 'auth/failure' => 'sessions#failure'
3 get 'auth/twitter', :as => 'login'
4 post 'logout' => 'sessions#destroy'
```

<https://gist.github.com/11f56e2f654393f34eb87009e3a25c2c>

```

1 class SessionsController < ApplicationController
2   # login & logout actions should not require user to be logged in
3   skip_before_filter :set_current_user
4   def create
5     auth = request.env["omniauth.auth"]
6     user =
7       Moviegoer.where(provider: auth["provider"], uid: auth["uid"]) ||
8       Moviegoer.create_with_omniauth(auth)
9     session[:user_id] = user.id
10    redirect_to movies_path
11  end
12  def destroy
13    session.delete(:user_id)
14    flash[:notice] = 'Logged out successfully.'
15    redirect_to movies_path
16  end
17 end
```

<https://gist.github.com/e309ab308e1de682960b2a29e7f13c26>

```

1 # Replace API_KEY and API_SECRET with the values you got from Twitter
2 Rails.application.config.middleware.use OmniAuth::Builder do
3   provider :twitter, "API_KEY", "API_SECRET"
4 end
```

Figure 5.8: (a) Top: If auth succeeds (line 1), OmniAuth will generate a GET to the `create` action in `SessionsController`. Line 3 makes the route helper `login_path` route to GET `auth/twitter`, which OmniAuth will redirect to Twitter's login page, so that line 7 in Figure 5.5 will work correctly. (b) Middle: Line 3 skips the `before_filter` that we added to `ApplicationController` in Figure 5.5. Upon successful login, the `create` action remembers the user's ID in the session until the `destroy` action is called to forget it. (c) Bottom: Files in `config/initializers` are loaded before the app starts. This one, `omniauth.rb`, specifies the API keys to use for Twitter SSO.

failed authentication, and finally generates GET requests to well-known routes in your app to handle these cases. To use OmniAuth, you install both the OmniAuth gem and the necessary additional gems for each auth provider *strategy*.

Figure 5.8 shows the changes necessary to your routes, controllers, and configuration to use OmniAuth. Most auth providers require you to register any apps that will use their site for authentication, so in this example you would need to create a Twitter developer account, which will assign you an API key and an API secret that you specify in `config/initializers/omniauth.rb`, as Figure 5.8(c) shows. The second aspect of handling authentication is keeping track of whether the current user has been authenticated. You may have already guessed that this information can be stored in the `session[]`. However, we should keep session management separate from the other concerns of the app, since the session may not be relevant if our app is used in a service-oriented architecture setting. To that end, Figure 5.8(b) shows how we can “create” a session when a user successfully authenticates (lines 4–11) and “destroy” it when they log out (lines 12–16). The “scare quotes” are there because the only thing actually being created or destroyed is the value of `session[:user_id]`, which is set to the primary key of the logged-in user during the session and `nil` at other times. Figure 5.5 shows how this check is abstracted by a `before_filter` in `ApplicationController` (which will be inherited by all controllers) that sets `@current_user` accordingly, so that controller methods or views can just look at `@current_user` without being coupled to the details of how the user was authenticated.

The third aspect is linking our own representation of a user’s identity—that is, her primary key in the `moviegoers` table—with the auth provider’s representation, such as the `uid` in the case of Twitter. Since we may want to expand which auth providers our customers can use in the future, the migration in Figure 5.7(a) that creates the `Moviegoer` model specifies both a `uid` field and a `provider` field. What happens the very first time Alice logs into RottenPotatoes with her Twitter ID? The query in line 7 of the sessions controller (Figure 5.8(b)) will return `nil`, so `Moviegoer.create_with_omniauth` (Figure 5.7(b), lines 3–8) will be called to create a new record for this user. Note that “Alice as authenticated by Twitter” would therefore be a different user from our point of view than “Alice as authenticated by Facebook,” because we have no way of knowing that those represent the same person. That’s why some sites that support multiple third-party auth providers give users a way to “link” two accounts to indicate that they identify the same person.

This may seem like a lot of moving parts, but compared to accomplishing the same task without an abstraction such as OmniAuth, this is very clean code: we added fewer than two dozen lines, and by incorporating more OmniAuth strategies, we could support additional third-party auth providers with essentially no new work. Screencast 5.2.1 shows the user experience associated with this code.



Screencast 5.2.1: Logging into RottenPotatoes with Twitter.

<http://youtu.be/R3S3efTtwmI>

This version of RottenPotatoes, modified to use the OmniAuth gem as described in the text, allows moviegoers to login using their existing Twitter IDs.

However, we must be careful to avoid creating a security vulnerability. What if a malicious attacker crafts a form submission that tries to modify `params[:moviegoer][:uid]` or `params[:moviegoer][:provider]`—fields that should only be modified by the authentication logic—by posting **hidden form fields** named `moviegoer[uid]` and so on? Section 4.4 explained how the “strong parameters” feature of Rails can be used to block

assignment of model attributes that regular users shouldn't be able to set. While it's fine for the `create_with_omniauth` method to create a moviegoer with the appropriate uid, a regular moviegoer should not be able to set their own uid since it would allow them to impersonate being logged in! To ensure this can't happen, we must make sure uid does not appear in any calls to `params.permit` or `params.require` in the Moviegoers controller.

Summary

- Single sign-on refers to an end-user experience in which a single set of credentials (such as their Google or Facebook username and password) will sign them in to a variety of different services.
- Third-party authentication using standards such as **OAuth** is one way to achieve single-sign on: the requesting app can verify the identity of the user via an authentication provider, without the user revealing her credentials to the requesting app.
- The cleanest way to factor out authentication in Rails apps is to abstract the concept of a session. When a user successfully authenticates (perhaps using a framework such as OmniAuth¹⁰), a session is created by storing the authenticated user's id (primary key) in the `session[]`. When they sign out, the session is destroyed by deleting that information from the `session[]`.
- Use Rails' strong parameters to ensure that model attributes that are "sensitive" and should be excluded from mass assignment do not appear in `params.require` or `params.permit` calls in your controllers.

■ Elaboration: SSO side effects

In some cases, using SSO enables other features as well; for example, Facebook Connect enables sites to take advantage of Facebook's social network, so that (for example) Marsalis can see which New York Times articles his friends have been reading once he authenticates himself to the New York Times using Facebook. While these appealing features further strengthen the case for using SSO rather than "rolling your own" authentication, they are separate from the basic concept of SSO, on which this discussion focuses.

Self-Check 5.2.1. *Briefly describe how RottenPotatoes could let you log in with your Twitter ID without you having to reveal your Twitter password to RottenPotatoes.*

- ◊ RottenPotatoes redirects you to a page hosted by Twitter where you log in as usual. The redirect includes a URL to which Twitter posts back a message confirming that you've authenticated yourself and specifying what actions RottenPotatoes may take on your behalf as a Twitter user. ■

Self-Check 5.2.2. *True or false: If you log in to RottenPotatoes using your Twitter ID, RottenPotatoes becomes capable of tweeting using your Twitter ID.*

- ◊ False: authentication is separate from permissions. Most third-party authentication providers, including Twitter, allow the requesting app to ask for permission to do specific things, and leave it up to the user to decide whether to allow it. ■



Figure 5.9: Each end of an association is labeled with its *cardinality*, or the number of entities participating in that “side” of the association, with an asterisk meaning “zero or more”. In the figure, each Review belongs to a single Moviegoer and a single Movie, and a Review without a Moviegoer or without a Movie is not allowed. (A cardinality notation of “0..1” rather than “1” would allow “orphaned” reviews.)

5.3 CHIPS: Rails Intro



CHIPS 5.3: Rails Intro

<https://github.com/saasbook/hw-rails-intro>

Starting with RottenPotatoes, a Rails app we provide for keeping track of movie info and reviews, we will add some simple features to sort and filter the list of movies and to allow a moviegoer to establish an account and log in using single sign-on.

5.4 Associations and Foreign Keys

An *association* is a logical relationship between two types of entities in a software architecture. For example, the previous CHIPS added a Moviegoer class to RottenPotatoes; we could now add a Review class to allow a moviegoer to write reviews of their favorite movies. Because each review is about exactly one movie, but a single movie can have many reviews, we say that there is a one-to-many association from movies to reviews. Similarly, there is a one-to-many association from moviegoers to reviews. Figure 5.9 shows these associations using one type of **Unified Modeling Language (UML)** diagram. We will see more examples of UML in Chapter 11.

In Rails parlance, Figure 5.9 shows that:

- A Moviegoer has many Reviews
- A Movie has many Reviews
- A Review belongs to one Moviegoer and to one Movie

In Rails, the “permanent home” for our model objects is the database, so we need a way to represent associations for objects stored there. Fortunately, associations are so common that relational databases provide a special mechanism to support them: **foreign keys**. A foreign key is a column in one table whose job is to reference the primary key of another table to establish an association between the objects represented by those tables. Recall that by default, Rails migrations create tables whose primary key column is called `id`. Figure 5.10 shows a Moviegoers table to keep track of different users and a Reviews table with foreign key columns `moviegoer_id` and `movie_id`, allowing each review to refer to the primary keys (`ids`) of the user who authored it and the movie it’s about.

id	title	rating
13	Inception	PG-13
41	Star Wars	PG
43	It's Complicated	R

id	movie_id	moviegoer_id	potatoes
21	41	1	5
22	13	2	3
23	13	1	4

id	username
1	alice
2	bob
3	carol

Figure 5.10: In this figure, Alice has given 5 potatoes to Star Wars and 4 potatoes to Inception, Bob has given 3 potatoes to Inception, Carol hasn't provided any reviews, and no one has reviewed It's Complicated. For brevity and clarity, the other fields of the movies and reviews tables are not shown.

<https://gist.github.com/620547b36e435fb06febcb6f889d829f>

```

1 # it would be nice if we could do this:
2 inception = Movie.where(:title => 'Inception')
3 alice,bob = Moviegoer.find(alice_id, bob_id)
4 # alice likes Inception, bob less so
5 alice_review = Review.new(:potatoes => 5)
6 bob_review = Review.new(:potatoes => 3)
7 # a movie has many reviews:
8 inception.reviews = [alice_review, bob_review]
9 # a moviegoer has many reviews:
10 alice.reviews << alice_review
11 bob.reviews << bob_review
12 # can we find out who wrote each review?
13 inception.reviews.map { |r| r.moviegoer.name } # => ['alice','bob']

```

Figure 5.11: A straightforward implementation of associations would allow us to refer directly to associated objects, even though they're stored in different database tables.

For example, to find all reviews for *Star Wars*, we would first form the **Cartesian product** of all the rows of the movies and reviews tables by concatenating each row of the movies table with each possible row of the reviews table. This would give us a new table with 9 rows (since there are 3 movies and 3 reviews) and 7 columns (3 from the movies table and 4 from the reviews table). From this large table, we then select only those rows for which the id from the movies table equals the movie_id from the reviews table, that is, only those movie-review pairs in which the review is about that movie. Finally, we select only those rows for which the movie id (and therefore the review's movie_id) are equal to 41, the primary key ID for *Star Wars*. This simple example (called a **join** in relational database parlance) illustrates how complex relationships can be represented and manipulated using a small set of operations (relational algebra) on a collection of tables with uniform data layout. In SQL, the Structured Query Language used by substantially all relational databases, the query would look something like this:

<https://gist.github.com/cbf6e0db3e759d83fe7cf88d1f58c392>

```

1 SELECT reviews.*
2   FROM movies JOIN reviews ON movies.id=reviews.movie_id
3 WHERE movies.id = 41;

```

If we weren't working with a database, though, we'd probably come up with a design in which each object of a class has "direct references" to its associated objects, rather than constructing the query plan above. A Moviegoer object would maintain an array of references to Reviews authored by that moviegoer; a Review object would maintain a reference to the Moviegoer who wrote it; and so on. Such a design would allow us to write code that looks like Figure 5.11.

Rails' ActiveRecord::Associations¹¹ module supports exactly this design, as we'll learn by doing. Apply the code changes in Figure 5.12 as directed in the caption, and you

<https://gist.github.com/fc8fd0b1a47b0c080edd750bd7d03733>

```

1 # Run 'rails generate migration create_reviews' and then
2 #   edit db/migrate/*_create_reviews.rb to look like this:
3 class CreateReviews < ActiveRecord::Migration
4   def change
5     create_table 'reviews' do |t|
6       t.integer    'potatoes'
7       t.text      'comments'
8       t.references 'moviegoer'
9       t.references 'movie'
10      end
11    end
12  end

```

<https://gist.github.com/9d29abef00dfb898cf95b3de40f39530>

```

1 class Review < ActiveRecord::Base
2   belongs_to :movie
3   belongs_to :moviegoer
4 end

```

<https://gist.github.com/c716c414b5bb050aacc03a18c6f43057>

```

1 # place a copy of the following line anywhere inside the Movie class
2 # AND inside the Moviegoer class (idiomatically, it should go right
3 # after 'class Movie' or 'class Moviegoer'):
4 has_many :reviews

```

Figure 5.12: Top (a): Create and apply this migration to create the `Reviews` table. The new model’s foreign keys are related to the existing `movies` and `moviegoers` tables by convention over configuration. Middle (b): Put this new `Review` model in `app/models/review.rb`. Bottom (c): Make this one-line change to each of the existing files `movie.rb` and `moviegoer.rb`.

should then be able to start `rails console` and successfully execute the examples in Figure 5.11.

How does this work? Since everything in Ruby is a method call, we know that Line 8 in Figure 5.11 is really a call to the instance method `reviews=` on a `Movie` object. This instance method remembers its assigned value (an array of Alice’s and Bob’s reviews) in memory. Recall, though, that since a `Review` is on the “belongs to” side of the association (`Review` belongs to a `Movie`), to associate a review with a movie we must set the `movie_id` field for that review. *We don’t actually have to modify the `movies` table.* So in this simple example, the call to `inception.reviews=` isn’t actually updating the `movie` record for `Inception` at all: it’s setting the `movie_id` field of both Alice’s and Bob’s reviews to “link” them to `Inception`.

has_one is a close relative
of **has_many** that
singularizes the association
method name and operates
on a single owned object
rather than a collection.

Figure 5.13 lists some of the most useful methods added to a `movie` object by virtue of declaring that it `has_many reviews`. Of particular interest is that since `has_many` implies a *collection* of the owned object (`Reviews`), the `reviews` method quacks like a collection. That is, you can use all the collection idioms of Figure 2.11 on it—iterate over its elements with `each`, use functional idioms like `sort`, `map`, and so on, as in lines 8, 10 and 13 of Figure 5.11.

What about the `belongs_to` method calls in `review.rb`? As you might guess, `belongs_to :movie` gives `Review` objects a `movie` instance method that looks up and returns the movie to which this review belongs. Since a review belongs to at most one movie, the method name is singular rather than plural, and returns a single object rather than an enumerable.

<code>m.reviews</code>	Returns an <code>Enumerable</code> of all owned reviews
<code>m.reviews=[r1,r2]</code>	Replaces the set of owned reviews with the set <code>r1, r2</code> , adding or deleting as appropriate, by setting the <code>movie_id</code> field of each of <code>r1</code> and <code>r2</code> to <code>m.id</code> (<code>m</code> 's primary key) in the database immediately.
<code>m.reviews<<r1</code>	Adds <code>r1</code> to the set of <code>m</code> 's reviews by setting <code>r1</code> 's <code>movie_id</code> field to <code>m.id</code> . The change is written to the database immediately (you don't need to do a separate <code>save</code>).
<code>r = m.reviews.build(:potatoes=>5)</code>	Makes <code>r</code> a new, <i>unsaved</i> <code>Review</code> object whose <code>movie_id</code> is preset to indicate that it belongs to <code>m</code> . Arguments are the same as for <code>Review.new</code> .
<code>r = m.reviews.create(:potatoes=>5)</code>	Like <code>build</code> but saves the object immediately (analogous to the difference between <code>new</code> and <code>save</code>).
Note: if the parent object <code>m</code> has never been saved, that is, <code>m.new_record?</code> is true, then the child objects aren't saved until the parent is saved.	
<code>m = r.movie</code>	Returns the <code>Movie</code> instance associated with this review.
<code>r.movie = m</code>	Sets <code>m</code> as the movie associated with review <code>r</code> .

Figure 5.13: A subset of the association methods created by `movie has_many :reviews` and `review belongs_to :movie`, assuming `m` is an existing `Movie` object and `r1, r2` are `Review` objects. Consult the

`ActiveRecord::Associations` documentation¹³ for a full list. Method names of association methods follow convention over configuration based on the name of the associated model.

Summary:

- Associations are one-to-one, one-to-many, or many-to-many relationships among application entities.
- Relational databases (RDBMSs) use foreign keys to represent these relationships.
- ActiveRecord's Associations module uses Ruby metaprogramming to create new methods to “traverse” associations by constructing the appropriate database queries. You must still add the necessary foreign key fields yourself with a migration.

■ ***Elaboration: Associations in a NoSQL world***

The use of foreign keys to represent associations relies on relational algebra. Rails' implementation of ActiveRecord makes associations particularly convenient by providing methods for automatic traversal of associations by synthesizing and optimizing the appropriate foreign-key joins in SQL. But such traversal, if not handled carefully, can lead to performance bottlenecks, as we will see in Section 12.7. One response to such bottlenecks has been the deployment of “NoSQL” databases, such as Cassandra and MongoDB, which omit all but the simplest foreign key support in order to achieve horizontal scalability superior to most RDBMSs. An example of an alternative implementation choice for associations with such databases is Data Mapper (Figure 5.14). In this pattern, each model is associated with a corresponding Mapper class that defines how that model's instances are stored, and provides its own code to represent and traverse model associations. Depending on the complexity of the associations, this code may find itself essentially reimplementing parts of a traditional RDBMS, but without the benefit of the millions of engineer-hours that have gone into optimizing the latter. Indeed, as Chapter 12 describes, “traditional” relational databases can scale impressively far when combined with careful development and operations techniques (dev/ops), so today's SaaS developers can enjoy the expressiveness and advantages of traditional RDBMSs for a long time before the database becomes the bottleneck.

Self-Check 5.4.1. *In Figure 5.12(a), why did we add foreign keys (references) only to the reviews table and not to the moviegoers or movies tables?*

- ◊ Since we need to associate many reviews with a single movie or moviegoer, the foreign keys must be part of the model on the “owned” side of the association, in this case Reviews.

■

Self-Check 5.4.2. *In Figure 5.13, are the association accessors and setters (such as `m.reviews` and `r.movie`) instance methods or class methods?*

- ◊ Instance methods, since a collection of reviews is associated with a particular movie, not with movies in general. ■

5.5 Through-Associations

Referring back to Figure 5.9, there are direct associations between Moviegoers and Reviews as well as between Movies and Reviews. But since any given Review is associated with both a Moviegoer and a Movie, we could say that there's an *indirect* association between Moviegoers and Movies. For example, we might ask “What are all the movies Alice has reviewed?” or “Which moviegoers have reviewed *Inception*?”. Indeed, line 13 in Figure 5.11 essentially answers the second question.

This kind of indirect association is so common that Rails and other frameworks provide an abstraction to simplify its use. It's sometimes called a *through-association*, since Moviegoers are related to Movies *through* their reviews and vice versa. Figure 5.15 shows how to use the `:through` option to Rails' `has_many` to represent this indirect association. You can similarly add `has_many :moviegoers, :through=>:reviews` to the `Movie` model, and write `movie.moviegoers` to ask which moviegoers are associated with (wrote reviews for) a given movie.

How is a through-association “traversed” in the database? Referring again to Figure 5.10, finding all the movies reviewed by Alice first requires forming the Cartesian product of the *three* tables (`movies`, `reviews`, `moviegoers`), resulting in a table that conceptually has 27

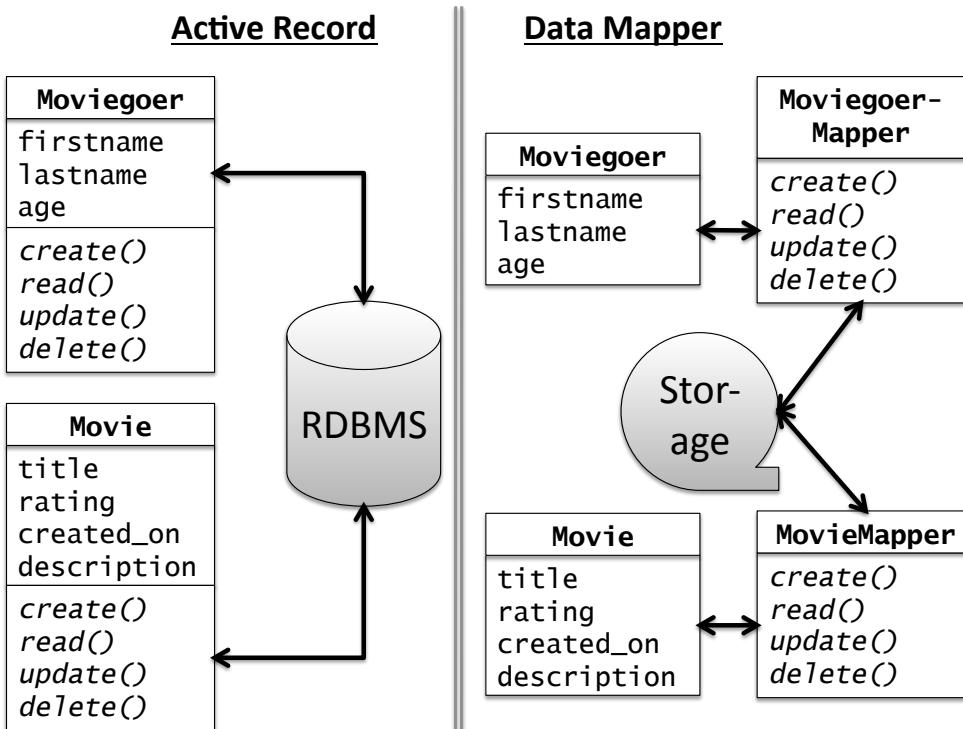


Figure 5.14: In the Active Record design pattern (left), used by Rails and implemented in the `ActiveRecord` module, the model object itself knows how it's stored in the persistence tier, and how its relationship to other types of models is represented there. In the Data Mapper pattern (right), used by Google AppEngine, PHP and Sinatra, a separate class isolates model objects from the underlying storage layer. Each approach has pros and cons. This *class diagram* is one form of Unified Modeling Language (UML) diagram, which we'll learn more about in Chapter 11.

<https://gist.github.com/38c7992c280c175cc6e732ee8b44157d>

```

1 # in moviegoer.rb:
2 class Moviegoer
3   has_many :reviews
4   has_many :movies, :through => :reviews
5   # ...other moviegoer model code
6 end
7 alice = Moviegoer.where(:name => 'Alice')
8 alice_movies = alice.movies
9 # MAY work, but a bad idea - see caption:
10 alice.movies << Movie.where(:title => 'Inception') # Don't do this!

```

Figure 5.15: Using through-associations in Rails. As before, the object returned by `alice.movies` in line 8 quacks like a collection. Note, however, that since the association between a `Movie` and a `Moviegoer` occurs *through* a `Review` belonging to both, the syntax in line 10 will cause a `Review` object to be created to "link" the association, and by default all its attributes will be `nil`. This is almost certainly not what you want, and if you have validations on the `Review` object (for example, the number of potatoes must be an integer), the newly-created `Review` object will fail validation and cause the entire operation to abort.

<https://gist.github.com/5576ce01f90d288484a9d917462c099c>

```

1 class Review < ActiveRecord::Base
2   # review is valid only if it's associated with a movie:
3   validates :movie_id, :presence => true
4   # can ALSO require that the referenced movie itself be valid
5   # in order for the review to be valid:
6   validates_associated :movie
7 end

```

Figure 5.16: This example validation on an association ensures that a review is only saved if it has been associated with some movie.

rows and 9 columns in our example. From this table we then select those rows for which the movie’s ID matches the review’s `movie_id` and the moviegoer’s ID matches the review’s `moviegoer_id`. Extending the explanation of Section 5.4, the SQL query might look like this:

<https://gist.github.com/2c08943e4810a88ed8c0806992d5d22d>

```

1 SELECT movies.*
2   FROM movies JOIN reviews ON movies.id = reviews.movie_id
3   JOIN moviegoers ON moviegoers.id = reviews.moviegoer_id
4 WHERE moviegoers.id = 1;

```

For efficiency, the intermediate Cartesian product table is usually not materialized, that is, not explicitly constructed by the database. Indeed, Rails 3 has a sophisticated relational algebra engine that constructs and performs optimized SQL join queries for traversing associations.

The point of this section and the previous one, though, is not only to explain how to use associations, but also to point out the elegant use of duck typing and metaprogramming that makes them possible. In Figure 5.12(c) you added `has_many :reviews` to the `Movie` class. The `has_many` method performs some metaprogramming to define the new instance method `reviews=` that we used in Figure 5.11. `has_many` is not a declaration, but a regular method call that does all of this work at runtime, adding several new instance methods to your model class to help manage the association. As you’ve no doubt guessed, convention over configuration determines the name of the new method, the table it will use in the database, and so on.

Associations are one of the most feature-rich aspects of Rails, so take a good look at the full documentation¹⁴ for them. In particular:

- Just like `ActiveRecord` lifecycle hooks, associations provide additional hooks that can be triggered when objects are added to or removed from an association (such as when new `Reviews` are added for a `Movie`), which are distinct from the lifecycle hooks of `Movies` or `Reviews` themselves.
- Validations can be declared on associated models, as Figure 5.16 shows.
- Because calling `save` or `save!` on an object that uses associations also affects the associated objects, various caveats apply to what happens if any of the saves fails. For example, if you have just created a new `Movie` and two new `Reviews` to link to it, and you now try to save the `Movie`, any of the three saves could fail if the objects aren’t valid (among other reasons).
- Additional options to association methods control what happens to “owned” objects when an “owning” object is destroyed. For example,



`has_many :reviews, dependent: destroy` specifies that the reviews belonging to a movie should be deleted from the database if the movie is destroyed.

Through-associations summary:

- When two models A and B each have a has-one or has-many relationship to a common third model C, a many-to-many association between A and B can be established through C.
- The `:through` option to `has_many` allows you to manipulate either side of a through-association just as if it were a direct association. However, if you modify a through-association directly, the intermediate model object must be automatically created, which is probably not what you intended.

■ Elaboration: Has and belongs to many

Given that `has_many :through` creates “many-to-many” associations between the two outer entities (Movies and Moviegoers in our running example), could we create such many-to-many relationships directly, without going through an “intermediate” table? ActiveRecord provides another association we don’t discuss here, `has_and_belongs_to_many` (HABTM), for pure many-to-many associations in which you don’t need to maintain any other information about the relationship besides the fact that it exists. For example, on Facebook, a given user might “like” many wall posts, and a given wall post might be “liked by” many users; thus “like” is a many-to-many relationship between users and wall posts. However, even in that simple example, to keep track of *when* someone liked or unliked a wall post, the concept of a “like” would then need its own model to track these extra attributes. In most cases, therefore, `has_many :through` is more appropriate because it allows the relationship itself (in our example, the movie review) to be represented as a separate model. In Rails, HABTM associations are represented by a **join table** that by convention has no primary key and is created with a special migration syntax.

Self-Check 5.5.1. *Describe in English the steps required to determine all the moviegoers who have reviewed a movie with some given id (primary key).*

◊ Find all the reviews whose `movie_id` field contains the `id` of the movie of interest. For each review, find the moviegoer whose `id` matches the review’s `moviegoer_id` field. ■

5.6 RESTful Routes for Associations

How should we RESTfully refer to actions associated with movie reviews? In particular, at least when creating or updating a review, we need a way to link it to a moviegoer and a movie. Presumably the moviegoer will be the `@current_user` we set up in Figure 5.5 (Section 5.1). But what about the movie?

Chapter 7 discusses Behavior-Driven Design, which emphasizes that development should be driven by scenarios that describe actual user behaviors. According to this view, since it only makes sense to create a review when you have a movie in mind, most likely the “Create Review” functionality will be accessible from a button or link on the Show Movie Details page for a particular movie. Therefore, at the moment we display this form element, we

<https://gist.github.com/7979dd386c876060a9ed9e3e01445d10>

```

1 # in routes.rb, change the line 'resources :movies' to:
2 resources :movies do
3   resources :reviews
4 end

```

Helper method	RESTful route and action		
movie_reviews_path(m)	GET /movies/:movie_id/reviews		index
movie_review_path(m)	POST /movies/:movie_id/reviews		create
new_movie_review_path(m)	GET /movies/:movie_id/reviews/new		new
edit_movie_review_path(m,r)	GET /movies/:movie_id/reviews/:id/edit		edit
movie_review_path(m,r)	GET /movies/:movie_id/reviews/:id		show
movie_review_path(m,r)	PUT /movies/:movie_id/reviews/:id		update
movie_review_path(m,r)	DELETE /movies/:movie_id/reviews/:id		destroy

Figure 5.17: Specifying nested routes in `routes.rb` (top) also provides nested URI helpers (bottom), analogous to the simpler ones provided for regular resources.

know what movie the review is going to be associated with. The question is how to get this information to the `new` or `create` method in the `ReviewsController`.

One method we might use is that when the user visits a movie’s Show Details page, we could use the `session[]`, which persists across requests, to remember the ID of the movie whose details have just been rendered as the “current movie.” When `ReviewsController#new` is called, we’d retrieve that ID from the `session[]` and associate it with the review by populating a hidden form field in the review’s form, which in turn will be available to `ReviewsController#create`. However, this approach isn’t RESTful, since the movie ID—a critical piece of information for creating a review—is “hidden” in the session.

A more RESTful alternative, which makes the movie ID explicit, is to make the RESTful routes themselves reflect the logical “nesting” of Reviews inside Movies, as the top part of Figure 5.17 shows. Since `Movie` is the “owning” side of the association, it’s the outer resource. Just as the original `resources :movies` provided a set of RESTful URI helpers for CRUD actions on movies, this *nested resource* route specification provides a set of RESTful URI helpers for CRUD actions on *reviews that are owned by a movie*. The bottom part of Figure 5.17 summarizes the new routes, which are provided *in addition* to the basic RESTful routes on Movies that we’ve been using all along. Note that via convention over configuration, the URI wildcard `:id` will match the ID of the resource itself—that is, the ID of a review—and Rails chooses the “outer” resource name to make `:movie_id` capture the ID of the “owning” resource. The ID values will therefore be available in controller actions as `params[:id]` (the review) and `params[:movie_id]` (the movie with which the review will be associated).

Figure 5.18 shows a simplified example of using such nested routes to create the views and actions associated with a new review. Of particular note is the use of a before-filter in `ReviewsController` to ensure that before a review is created, *two* conditions are true:

1. `@current_user` is set (that is, someone is logged in and will “own” the new review).
2. The movie captured from the route (Figure 5.17) as `params[:movie_id]` exists in



<https://gist.github.com/77a5457f6727787aa6b25802cac2905c>

```

1 class ReviewsController < ApplicationController
2   before_filter :has_moviegoer_and_movie, :only => [:new, :create]
3   protected
4   def has_moviegoer_and_movie
5     unless @current_user
6       flash[:warning] = 'You must be logged in to create a review.'
7       redirect_to login_path
8     end
9     unless (@movie = Movie.where(:id => params[:movie_id]))
10      flash[:warning] = 'Review must be for an existing movie.'
11      redirect_to movies_path
12    end
13  end
14  public
15  def new
16    @review = @movie.reviews.build
17  end
18  def create
19    # since moviegoer_id is a protected attribute that won't get
20    # assigned by the mass-assignment from params[:review], we set it
21    # by using the << method on the association. We could also
22    # set it manually with review.moviegoer = @current_user.
23    @current_user.reviews << @movie.reviews.build(params[:review])
24    redirect_to movie_path(@movie)
25  end
26 end

```

<https://gist.github.com/f19bf92a35686d87a5e2b8d4e0384970>

```

1 <h1> New Review for <%= @movie.title %> </h1>
2
3 <%= form_tag movie_review_path(@movie), class: 'form' do %>
4   <label class="col-form-label"> How many potatoes:</label>
5   <%= select_tag 'review[potatoes]', options_for_select(1..5), class: 'form-
6     control' %>
7   <%= submit_tag 'Create Review', :class => 'btn btn-success' %>
<% end %>

```

Figure 5.18: Top (a): a controller that manipulates Reviews that are “owned by” both a Movie and a Moviegoer, using before-filters to ensure the “owning” resources are properly identified in the route URL. Bottom (b): A possible view template for creating a new review, that is, app/views/reviews/new.html.erb.

the database.

If either condition is not met, the user is redirected to an appropriate page with an error message explaining what happened. If both conditions are met, the controller instance variables `@current_user` and `@movie` become accessible to the controller action and view.

The view uses the `@movie` variable to create a submission path for the form using the `movie_review_path` helper (Figure 5.17 again). When that form is submitted, once again `movie_id` is parsed from the route and checked by the before-filter prior to calling the `create` action. Similarly, we could link to the page for creating a new review by calling `link_to` with the route helper `new_movie_review_path(@movie)` as its URI argument.

Summary: controller and view support for associations

- The RESTful way to create routes for associations is to capture the IDs of both the resource itself and its associated item(s) in a “nested” route URI.
- When manipulating “owned” resources that have a parent, such as Reviews that are “owned by” a Movie, before-filters can be used to capture and verify the validity of the IDs embedded in the RESTful nested route.

■ Elaboration: SOA, RESTful association routes, and the session

RESTful SOA design guidelines suggest that every request be self-contained, so that there is no concept of a session (nor any need for one). In our example, we used nested RESTful resource routes to keep the movie and review IDs together and relied on our authentication framework to set up `@current_user` as the moviegoer who owns the review. For a pure SOA API, we would need to capture the moviegoer ID *and* review ID along with the movie ID. Rails’ routing subsystem is flexible enough to allow defining routes with multiple wildcard components for this purpose. In general, this design problem arises whenever you need to create an object with multiple “owners” such as a Review. If not all the owning objects are required in order for the owned object to be valid—for example, if it were possible for a Review to be “anonymous”—another solution would be to separate creation of the review and assigning it to a moviegoer into different RESTful actions.

Self-Check 5.6.1. *Why must we provide values for a review’s `movie_id` and `moviegoer_id` to the `new` and `create` actions in `ReviewsController`, but not to the `edit` and `update` actions?*

- ◊ Once the review is created, the stored values of its `movie_id` and `moviegoer_id` fields tell us the associated movie and moviegoer. ■

5.7 CHIPS: Associations

CHIPS 5.7: Associations

<https://github.com/saasbook/hw-associations-reviews>

We add a `Reviews` model to `RottenPotatoes`, and make it possible for a logged-in moviegoer to leave a review and to view all reviews for a movie.

5.8 Other Types of Code

The basic Rails app structure is apparent from the arrangement of the `app` directory: there are models backed by the database; views that render to HTML; controllers that should contain the bare minimum code to mediate between models and views; and view helpers (subdirectory `helpers`) for code whose only job is to “prettify” model information in the views.

In this section we describe many other types of code necessary in large apps that don’t fit neatly into any of the above categories. There’s no fixed consensus on where these go in a Rails app, but it’s probably helpful to create additional subdirectories under `app`, since anything in that directory is automatically loaded and available within your Rails app.

Our short (and incomplete) list of examples of other types of useful objects can be divided into three categories, based on the main role of each object type:

1. Objects that factor out code (presenter, value object, adapter/decorator) provide a place to put additional code that works directly to help a particular model, view, or controller, but isn’t part of the core functionality of the class it helps.
2. Objects that DRY out code (concerns, automations) allow reuse of behaviors.
3. Objects that encapsulate coupling (service object, form object, query object, policy object) perform operations that express inherent dependencies among different classes, so that those dependencies don’t creep into the classes themselves.

Presenters (sometimes called **view objects**) contain code that helps in rendering complex views. Recall from Section 4.1 that Rails views are really view templates: ideally they contain little or no code other than calls to view helpers. But sometimes the code needed to gracefully manipulate and present a view starts getting too heavyweight to stuff into a view helper, which is just a namespace of methods that get mixed into views. Presenters are full classes and provide a more appropriate place for complex view logic.

Value objects encapsulate a type of object whose comparisons are based on values. For example, consider an object representing a range of dates. Depending on your app’s needs, you might define one such object instance to be “less than” another if its starting date is earlier, or if its ending date is earlier; you could also come up with a definition for “between.” Encapsulating such an object in a class lets you define the spaceship comparison operator `<=>` on instances of that class, so mixed-in methods like `sort` will “just work.”

Adapters and decorators are design patterns related to the Open-Closed Principle (Section 11.4) and Dependency Injection Principle (Section 11.6). As their name suggests, these typically provide extra functionality to a particular class, usually a model.

Concerns are complex behaviors mixed into multiple models. For example, the Apache Solr¹⁵ engine adds sophisticated full-text searching to any database-backed app. Any ActiveRecord model that “mixes in” Solr gets new search methods added in. A simpler example might be allowing any model to be “voted on” (likes/dislikes): the functionality of managing and storing vote information is generic, but each model needs to track it separately. A third example might be allowing any model to be associated with an uploaded file. Concerns should be used with care because they represent a kind of inheritance, whereas (as Chapter 11 describes) cleaner code can often be achieved by preferring composition over inheritance.

Automations are just that—automated workflows that save you from having to manually repeat actions, usually related to app management and deployment. For example, creating fake staging data for your app would be a great candidate for automation, since the data

needs to be re-created each time the app is deployed to staging. Most Rails app automations are best accomplished by adding `rake` tasks, since these have access to the app's classes, environment settings, and so on, though that's certainly not the only way to do it.

Service objects typically perform a complex operation that touches multiple models, so its logic doesn't naturally fit into a single model. For example, finalizing a purchase on an e-commerce site might touch a table of sales transactions, a table of inventory, and a table representing the customer's orders. These tables probably back three different models. A service object is often stateless (not backed by its own database table) but it can be helpful to include an instance of `ActiveModel::Errors` as part of the object, so the object's error reporting is just like that of ActiveRecord models, making it easy for controllers to call the object and report its errors. Service objects in Rails apps are particularly useful when combined with a **database transaction** to ensure that either all the updates occur or none do.

Form objects encapsulate the processing of a single form that may update multiple models. Continuing the example above, a single form for completing a purchase may include information that updates the customer's shipping address, the history of all orders, and so on. The form object contains logic that coordinates the changes to the various models when the form is submitted.

Query objects can similarly encapsulate queries that touch many different models. While a query in model A can use joins and eager loading (Section 12.7) to reference fields in model B, such queries often end up exposing substantial details of each of the models, introducing coupling between them. While this coupling is necessary in order to perform the query, encapsulating it in a query object allows the models themselves to remain decoupled from each other and easier to test.

Policy objects can be thought of as a special case of service object: they encapsulate policies, such as "who is allowed to do what," especially those that touch multiple models and therefore don't naturally belong in any of them. For example, consider an airline loyalty program that reserves its best seats for VIP frequent flyers. The policy decision of "Is customer X allowed to reserve seat Y on flight Z" may depend on many factors, including the customer's status level, how full the flight is, and so on. A policy object knows how to retrieve the necessary details from the relevant model classes and make a policy decision.

Summary: A Model–View–Controller app will make use of other kinds of code in addition to models, views, and controllers. While some such code is there to DRY out the app or provide support beyond the core functionality of a specific class, an important category of "other types of code" encapsulates dependencies among multiple classes, so that the classes themselves can remain relatively decoupled.

Self-Check 5.8.1. *Rails database migrations are an example of which of the kinds of code described in this section?*

- ◊ Migrations are an example of automation, since the same migration code is used to update the development, test, and production databases. ■

Self-Check 5.8.2. *True or false: Query objects exist because Rails makes it illegal/impossible for an ActiveRecord query defined in model A to make reference to fields in model B.*

- ◊ False: ActiveRecord queries constructed as a result of associations (such as "A has many

Bs") necessarily refer to other models, and any query defined in model A can join with and refer to any fields in model B. But a query object lets you extract such logic into its own class when the queries get so complex that they expose too much detail about model B to model A. ■

5.9 Fallacies and Pitfalls



Pitfall: Too many filters or model lifecycle callbacks, or overly complex logic in filters or callbacks.

Filters and callbacks provide convenient and well-defined places to DRY out duplicated code, but too many of them can make it difficult to follow the app's logic flow. For example, when there are numerous before-filters, after-filters and around-filters that trigger on different sets of controller actions, it can be hard to figure out why a controller action fails to execute as expected or which filter "stopped the show." Things can be even worse if some of the filters are declared not in the controller itself but in a controller from which it inherits, such as `ApplicationController`. Filters and callbacks should be used when you truly want to centralize code that would otherwise be duplicated.



Pitfall: Not checking for errors when saving associations.

Saving an object that has associations implies potentially modifying multiple tables. If any of those modifications fails, perhaps because of validations either on the object or on its associated objects, other parts of the save might silently fail. Be sure to check the return value of `save`, or else use `save!` and rescue any exceptions.



Pitfall: Nesting resources more than 1 level deep.

Although it's technically possible to have nested resources multiple levels deep, the routes and actions quickly become cumbersome, which may be a sign that your design isn't properly factored. Perhaps there is an additional entity relationship that needs to be modeled, using a shortcut such as `has_many :through` to represent the final association.



5.10 Concluding Remarks: Languages, Productivity, and Beauty



This chapter showed two examples of using language features to support the productive creation of beautiful and concise code. The first is the use of metaprogramming, closures and higher-order functions to allow model validations and controller filters to be DRYly declared in a single place, yet called from multiple points in the code. Validations and filters are an example of **aspect-oriented programming** (AOP), a methodology that has been criticized because it obfuscates control flow but whose well-circumscribed use can enhance DRYness. All in all, validations, filters, and association helper methods are worth studying as successful examples of tastefully exploiting programming language features to enhance code beauty and productivity.

The second example is the design choices reflected in the association helper methods. For example, you may have noticed that while the foreign key field for a `Movie` object associated with a review is called `movie_id`, the association helper methods allow us to reference

AOP has been compared with the fictitious **COME FROM** programming language construct, which began as a humorous response to Edsger Dijkstra's letter **Go To Statement Considered Harmful** (Dijkstra 1968) promoting structured programming.

`review.movie`, allowing our code to focus on the *architectural* association between Movies and Reviews rather than the *implementation detail* of the foreign key names. You could certainly manipulate the `movie_id` or `review_id` fields in the database directly, as Web applications based on less-powerful frameworks are often forced to do, or do so in your Rails app, as in `review.movie_id=some_movie.id`. But besides being harder to read, this code hardwires the assumption that the foreign key field is named `movie_id`, which may not be true if your models are using advanced Rails features such as polymorphic associations, or if ActiveRecord has been configured to interoperate with a legacy database that follows a different naming convention. In such cases, `review.movie` and `review.movie=` will still work, but referring to `review.movie_id` will fail. Since someday *your* code will be legacy code, help your successors be productive—keep the logical structure of your entities as separate as possible from the database representation.



We might similarly ask, now that we know how associations are stored in the RDBMS, why `movie.save` actually also causes a change to the `reviews` table when we save a movie after adding a review to it. In fact, calling `save` on the new review object would also work, but having said that a Movie has many Reviews, it just makes more sense to think of saving the Movie when we update which Reviews it has. In other words, it's designed this way in order to make sense to programmers and make the code more beautiful.



Finally, as we saw in Section 5.8, an application framework provides direct support for the major architectural components of the application (in our case, models, views, and controllers), but any large software system contains many other kinds of code as well. Indeed, some of the examples of that section are best understood as additional patterns for solving software problems—a theme to which we will frequently return, and that we treat in depth in Chapter 11.

E. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3): 147–148, March 1968. URL <https://dl.acm.org/purchase.cfm?id=362947&CFID=100260848&CFTOKEN=27241581>.

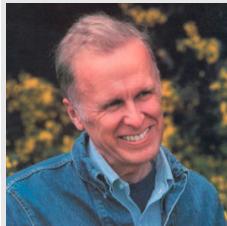
Notes

- ¹<https://www.khanacademy.org/computing/computer-programming/sql>
- ²<http://api.rubyonrails.org/classes/ActiveModel/Validations/ClassMethods.html#method-i-validates>
- ³<http://api.rubyonrails.org/classes/ActiveModel/Validations/ClassMethods.html#method-i-validates>
- ⁴<http://api.rubyonrails.org/classes/ActiveModel/Errors.html>
- ⁵http://guides.rubyonrails.org/v3.2.19/active_record_validations_callbacks.html
- ⁶http://guides.rubyonrails.org/v3.2.19/active_record_validations_callbacks.html
- ⁷<http://api.rubyonrails.org/v3.2.19/classes/ActiveRecord/Callbacks.html>
- ⁸<https://oauth.net/2>
- ⁹<http://www.omniauth.org>
- ¹⁰<http://www.omniauth.org>
- ¹¹<http://api.rubyonrails.org/v3.2.19/classes/ActiveRecord/Associations/ClassMethods.html>
- ¹²<http://api.rubyonrails.org/v3.2.19/classes/ActiveRecord/Associations/ClassMethods.html>
- ¹³<http://api.rubyonrails.org/v3.2.19/classes/ActiveRecord/Associations/ClassMethods.html>
- ¹⁴<http://api.rubyonrails.org/v3.2.19/classes/ActiveRecord/Associations/ClassMethods.html>
- ¹⁵<https://lucene.apache.org/solr/>

6

Mobile and Desktop SaaS Clients: JavaScript Introduction

John Backus (1924–2007) received the 1977 Turing Award in part for “profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran,” which was the first widely used high-level language.



Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs.

—John Backus, quoted in IBM employee magazine *Think* in 1979

6.1	JavaScript: The Big Picture	154
6.2	Introducing ECMAScript	157
6.3	Classes, Functions and Constructors	163
6.4	The Document Object Model (DOM) and jQuery	166
6.5	The DOM and Accessibility	169
6.6	Events and Callbacks	173
6.7	AJAX: Asynchronous JavaScript And XML	178
6.8	Testing JavaScript and AJAX	183
6.9	CHIPS: AJAX Enhancements to RottenPotatoes	190
6.10	Single-Page Apps and JSON APIs	190
6.11	Fallacies and Pitfalls	195
6.12	Concluding Remarks: JavaScript Past, Present and Future	199

Prerequisites and Concepts

Concepts:

JavaScript is a dynamic, interpreted scripting language built into modern browsers. This chapter describes its main features, including some that we recommend avoiding because they represent questionable design choices, and how to use it to extend the types of content and applications that can be delivered as SaaS.

- A browser represents a web page as a data structure called the **Document Object Model** (DOM). JavaScript code running in the browser can inspect and modify this data structure, causing the browser to redraw the modified page elements.
- When a user interacts with the browser (for example, by typing, clicking, or moving the mouse) or the browser makes progress in an interaction with a server, the browser generates an **event** indicating what happened. Your JavaScript code can take app-specific actions to modify the DOM when such events occur.
- Using **AJAX**, or Asynchronous JavaScript And XML, JavaScript code can make HTTP requests to a Web server *without* triggering a page reload. The information in the response can then be used to modify page elements in place, giving a richer and often more responsive user experience than traditional Web pages. Rails partials and controller actions can be readily used to handle AJAX interactions.
- Just as we use the highly-productive Rails framework (Chapter 4) and RSpec TDD tool (Chapter 8) for server-side SaaS code, here we use the highly-productive **jQuery** framework and Jasmine¹ TDD tool to develop client-side code.
- We follow the best practice of “graceful degradation,” also referred to as “progressive enhancement”: legacy browsers lacking JavaScript support will still provide a good user experience, while JavaScript-enabled browsers will provide an even better experience.

6.1 JavaScript: The Big Picture

JavaScript had to “look like Java” only less so—be Java’s dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JavaScript would have happened.

—Brendan Eich, creator of JavaScript

Despite its name, JavaScript is unrelated to Java: LiveScript, the original name chosen by Netscape Communications Corp., was changed to JavaScript to capitalize on Java’s popularity. Brendan Eich, creator of JavaScript, originally proposed embedding Scheme in the browser (Seibel 2009). Although pressure to create a Java-like syntax prevailed, many Scheme ideas survive in JavaScript, including higher-order functions (HOFs), which take functions as arguments and/or produce a function as a return value, which make possible AJAX programming and the Jasmine TDD tool.

Indeed, JavaScript has almost nothing in common with Java except superficial syntax, and is actually much more similar to Ruby. Its dynamic type system is similar to Ruby’s and plays a similarly prominent role in how the language is used. If you have a solid grasp of these concepts from Chapters 2 and 8, and are comfortable using CSS selectors as you did in Chapters 3 and 7, getting started with JavaScript will be easy.

JavaScript has acquired a bad reputation that isn’t entirely deserved. It began as a language intended to allow Web browsers to run simple client-side code to validate form inputs, animate page elements, or communicate with Java applets. Inexperienced programmers began to copy-and-paste simple JavaScript examples to achieve appealing visual effects, albeit with terrible programming practices, giving the language itself a bad reputation. This is not to say the language has no quirks or pitfalls, but it is certainly possible to use it well.

Nonetheless, even when used well, JavaScript still presents a fundamental tension for SaaS. While JavaScript is definitively the language of client-side code, there are many languages for creating the server side. Thus when creating SaaS we must either find ways to integrate two (or more) languages in the same app, as we do in this chapter, or commit to creating both the front and back ends in a single language, as is done when using server-side JavaScript frameworks such as Node or Express.

While it seems appealing to simply pick one language, it is a decision your authors believe to be on the wrong side of history. Few modern complex software systems are written entirely in a single language simply because different languages solve different problems well. For example, the routing layer of Heroku, which receives and distributes incoming traffic from SaaS clients, is written in Erlang, a somewhat obscure language developed for programming highly-reliable telecommunications switches. Erlang is so good at simplifying the creation of code to manage event-driven tasks like handling multiple simultaneous connections that it is worth specializing the routing layer in this way. This philosophy is consistent with that of microservices: each service should be engineered to optimize its focused set of tasks, and as long as the external API stays the same, the implementation language matters little and can even be changed without harming interoperability. Heroku apps have access to shared or dedicated databases using PostgreSQL and Redis (both written mostly in C) and provides runtime support for apps written in a variety of languages. Systems that combine languages are more challenging to develop because of the need to pass information between subsystems in different languages in a disciplined way. Nonetheless, your authors believe that such “polyglot” (multilanguage) systems are the future, and managing such challenges is part of the domain of software engineering.

JavaScript, Microsoft JScript, and Adobe ActionScript are dialects of **ECMAScript**, the 1997 standard that codifies the language. We follow standard usage and use “JavaScript” to refer to the language generically.

Therefore, we concentrate primarily on the use of JavaScript for SaaS clients, showing how the above problems are addressed with respect to Rails as a specific example. We can differentiate four major approaches, which we list in order of “least JavaScript-intensive” to “most JavaScript-intensive”:

1. Adding JavaScript to HTML and CSS to enhance the user experience of server-centric SaaS apps. In this scenario, the user experience is that the app consists of a set of different HTML pages, some of which are JavaScript-enhanced. JavaScript requests.
2. Creating ***single-page applications*** (SPAs) in which the user experience is that once the initial page is loaded, *no* further page reloads or redraws occur, although elements on the page are updated continuously in response to communication with the server. In this scenario, the server appears to the app as one or several service endpoints that return data—most commonly encoded as JSON or XML, though in principle any data format is possible.
3. Writing full client-side applications such as Google Docs, comparable in complexity to desktop apps and possibly able to operate while disconnected from the Internet. Like all complex software, such apps reflect some underlying architecture and are increasingly built using JavaScript frameworks that support that architecture, such as Model-View-ViewModel (Vue), Model-View-Controller (Angular), and others.
4. Creating full server-side apps similar to those we’ve been building using Rails, but using JavaScript frameworks such as Node.js.

The browser determines
what version of ECMAScript is available. Especially if you plan to support mobiles, it's worth checking which browser versions your chosen JavaScript libraries are compatible with.

In this chapter we focus on cases 1 and 2. But how to choose among the ever-expanding selection of purely front-end JavaScript frameworks? Recall from Chapters 3 and 4 the benefits of choosing a framework whose model is a good fit for your app versus the pain of fighting a framework that is a poor fit. Rails is highly opinionated, and if the app you’re trying to build matches Rails’ MVC architecture, its facilities save you a lot of work. The same lesson applies to front-end frameworks, which we can classify on a spectrum from *incremental* to *opinionated*. Roughly speaking, an incremental framework is one in you can selectively use for parts of the front end of your SaaS app without having to commit the entire front end to a very specific (opinionated) app structure. For example, React is primarily a reactive view framework that you can choose to use for some but not all of your app’s views. However, most React components in practice rely on **JSX**, an XML-like extension to JavaScript’s syntax. The Vue view framework does not, making it perhaps easier to incrementally incorporate² into projects that otherwise have no need for JSX. In contrast to both React and Vue, Angular and Ember are full Model–View–Controller frameworks for the front end, and it is difficult to combine them with other (non-MVC) structures, so using them represents a serious commitment to (re)structuring the app’s front end.

We will stake out a conservative position in this introduction and introduce enhancing SaaS with ***unobtrusive JavaScript*** and the jQuery library. In general, unobtrusive JavaScript emphasizes:

- **Separation of HTML markup (content) from JavaScript (behavior):** JavaScript mixed into HTML pages (as was sadly common for early SaaS apps) is hard to read and maintain. JavaScript should instead reside in separate files, using event handlers to “tie” JavaScript code to page elements, as Section 6.7 describes.

	Server	Client
Language	Ruby	JavaScript
Framework	Rails	jQuery
Client-Server Architecture over HTTP	Controller receives request, interacts with model, renders new page (view)	Controller receives request, interacts with model, and renders a partial or an XML- or JSON-encoded object, which is used by JavaScript code running in browser to modify current page in place
Debugging	Ruby debugger, rails console	Firebug, browser's JavaScript console
Testing	RSpec with rspec-rails; isolate tests from database using ActiveRecord model fixtures and factories	Jasmine, jasmine-jquery; isolate tests from server using HTML and JSON fixtures

Figure 6.1: The correspondence between our exposition of server-side programming with Ruby and Rails and client-side programming with JavaScript continues our focus on productively creating DRY, concise code that is well covered by tests.

[quirksmode.org](#) tells you more about JSAPI browser incompatibilities than you want to know.

- **Progressive enhancement:** While JavaScript may enhance aspects of the site’s user experience, the site should remain accessible to users with disabilities (Section 6.5) and users whose browsers have compatibility issues related to the JavaScript Application Programming Interface (JSAPI), the browser functionality that lets JavaScript code manipulate the content of the current HTML page. Both conditions can be largely but not completely addressed by using JavaScript libraries such as jQuery (Section 6.4).

Testing JavaScript is particularly important since most browsers do not display user-visible error messages as a result of JavaScript bugs; instead, the site often simply refuses to work (“silent failure”), or the UI freezes, or the user sees a confusing error message resulting from a cascading failure rather than directly describing the actual failure. As we will see, integration-level testing of JavaScript-enhanced SaaS apps is straightforward and requires few or no changes to your existing skills using Cucumber and Capybara, but unit-testing of JavaScript is somewhat trickier.

The rest of this chapter introduces the JavaScript language and jQuery framework and how they interact with Rails. Figure 6.1 compares our exposition of server-side and client-side programming. Screencast 6.1.1 demonstrates the two JavaScript features we will add to RottenPotatoes in this chapter.

Screencast 6.1.1: Adding JavaScript features to RottenPotatoes.

http://youtu.be/fFe_tdWi2E

We will first add a checkbox that allows filtering the RottenPotatoes movie list to exclude films unsuitable for children. This behavior can be implemented entirely in client-side JavaScript using techniques described in Sections 6.4 and 6.6. Next we will change the behavior of the “More info” link for each movie to display the extra info in a “floating” window rather than loading a new page. This will require AJAX, since fetching the movie info requires communicating with the server. Section 6.7 introduces AJAX programming. Both behaviors will be implemented with graceful degradation so that legacy browsers still have a good experience.



Section 6.2 introduces the language and how code is connected to Web pages and Section 6.3 describes how its functions work, an understanding of which is the basis of writing

clean and unobtrusive JavaScript code. Section 6.4 introduces jQuery³, which overlays the separate browsers' incompatible JSAPIs with a single API that works across all browsers. Section 6.5 discusses how the use of JavaScript to manipulate the DOM intersects with considerations of accessibility for Web users with disabilities. Section 6.6 describes how jQuery's features make it easy to program interactions between page elements and JavaScript code, setting the stage for introducing AJAX in Section 6.7. In 1998, Internet Explorer 5 introduced a new mechanism that allowed JavaScript code to communicate with a SaaS server *after* a page had been loaded, and use information from the server to update the page "in place" without the user having to reload a new page. Other browsers quickly copied the technology. Developer Jesse James Garrett coined the term **AJAX**, for Asynchronous JavaScript And XML, to describe how the combination of this technology to power impressive "Web 2.0" apps like Google Maps.

jQuery can be viewed as an enhanced Adapter (Section 11.6) to the various browsers' JSAPIs.

Testing client-side JavaScript is challenging because browsers will fail silently when an error occurs rather than displaying JavaScript error messages to unsuspecting users. Fortunately, the Jasmine TDD framework will help you test your code, as Section 6.8 describes.

Finally, Section 6.10 describes the mechanisms for both developing and testing browser-based single-page apps (SPAs), which are becoming increasingly popular.

Ironically, modern AJAX programming involves much less XML than originally, as we'll see.

Summary of JavaScript background:

- JavaScript resembles Java in name and syntax only; despite nontrivial flaws, it embodies great ideas found in Scheme and Ruby.
- While all browser-based SaaS apps *must* use JavaScript as the client-side language, there are many choices for the server-side language. Combining languages in a system brings challenges, but in the end, your authors believe, and the history of successful software systems confirms, that the benefits of engineering good "polyglot" (multi-language) systems outweigh the costs.
- We focus on client-side JavaScript, that is, on using the language to enhance the user experience of pages delivered by a server-centric SaaS app. To that end, we choose to use jQuery, which is much closer to the "incremental" end of the front-end framework spectrum than to the "opinionated" end.

Self-Check 6.1.1. *With respect to the challenges of combining multiple languages in one software system, which challenge do you think the JSON data format (introduced in Section 3.6) addresses?*

- ◊ JSON provides a language-independent way to represent hierarchical data structures consisting of basic types such as numbers, strings, Booleans, hash maps (dicts), and arrays. Since virtually all modern languages support these basic types, languages can easily convert between JSON and their own internal representations of these types, allowing for data interchange among them. ■

6.2 Introducing ECMAScript

Stop me if you think you've heard this before.

—variously attributed

Our fast-paced introduction to JavaScript follows the same structure proposed in Section 2.1 and used in Section 2.3 to introduce Ruby. We'll first review the things that are fairly standard about the language—syntax, types and names, control flow, and so on, as Figure 6.2 summarizes. We then introduce the idioms that are central to using the language but less common: prototypes, first-class functions, and higher-order functions. An excellent resource to lend depth to this brief overview is the JavaScript documentation maintained by the Mozilla Developer Network⁴.

Types and typing. Almost everything is an object. There are only a few primitive (built-in) types: String, Number (64-bit double precision floating point), undefined (having no value), null (a specific value different from undefined), Boolean (either true or false), and BigInt (rarely needed, for expressing integers of arbitrary magnitude that would overflow the Number type). There is a new Symbol type (which behaves similar to Ruby's)

The most important compound type is Object, which is a collection of unordered key-value pairs; the keys are called *properties* or sometimes *slots*. JavaScript objects look and behave like Ruby hashes except that the property names must be strings, although JavaScript syntax allows omitting quotes around those strings under some circumstances. Properties can be added or removed after an object is created.

Finally, JavaScript has Arrays that can be indexed numerically, but they are actually implemented as objects (hashes) in which there is a particular relationship between property names that are integers and the array's length property.

Variables and Names. As in Ruby, variables don't have types, but the objects they refer to do, so the same variable can refer to objects of different types at different times (though that's rarely a good idea). Variable names must start with a letter, underscore, or dollar sign, and can also include digits, and idiomatically use UpperCamelCase or lowerCamelCase naming. A variable declaration preceded by var or let declares and optionally initializes the variable, as in var s="Hello world", and sets the scope of that variable to be its enclosing block. Unlike Ruby, but like C, JavaScript allows blocks of code to be nested; a variable declared with var is visible to blocks nested inside the one in which it's declared, whereas a variable declared with let is not. If your functions are short (as Chapter 9.5 suggests)

Functions are closures that carry their environment around with them, allowing them to execute properly at a different place and time than where they were defined. Just as anonymous blocks (do...end) are ubiquitous in Ruby, anonymous functions (function() {...}) are ubiquitous in JavaScript. Classes and types matter even less than they do in Ruby—in fact, despite the syntactic appearance of much JavaScript code in the wild, JavaScript does not have classes that behave the way they would in class-oriented OO languages like Ruby and Java, despite the appearance of the new class keyword in ECMAScript 6.

Figure 6.2 shows JavaScript's basic syntax and constructs, which should look familiar to Java and Ruby programmers. The Fallacies & Pitfalls section describes several JavaScript pitfalls associated with the figure; read them carefully after you've finished this chapter, or you may find yourself banging your head against one of JavaScript's unfortunate misfeatures or a JavaScript mechanism that looks and works almost but not quite like its Ruby counterpart. For example, whereas Ruby uses nil to mean both “undefined” (a variable that has never been given a value) and “empty” (a value that is always false), JavaScript's null is distinct from its undefined, which is what you get as the “value” of a variable that has never been initialized.

As the first row of Figure 6.2 shows, JavaScript's fundamental type is the object, an

let signals the JavaScript interpreter that the variable is likely to be reassigned later. Using const instead makes it an error to reassign the variable later. Using var, which was the only option in older version of JavaScript, leaves it ambiguous but may prevent the interpreter from doing certain optimizations.

Objects	<code>movie={title: 'The Godfather', 'releaseInfo': {year: 1972, rating: 'PG'}}</code> Quotes optional around property name if it's a legal variable name; objects can be nested. Access an object's properties with <code>movie.title</code> , or <code>movie['title']</code> if property name isn't a legal variable name or isn't known until runtime. <code>for (var in obj) {...}</code> iterates over <code>obj</code> 's property names in arbitrary order.
Types	<code>typeof x</code> returns a string representation of <code>x</code> 's primitive type: one of "object", "string", "array", "number", "boolean", "function", "undefined". <i>All numbers are doubles.</i>
Strings &Regexp	"string", 'also a string', 'joining'+'strings' <code>'mad, mad world'.split(/[,]+/) == ["mad", "mad", "world"]</code> <code>'mad, mad world'.slice(3,8)==", mad";</code> <code>'mad, mad world'.slice(-3)=="rld"</code> <code>'mad'.indexOf('d')==2, 'mad'.charAt(2)=='d',</code> <code>'mad'.charCodeAt(4)==100</code> <code>'mad'.replace(/(\w)\$/,'\$1\$1er')=="madder"</code> <code>/regexp/.exec(string)</code> if no match returns <code>null</code> , if match returns array whose zeroth element is whole string matched and additional elements are parenthesized capture groups. <code>string.match(/regexp/)</code> does the same, <i>unless</i> the <code>/g</code> regexp modifier is present. <code>/regexp/.test(string)</code> (faster) returns <code>true</code> or <code>false</code> but no capture groups. Alternate constructor: <code>new RegExp('[Hh]e(l+)o')</code>
Arrays	<code>var a = [1, {two: 2}, 'three'] ; a[1] == {two: 2}</code> Zero-based, grow dynamically; objects whose keys are numbers (see Fallacies & Pitfalls) <code>arr.sort(function (a,b) {...})</code> Function returns -1, 0 or 1 for <code>a<b, a==b, a>b</code>
Numbers	<code>+ - / %, also +=, etc., ++ --, Math.pow(num,exp)</code> <code>Math.round(n), Math.ceil(n), Math.floor(n)</code> round their argument to nearest, higher, or lower integer respectively <code>Math.random()</code> returns a random number in (0,1)
Conversions	<code>'catch'+22=='catch22', '4'+'11'=='411'</code> <code>parseInt('4oneone')==4, parseInt('four11')==NaN</code> <code>parseInt('0101',10)==101, parseInt('0101',2)==5,</code> <code>parseInt('0101')==65</code> (numbers beginning with 0 are parsed in octal by default, unless radix is specified) <code>parseFloat('1.1b23')==-1.1, parseFloat('1.1e3')==1100</code>
Booleans	<code>false, null, undefined</code> (undefined value, different from <code>null</code>), 0, the empty string '', and <code>NaN</code> (not-a-number) are <i>falsy</i> (Boolean false); <code>true</code> and all other values are <i>truthy</i> .
Naming	<code>localVar, local_var, ConstructorFunction, GLOBAL</code> All are conventions; JavaScript has no specific capitalization rules. <code>var</code> keyword scopes variable to the function in which it appears, otherwise it becomes a global (technically, a property of the global object, as Section 6.3 describes). Variables don't have types, but the objects they refer to do.
Control flow	<code>while(), for();, if...else if...else, ?: (ternary operator), switch/case, try/catch/throw, return, break</code> Statements separated by semicolons; interpreter tries to auto-insert "missing" ones, but this is perilous (see Fallacies & Pitfalls)

Figure 6.2: Analogous to Figure 2.2, this table summarizes basic constructs of JavaScript. See the text for important pitfalls. Whereas Ruby uses `nil` as both an explicit null value and the value returned for nonexistent instance variables, JavaScript distinguishes `undefined`, which is returned for undeclared or unassigned variables, from the special value `null` and Boolean `false`. However, all three are "falsy"—they evaluate to `false` in a conditional.

<https://gist.github.com/707970f51340686283f71858a4f02c7a>

```

1 let potatoReview =
2 {
3   "potatoes": 5,
4   "reviewer": "armandofox",
5   "movie": {
6     "title": "Casablanca",
7     "description": "Casablanca is a classic and iconic film starring ...",
8     "rating": "PG",
9     "release_date": "1942-11-26T07:00:00Z"
10   }
11 };
12 potatoReview['potatoes'] // => 5
13 potatoReview['movie'].title // => "Casablanca"
14 potatoReview.movie.title // => "Casablanca"
15 potatoReview['movie']['title'] // => "Casablanca"
16 potatoReview['blah'] // => undefined

```

Figure 6.3: JavaScript notation for object literals, that is, objects you specify by enumerating their properties and values explicitly. If the property name is a legal JavaScript variable name, quotes can be omitted or the idiomatic dot-notation shortcut (lines 13–14) can be used, although quotes are always required around all strings when an object is expressed in JSON format. Since objects can contain other objects, hierarchical data structures can be built (line 5) and traversed (lines 13–15).

unordered collection of key/value pairs, or as they are called in JavaScript, *properties* or *slots*. The name of a property can be any string, including the empty string. The value of a property can be any JavaScript expression, including another object; it cannot be `undefined`.

JavaScript allows you to express *object literals* by specifying their properties and values directly, as Figure 6.3 shows. This simple object-literal syntax is the basis of **JSON**, or JavaScript Object Notation, which we introduced in Section 3.6. Despite its name, JSON has become a language-independent way to represent data that can be exchanged between SaaS services or between a SaaS client and server. In fact, lines 2–11 in the figure (minus the trailing semicolon on line 11) are a legal JSON representation. Officially, each property value in a JSON object can be a Number, Unicode String, Boolean (`true` or `false` are the only possible values), `null` (empty value), or a nested Object recursively defined. Unlike full JavaScript, though, in the JSON representation of an object all strings *must* be quoted, so the example in the top row of Figure 6.2 would need quotes around the word `title` to comply with JSON syntax. Figure 6.4 summarizes a variety of tools for checking the syntax and style of both JavaScript code and JavaScript-related data structures and protocols that we'll meet in the rest of this chapter.

JSON.org defines JSON's precise syntax and lists parsing libraries available for other languages.

The fact that a JavaScript object can have function-valued properties is used by well-engineered libraries to collect all their functions and variables into a single *namespace*. For example, as we'll see in Section 6.4, jQuery defines a single global variable `jQuery` through which all features of the jQuery library are accessed, rather than littering the global namespace with the many objects in the library. We will follow a similar practice by defining a small number of global variables to encapsulate all our JavaScript code.

The term *client-side JavaScript* refers specifically to JavaScript code that is associated with HTML pages and therefore runs in the browser. Each page in your app that wants to use JavaScript functions or variables must include the necessary JavaScript code itself. The recommended and unobtrusive way to do this is using a `script` tag referencing the file containing the code, as Figure 6.5 shows. The Rails view helper `javascript_include_tag 'application'`, which generates the above tag, can be placed in your `app/views/layouts/application.html.erb` or other layout template



Name	Tool type	Description
JSLint	Web-based	Copy and paste your code into the form at jslint.com to check it for errors and stylistic pitfalls according to the guidelines in Doug Crockford's <i>JavaScript: The Good Parts</i> . Also checks for legal but unsafe constructions; some developers find it overly pedantic.
JavaScript Lint	Command-line	Matthias Miller's command-line tool, installed by our setup script, reports errors and warnings based on the same JavaScript interpreter used by the Firefox browser. To run it, type <code>jsl -process file.js</code>
Closure	Command-line	Google's source-to-source compiler ⁵ translates JavaScript to better JavaScript, removing dead code and minifying as it goes, and giving errors and warnings. Its associated Linter tool goes even further and enforces Google's JavaScript style guidelines.
YUI	Command-line	Yahoo's YUI Compressor ⁶ minifies JavaScript and CSS more aggressively than some other tools and looks for stylistic problems in the process.
JSONlint	Web-based	This tool at jsonlint.com ⁷ checks your JSON data structures for syntax errors.

Figure 6.4: A variety of tools for debugging your JavaScript code and associated data structures and server interactions. One challenge is that just as with the C language, there are many competing coding guidelines for JavaScript—Google's, Yahoo's, the Node.js project's, and others—and different tools check and enforce different coding styles.

<https://gist.github.com/b23448e3c5ddd3c066ddb3153ffa11e6>

```
1 | <script src="/public/javascripts/application.js"></script>
```

<https://gist.github.com/b6968a45858d79e5f5edb2a76670e1a3>

```
1 | <html>
2 |   <head><title>Update Address</title></head>
3 |   <body>
4 |     <!-- BAD: embedding scripts directly in page, esp. in body -->
5 |     <script>
6 |       <!-- // BAD: "hide" script body in HTML comment
7 |           // (modern browsers may not see script at all)
8 |           function checkValid() {      // BAD: checkValid is global
9 |             if (!fieldsValid(getElementById('addr'))) {
10 |               // BAD: > and < may confuse browser's HTML parser
11 |               alert('>>> Please fix errors & resubmit. <<<');
12 |             }
13 |             // BAD: "hide" end of HTML comment (1.3) in JS comment: -->
14 |           </script>
15 |           <!-- BAD: using HTML attributes for JS event handlers -->
16 |           <form onsubmit="return checkValid()" id="addr" action="/update">
17 |             <input onchange="RP.filter_adult" type="checkbox"/>
18 |             <!-- BAD: URL using 'javascript:' -->
19 |             <a href="javascript:back()">Go Back</a>
20 |           </form>
21 |         </body>
22 |       </html>
```

Figure 6.5: Top: The unobtrusive and recommended way to load JavaScript code in your HTML view(s). Bottom: Three obtrusive ways to embed JavaScript into HTML pages, all deprecated because they mix JavaScript code with HTML markup. Sadly, all are common in the “street JavaScript” found on poorly-engineered sites, yet all are easily avoided by using the `script src=` method and by using the unobtrusive techniques described in the rest of this chapter for connecting JavaScript code to HTML elements.

that is part of every page served by your app. If you then place your code in one or more separate .js files in `app/assets/javascripts`, when you deploy to production Rails will do the following steps automatically:

1. Concatenate the contents of all JavaScript files in this directory;
2. **Minify** the result by removing whitespace and comments, and possibly compress the result;
3. Place the result in a single large file in the `public` subdirectory that will be served directly by the presentation tier with no Rails intervention;
4. Adjust the URLs emitted by `javascript_include_tag` so that the user's browser loads not only your own JavaScript files but also the jQuery library.

This automatic behavior, supported by modern production environments including Heroku, is called the *asset pipeline* and is described more fully in this guide⁸. You might think it wasteful for the user's browser to load a single enormous JavaScript file, especially if only a few pages in your app use JavaScript and any given page only uses a small subset of your JavaScript code. But the user's browser only loads the large file once and then caches it until you redeploy your app with changed .js files. Also, in development mode, the asset pipeline skips the “precompilation” process and just loads each of the JavaScript files separately, since they're likely to be changing frequently while you're developing.

The Rails asset pipeline performs similar operations on CSS files in your project, and can even arrange (with the help of external plug-ins or gems) to have images and other large static assets served from a **content distribution network**.

Summary of Client-Side JavaScript and HTML:

- Like Ruby, JavaScript is interpreted and dynamically typed. The basic object type is a hash with keys that are strings and values of arbitrary type, including other hashes.
- The fundamental JavaScript data type is an object, which is an unordered collection of property names and values, similar to a hash. Since objects can nest, they can represent hierarchical data structures. JavaScript's simple object-literal notation is the inspiration for the JSON data interchange format.
- The preferred unobtrusive way to associate JavaScript with an HTML page is to include in the HTML document's head element a `script` tag whose `src` attribute gives the URL of the script itself, so that the JavaScript code can be kept separate from HTML markup. The Rails helper `javascript_include_tag` generates the correct URL that takes advantage of Rails' asset pipeline.

Self-Check 6.2.1. Is every valid JSON object parsable by JavaScript? If not, give an example of one that isn't.

◊ Yes, every valid JSON object is a valid JavaScript object. Whereas JSON requires quotes around every slot name, JavaScript sometimes does and sometimes doesn't, but it is always safe to use quotes. ■

Self-Check 6.2.2. If we make sure to put slot names in quotes, is every valid JavaScript object a valid JSON object? If not, give an example of one that isn't.

◊ No, even if all the slot names are quoted, some JavaScript objects are *not* valid JSON. For

example, if one of the object's slots is a function, that object would not be valid JSON, since JSON slot values are limited to simple types (numbers, strings, Booleans) and collections (arrays or other JSON objects). ■

6.3 Classes, Functions and Constructors

In Chapter 2 we mentioned that object-orientation and class inheritance are distinct language design concepts, although many people mistakenly conflate them because popular languages like Java use both. While JavaScript is object-oriented and supports inheritance, *it does not have classes*, despite the addition of a new `class` keyword in the ECMAScript 6 standard⁹. However, classes have *not* been added to JavaScript; the keyword is ***syntactic sugar*** for JavaScript's built-in mechanism of ***prototype inheritance***, in which every object inherits from some prototype object and delegates to its prototype any slot lookup that fails on the object itself.

Unfortunately, the design of this mechanism has led to confusion for newcomers to JavaScript, especially regarding the behavior of the keyword `this`. We will concern ourselves with three common uses of `this`. In this section we introduce the first two of these uses, and an associated pitfall. In Section 6.6 we introduce the third use.

Lines 1–8 of Figure 6.6 show a function called `Movie`. This syntax for defining functions may be unfamiliar, whereas the alternate syntax in lines 9–11 looks comfortably familiar. Nonetheless, we will use the first syntax for two reasons. First, unlike Ruby, functions in JavaScript are true ***first-class objects***—you can pass them around, assign them to variables, and so on. The syntax in line 1 makes it clear that `Movie` is simply a variable whose value happens to be a function. Second, although it's not obvious, the variable `Movie` in line 9 is being declared in JavaScript's global namespace—hardly beautiful. In general we want to minimize clutter in the global namespace, so we will usually create one or a few objects named by global variables associated with our app, and all of our JavaScript functions will be the values of properties of those objects.

Prototype inheritance comes from the experimental language Self developed at the legendary Xerox PARC (Palo Alto Research Center), where many technologies that define personal computing were invented, and appeared in the NewtonScript language that powered Apple's first ill-fated "personal digital assistant."

If we call the `Movie` function using JavaScript's `new` keyword (line 13), the value of `this` in the function body will be a new JavaScript object that will eventually be returned by the function, similar to Ruby's `self` inside an `initialize` constructor method. In this case, the returned object will have properties `title`, `year`, `rating`, and `full_title`, the last of which is a property whose value is a function. If line 14 looks like a function call to you, then you've been hanging around Ruby too long; since functions are first-class objects in JavaScript, this line just returns the value of `full_title`, which is the function itself, not the result of calling it! To actually call it, we need to use parentheses, as in line 15. When we make that call, within the body of `full_title`, `this` will refer to the object whose property the function is, in this case `pianist`.

Check your browser's documentation for how to display its built-in JavaScript console, where you can try these examples interactively.

Remember, though, that while these examples look just like calling a class's constructor and calling an instance method in Ruby, JavaScript has no concept of classes or instance methods. In fact, there is nothing about a particular JavaScript function that makes it a constructor; instead, it's the use of `new` when calling the function that makes it a constructor, causing it to create and return a new object. The reason this works is because of JavaScript's ***prototype inheritance*** mechanism, which we don't discuss further (but see the Elaboration below to learn more). Nonetheless, forgetting this subtle distinction may confuse you when you expect class-like behaviors and don't get them.

However, a JavaScript misfeature can trip us up here. It is (unfortunately) perfectly legal

```
https://gist.github.com/e6bf83dabd38ebde2a25e22b0a98957b
1 let Movie = function(title,year,rating) {
2   this.title = title;
3   this.year = year;
4   this.rating = rating;
5   this.full_title = function() { // "instance method"
6     return(this.title + ' (' + this.year + ')');
7   };
8 };
9 function Movie(title,year,rating) { // this syntax may look familiar...
10   // ...
11 }
12 // using 'new' makes Movie the new objects' prototype:
13 pianist = new Movie('The Pianist', 2002, 'R');
14 pianist.full_title; // => function() {...}
15 pianist.full_title(); // => "The Pianist (2002)"
16 // BAD: without 'new', 'this' is bound to global object in Movie call!!
17 juno = Movie('Juno', 2007, 'PG-13'); // DON'T DO THIS!!
18 juno; // undefined
19 juno.title; // error: 'undefined' has no properties
20 juno.full_title(); // error: 'undefined' has no properties
```

Figure 6.6: Since functions are first-class objects, it is fine for an object to have a property whose value is a function, as `full_title` is. We will make extensive use of this characteristic. Note the pitfall in lines 14–18.

to call `Movie` as a plain old function *without* using the `new` keyword, as in line 17. If you do this, JavaScript’s behavior is completely different in two horrible, horrible ways. First, in the body of `Movie`, `this` will not refer to a brand-new object but instead to the *global object*, which defines various special constants such as `Infinity`, `NaN`, and `null`, and supplies various other parts of the JavaScript environment. When JavaScript is run in a browser, the global object happens to be a data structure representing the browser window. Therefore, lines 2–5 will be creating and setting new properties of this object—clearly not what we intended, but unfortunately, when `this` is used in a scope where it would otherwise be `undefined`, it refers to the global object, a serious design defect in the language. (See Fallacies and Pitfalls and To Learn More if you want to learn about the reasons for this odd behavior, a discussion of which is beyond the scope of this introduction to the language.)

Second, since `Movie` doesn’t explicitly return anything, its return value (and therefore the value of `juno`) will be `undefined`. Whereas a Ruby function returns the value of the last expression in the function by default, a JavaScript function returns `undefined` unless it has an explicit `return` statement. (The `return` in line 6 belongs to the `full_title` function, not to `Movie` itself.) Hence, lines 19–20 give errors because we’re trying to reference a property (`title`) on something that isn’t even an object.

You can avoid this pitfall by rigorously following the widespread JavaScript convention that a function’s name should be capitalized if and only if the function is intended to be called as a constructor using `new`. Functions that are not “constructor-like” should be given names beginning with lowercase letters.

Node.js provides a different global object, called `global`, that provides these values. The JavaScript identifier `globalThis` always refers to the global object, whatever it happens to be named.



Summary: Functions and Constructors

- JavaScript functions are first-class objects: they can be assigned to variables, passed to other functions, or returned from functions.
- Although JavaScript doesn't have classes, one way of managing namespaces in an orderly way in JavaScript is to store functions as object properties, allowing a single object (hash) to collect a set of related functions as a class would.
- If the `new` keyword is used to call a function, `this` in the function body will refer to a new object whose property values can be initialized in the “constructor.” This mechanism is similar to creating new instances of a class, though JavaScript lacks classes.
- However, if a function is called *without* the `new` keyword, `this` in the function body will refer to the global object, which is almost never what you wanted, and the function will return `undefined`. To avoid this pitfall, capitalize the names of constructor-like functions intended to be called with `new`, but don't capitalize the names of any other functions.
- JavaScript has no classes; the `new` keyword `class` in ES6 does not introduce any new mechanisms into JavaScript, but is syntactic sugar to make the syntax for prototype-based inheritance look more similar to the syntax for defining classes in class-based languages.

■ Elaboration: Prototypal inheritance

Every JavaScript object inherits from exactly one prototype object—new strings inherit from `String.prototype`, new arrays from `Array.prototype`, and so on, up to `Object` (the empty object). If you look up a property on an object that doesn't have that property, its prototype is checked, then its prototype's prototype, and so on until one of the prototypes responds with the property or `undefined` is returned. Given this background, the effect of calling a function using the `new` keyword is to create a new object whose prototype is the same as the function's prototype.

Prototypes come from Self, a language originally designed at the legendary Xerox PARC and which heavily influenced **NewtonScript**, the programming language for the ill-fated Apple Newton “handheld.” Proper use of prototypal inheritance affords an effective kind of implementation reuse that is different from what classes provide. Unfortunately, as Crockford notes in *JavaScript: The Good Parts* (Crockford 2008), JavaScript's implementation of prototypal inheritance is halfhearted and uses a confusing syntax, perhaps in an effort to resemble “classical” languages with class inheritance.

Self-Check 6.3.1. What is the difference between evaluating `square.area` and `square.area()` in the following JavaScript code?

<https://gist.github.com/cc72ba2ff92c2f387ebac8ab56e54af7>

```
1 let square = {
2   side: 3,
3   area: function() {
4     return this.side*this.side;
5   }
6 };
```

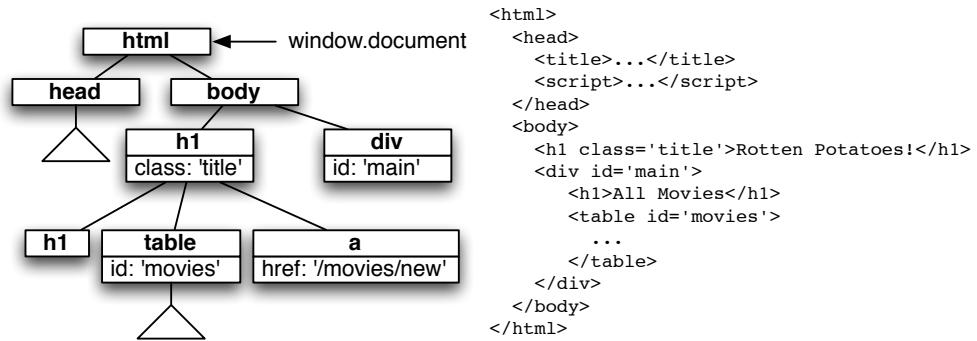


Figure 6.7: A simplified view of the DOM tree corresponding to the RottenPotatoes “list of movies” page with skeletal HTML markup. An open triangle indicates places where we’ve elided the rest of the subtree for brevity. `this.document` is set to point to the DOM tree’s root when a page is loaded.

- ◊ `square.area()` is a function call that in this case will return 9, whereas `square.area` is an unapplied function object. ■

Self-Check 6.3.2. Given the code in Self-Check 6.3.1, explain why it’s is incorrect to write `s=new square`.

- ◊ `square` is just an object, not a function, so it cannot be called as a constructor (or at all). ■

Self-Check 6.3.3. In Ruby, when a method call takes no arguments, the empty parentheses following the method call are optional. Why wouldn’t this work in JavaScript?

- ◊ Because JavaScript functions are first-class objects, a function name without parentheses would be an expression whose value is the function itself, rather than a call to the function. ■

6.4 The Document Object Model (DOM) and jQuery

The World Wide Web Consortium Document Object Model (W3C DOM)¹⁰ is “a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents”—in other words, a standard representation of an HTML, XML, or XHTML document consisting of a hierarchy of elements. A DOM element is recursively defined in that one of its properties is an array of child elements, as Figure 6.7 shows. Hence a DOM node representing the `<html>` element of an HTML page is sufficient to represent the whole page, since every element on a well-formed page is a descendant of `<html>`. Other DOM element properties correspond to the HTML element’s attributes (`href`, `src`, and so on). When a browser loads a page, the HTML of the page is parsed into a DOM tree similar to Figure 6.7.

DOM technically refers to the standard itself, but developers often use it to mean the specific DOM tree corresponding to the current page.

How does JavaScript get access to the DOM? When JavaScript is embedded in a browser, the global object, named by the global variable `window`, defines additional browser-specific properties and functions, collectively called the JSAPI. Whenever a new page is loaded, a new global `window` object is created that shares no data with the global objects of other visible pages. One of the properties of the global object is `window.document`, which is the root element of the current document’s DOM tree and also defines some functions to query,

traverse, and modify the DOM; one of the most common is `getElementById`, which you may have run across while perusing others' JavaScript code.

However, to avoid compatibility problems stemming from different browsers' implementations of the JSAPI, we will bypass these native JSAPI functions entirely in favor of jQuery's more powerful "wrappers" around them. jQuery also adds additional features and behaviors absent from the native JSAPIs, such as animations and better support for CSS and AJAX (Section 6.7). jQuery defines a global function `jQuery()` (aliased as `$()`) that, when passed a CSS selector, returns all of the current page's DOM elements matching that selector. For example, `jQuery('#movies')` or `$('#movies')` would return the single element whose ID is `movies`, if one exists on the page; `$('.h1.title')` would return all the `h1` elements whose CSS class is `title`. A more general version of this functionality is `.find(selector)`, which only searches the DOM subtree rooted at the target. To illustrate the distinction, `'p span'` finds *any* `span` element that is contained inside a `p` element, whereas if `elt` already refers to a *particular* `p` element, then `elt.find('span')` only finds `span` elements that are descendants of `elt`.

Whether you use `$()` or `find`, the return value is a node set (collection of one or more elements) matching the selector, or `null` if there were no matches. Each element is "wrapped" in jQuery's DOM element representation, giving it abilities beyond the browser's built-in JSAPI. From now on, we will refer to such elements as "jQuery-wrapped" elements, to distinguish them from the representation that would be returned by the browser's native JSAPI.

In particular, you can do various things with jQuery-wrapped elements in the node set, as Figure 6.8 shows:

- To change an element's visual appearance, define CSS classes that create the desired appearances, and use jQuery to add or remove CSS class(es) from the element at runtime.
- To change an element's content, use jQuery functions that set the element's HTML or plain text content.
- To animate an element (show/hide, fade in/out, and so on), invoke a jQuery function on that element that manipulates the DOM to achieve the desired effect.

The call
`jQuery.noConflict()`
"undefines" the `$` alias, in
case your app uses the
browser's built-in `$` (usually
an alias for `document.getElementById`) or
loads another JavaScript
library that also tries to
define `$`.

Some jQuery developers
use `$` to prefix variable
names that refer to
jQuery-wrapped objects, as
in `var $h1=$('#h1')`.

Note, however, that even when a node set includes multiple matching elements, it is not a JavaScript array and you cannot treat it like one: you cannot write `'tr')[0]` to select the first row of a table, even if you first call jQuery's `toArray()` function on the node set. Instead, following the Iterator design pattern, jQuery provides an `each` iterator defined on the collection that returns one element at a time while hiding the details of how the elements are stored in the collection, just as `Array#each` does in Ruby.

Screencast 6.4.1 shows some simple examples of these behaviors from the browser's JavaScript console. We will use these to implement the features of Screencast 6.1.1.

Screencast 6.4.1: Manipulating the DOM with jQuery.

<http://youtu.be/kwdYHc5M0Ac>

jQuery makes it easy to manipulate the DOM from JavaScript and provides a built-in library of useful visual effects. These simple examples show that JavaScript can not only read element and content information on the page, but also modify the elements, causing the browser to redraw them. This behavior is the key to client-side JavaScript.

Property or function, example	Value/description
<code>\$(dom-element)</code> <code>\$(this)</code>	Returns a set of jQuery-wrapped DOM element(s) specified by the argument, which can be a CSS3 selector (such as <code>'span.center'</code> or <code>'#main'</code>), the element object returned by the browser's <code>getElementById</code> function, or in an event handler, the element that received the event, named by <code>this</code> . The return value of this function is suitable as the target for any of the below calls. (Recall that the term <i>target</i> is used in JavaScript the way <i>receiver</i> is used in Ruby.)
<code>is(cond)</code>	Test if the element is <code>':checked'</code> , <code>':selected'</code> , <code>'enabled'</code> , <code>'disabled'</code> . Note that these strings were chosen to resemble Ruby symbols, though JavaScript doesn't have symbols.
<code>addClass()</code> , <code>removeClass()</code> , <code>hasClass()</code>	Shortcuts for manipulating the <code>class</code> attribute: add or remove the specified CSS class (a string) from the element, or test whether the given class is currently associated with the element.
<code>insertBefore()</code> , <code>insertAfter()</code>	Insert the target element(s) before or after the argument. That is, <code>newElt.insertBefore(existingElt)</code> inserts <code>newElt</code> just before <code>existingElt</code> , which must exist.
<code>remove()</code>	Remove the target element(s) from the DOM.
<code>replaceWith(new)</code>	Replace the target element(s) with the <code>new</code> element(s) provided.
<code>clone()</code>	Return a complete copy of the target element, recursively cloning its descendants.
<code>html()</code> , <code>text()</code>	Return (with no argument) or set (with one argument) the element's complete HTML content or plain-text content. If the element has other elements nested inside it, you can replace its HTML with nested elements but don't have to, but replacing its text will obliterate the nested elements.
<code>val()</code>	Return (with no argument) or set (with one argument) the current value of the element. For text boxes, value is the current string contents; for buttons, the button's label; for select menus, the text of the currently selected value.
<code>attr(attr,[newval])</code> <code>\$('img').attr('src', 'http://imgur.com/xyz')</code>	Return or (with second argument) set the value of the given attribute on the element.
<code>hide(duration,callback)</code> , <code>show()</code> , <code>toggle()</code> <code>slideUp()</code> , <code>slideDown()</code> , <code>slideToggle()</code> <code>fadeOut()</code> , <code>fadeIn()</code> , <code>fadeTo(duration,target,callback)</code>	Hide or show elements selected by the target. Optional <code>duration</code> is one of <code>'fast'</code> , <code>'slow'</code> , or the integer number of milliseconds that the animation should last. Optional <code>callback</code> is a function to call when animation completes. Other sets of animations with same arguments include <code>slideDown/slideUp</code> / <code>slideToggle</code> and <code>fadeOut/fadeIn</code> . For <code>fadeTo</code> , second argument is target opacity, from 0.0 (transparent) to 1.0 (opaque).
<code>evt(func)</code> <code>\$('li').click(function(){ \$(this).hide(); })</code>	Set <code>func</code> as the handler for event <code>evt</code> on the element(s) selected by the target. <code>func</code> can be an anonymous function or a named function. See Figure 6.9 for some of the most important event types.

Figure 6.8: Some attributes and functions defined on jQuery's enhanced DOM element objects; they should be called with the appropriate element or collection of elements as the target of the call (like receiver in Ruby). Functions that only make sense applied to a single element, such as `attr`, apply to the first element when used on a collection of elements. Functions that can both read and modify element properties act as getters when the final (or only) argument is absent, and setters when it's present. Unless otherwise noted, all functions return their target, so calls can be chained, as in `elt.insertBefore(...).hide()`. See the jQuery documentation¹² for more features beyond this subset.

Finally, as we will see, the `jQuery()` or `$()` function is **overloaded**: its behavior depends on the number and types of arguments with which it's called. In this section we introduced just one of its four behaviors, namely for selecting elements in the DOM; we will soon see the others.

Summary of the DOM and jQuery:

- The World Wide Web Consortium Document Object Model (W3C DOM) is a language-independent representation of the hierarchy of elements that constitute an HTML document.
- All JavaScript-enabled browsers provide JavaScript language bindings to access and traverse the DOM. This set of functionality, together with JavaScript access to other browser features, is collectively called the JavaScript Application Programming Interface or JSAPI.
- The powerful jQuery library provides a uniform adapter to browsers' differing JS APIs and adds many enhanced functions such as CSS-based DOM traversal, animation, and other special effects.

■ *Elaboration: To jQuery or not to jQuery?*

jQuery has evolved over many years, and many packages now build on it, including the Bootstrap front-end framework. As of this writing, Bootstrap version 5 is aiming to eliminate jQuery as a dependency by rewriting the code that relies on it using “plain JavaScript” and browsers’ native JS APIs. Some developers see this as a welcome change: it eliminates a dependency on legacy code, eliminates some cases where jQuery and Bootstrap CSS styles conflict with each other, and reduces the total footprint of Bootstrap. Others question its utility since the amount of code actually saved is not very much by today’s standards, and re-creating jQuery’s well-tested functionality from scratch is hardly DRY and is likely to introduce new bugs. This debate is a modern example of the ever-present tension between continuing to rely on well-tested legacy code versus reimplementing a subset of the legacy code’s functionality from scratch as other frameworks evolve.

Self-Check 6.4.1. *Why is `this.document`, when it appears outside the scope of any function, equivalent to `window.document`?*

- ◊ Outside of any function, the value of `this` is the global object. When JavaScript runs in a Web browser, the global object is the `window` object. ■

Self-Check 6.4.2. *True or false: even after the user closes a window in her Web browser, the JavaScript code associated with that window can still access and traverse the HTML document the window had been displaying.*

- ◊ False. Each new HTML document gets its own global object and DOM, which are destroyed when the document’s window is closed. ■

6.5 The DOM and Accessibility

Many people navigate a website by clicking on buttons or links. This relies on using a mouse or touchscreen as well as being able to perceive the visual affordances of the web page. Ac-

cessibility, sometimes abbreviated *a11y*, is about ensuring that your users can access your applications using different input methods. For example, some users may need (or prefer) to navigate a website using the keyboard, so they must have a way of “clicking” on a button, perhaps by pressing the space bar on their keyboards. Blind or low-vision users may use **screen reader** software or other **assistive technology** that translates text and interactions on screen into audio, and to include these users you must ensure there’s a way for the computer to generate a text-based description of your page elements. The `alt` attribute of an `img` element, which has been part of HTML since the beginning, is one simple example of providing a textual alternative to visual element.

Section 1.8 motivated the use of HTML/CSS frameworks. One advantage of a good framework is that it provides some built-in support for using the techniques we describe in this section to improve your app’s accessibility: using Semantic HTML to structure your pages, and adding ARIA (Accessible Rich Internet Applications) attributes to give hints to screen readers.

Semantic HTML. HTML is very flexible: in principle you can accomplish just about any visual styling using only `div` (“divider”) elements with appropriate CSS rules. However, a `div` doesn’t convey much about the purpose of that element. *Semantic HTML* involves choosing the *most appropriate* HTML element type tag that describes the element’s logical role on the page, so that (for example) a screen reader can convey that a particular element is a button, heading, paragraph, or maybe even a timestamp. Beyond accessibility, using the proper HTML element type communicates your intentions to other developers on the project, and allows search engines to improve their search results. We distinguish three categories of HTML elements: structural, content, and interactive.

The HTML Specification¹³ is constantly evolving and is the definitive list of existing elements.

Structural elements break down large and often visually-distinct sections of a web page, allowing screen readers to quickly jump between sections:

- `h1`...`h6` are elements that give your page an outline. Most pages start with `h1`.
- `nav` is used for sections of links that direct users to different parts of your website. Commonly, this is used in a “navigation bar” at the top of a web page.
- `main` This is for the body of your web page. There should only be one of these per page.
- `header` and `footer` These are common sections that you might include. They should be *outside* of the main content.
- `section` should be used to group elements into a larger component, like a toolbar.

Content elements give meaningful descriptions to content that is primarily designed to be read, such as text and graphics:

- `p` is for a “paragraph”, which can include other elements inside, like `img`, `a`, and so on.
- `strong` and `em` for bold and italic text, respectively.
- `time` is useful for dates and times, and many browsers give users additional features, like the ability to easily create a calendar link.

Interactive elements are those with which users interact, so they are perhaps the most important to target for accessibility.

- a (“anchor”) links to another page or another part of the same page.
- button is the main way to take some action. *When you use a button, the browser automatically provides functions for keyboard accessibility, as well as describing the button as “clickable” to a screen reader.*
- input is the primary method for accepting user input, such as within a form. inputs have a type attribute that can take on one of 22 different values¹⁴ such as email, password, checkbox, and so on.
- select builds simple dropdown lists.

Buttons and inputs come with quite a few default features that are critical for accessibility. However, each of these components can be controlled individually.

- *Focus:* An interactive element needs to be able to “receive focus”, which means that when a user is navigating a web page with a keyboard, they are able to take actions on a focused element. (This is analogous to how the styling of an element might change when you hover over it.) Focusability is controlled by the tabindex attribute. Native elements like a button have a property tabindex="0" internal to them. If you’re trying to make a span interactive, you’ll need to take care to manually set this.
- *Keyboard Actions:* When a button or link has focus, a button is activated when Enter or Space is pressed, but links are followed only when Enter is pressed. When adding interactivity to other element types, take care to add an onkeypress event handler that responds to the appropriate keys. The keypress handler will often call the onclick handler, mirroring the behavior of the built-in button.
- *Visual Focus:* Using the :focus CSS pseudo-selector allows styling an element differently when it has focus, so that when users tab with the keyboard they can see which element is active.

A corollary of the Semantic HTML guideline of using the most appropriate element for a given task is that you shouldn’t deliberately make non-interactive elements interactive unless you have no other alternatives, such as having a plain p (paragraph) element respond to clicks. Instead, find a way to achieve your goal using existing interactive elements such as buttons and links.

Going beyond using Semantic HTML to select the right elements, we can further improve accessibility using **WAI-ARIA**, or simply ARIA: The Web Accessibility Initiative/Accessible Rich Internet Applications specification¹⁵ stewarded by the World Wide Web Consortium (W3C), which specifies how to increase the accessibility of web pages, in particular those enhanced by JavaScript.

The first rule of ARIA is “Don’t use ARIA”—that is, try to structure your HTML and JavaScript so that the additional support ARIA provides is unnecessary. ARIA is a purely *descriptive* tool: Adding ARIA attributes to an element will not change how the element works, but only how it is read aloud by the screen reader. ARIA applies to both interactive and non-interactive elements.

In ARIA, every HTML element has a *role* and a *label*. The ARIA specification defines the allowed roles¹⁶. Many roles such as link and button follow from the HTML element type

itself, but others describe more complex UI elements such as “tooltip” that do not correspond directly to a specific HTML element type.

The label, sometimes called an *accessible name*, is typically generated from the visual text of the element, and sometimes includes the role. For example a button (`<button>Submit</button>`) might be read as “Submit, button” by a screen reader. But when there is no text content in an interactive element, such as button whose content is an icon, for accessibility you *must* provide a label using ARIA. For example, adding the attribute `aria-label="add to cart"` to an element will cause the screen reader to read the phrase “add to cart.” Labels should be detailed enough to convey the element’s purpose, but as concise as possible.

Finally, in this chapter we’ve seen many examples of how JavaScript can “live-update” the page’s DOM (and therefore the rendered HTML) after the page has been loaded, even if the user doesn’t do anything, as in the case of a page change triggered by a timer event. But if a page doesn’t specify when and where content might be updated, then screen-reader users might not realize something has changed. Adding the `aria-live` attribute to an element tells a screen reader to notify the user when the content in that element changes. Most often, you should use `aria-live="polite"`, which tries to minimize interruptions to the user, but in special cases, you can use `aria-live="assertive"` to convey an urgent message.



Although mature HTML/CSS frameworks such as Bootstrap include appropriate ARIA notations on the elements and components they provide, accessibility is an ongoing concern during app development. The Web Content Accessibility Guidelines¹⁷ WCAG are the requirements that applications need to follow to be considered legally accessible in many jurisdictions. The axe browser extension¹⁸, available for Chrome and Firefox, runs numerous checks for WCAG 2 compliance on any displayed page. Similarly, the W3C maintains the ARIA Authoring Practices¹⁹ specification, which contains dozens of examples (with working JavaScript) for common UI patterns across applications.

Summary of DOM and Accessibility:

- Accessibility, sometimes abbreviated a11y, is about ensuring that your users can interact with your app using a variety of input and output methods.
- One way to make the page friendly to assistive technologies that accomplish this is to clearly identify the logical role of each element in the page, not just its visual appearance.
- Semantic HTML categorizes elements into structural (describe page structure), content (generic text, images, specialized text such as dates and times), or interactive (allows user input). The page’s logical structure and the roles of different portions of the page are suggested by appropriate choices of HTML elements during page authoring.
- When Semantic HTML is not enough, such as for components (logical structures on a page made up of multiple elements) or JavaScript-enhanced UI effects, the ARIA (Accessible Rich Internet Applications) standard provides ways to attach additional attributes to elements that cue a11y behaviors in assistive software.

Events on arbitrary elements	<code>click</code> , <code>dblclick</code> , <code>mousedown/mouseup</code> , <code>mouseenter/mouseleave</code> , <code>keypress</code> (event. <code>which</code> gives the ASCII code of the key pressed) <code>focus/blur</code> (element gains/loses focus), <code>focusin/focusout</code> (parent gains/loses focus)
Events on user-editable controls (forms, checkboxes, radio buttons, text boxes, text fields, menus)	<code>change</code> fires when any control's state or content is changed. <code>select</code> (user selects text; string event. <code>which</code> is selected text) <code>submit</code> fires when the user attempts to submit the form by any means.

Figure 6.9: A few of the JavaScript events defined by the jQuery API. Set a handler for an event with `element.on('evt', func)` or as a shortcut, `element.evt(func)`. Hence, `$(‘h1’).on(‘click’, function() {...})` is equivalent to `$(‘h1’).click(function() {...})`. The callback `func` will be passed an argument (which you’re free to ignore) whose value is the jQuery Event object describing the event that was triggered. Remember that `on` and its shortcuts will bind the callback to *all* elements matching the selector, so be sure the selector you pass is unambiguous, for example by identifying an element by its ID.

6.6 Events and Callbacks

So far all of our DOM manipulation has been by typing JavaScript commands directly. As you’ve no doubt guessed, much more interesting behaviors are possible when DOM manipulation can be triggered by user actions. As part of the JSAPI for the DOM, browsers allow attaching JavaScript *event handlers* to the user interface: when the user performs a certain UI action, such as clicking a button or moving the mouse into or out of a particular HTML element, you can designate a JavaScript function that will be called and have the opportunity to react. This capability makes the page behave more like a desktop UI in which individual elements respond visually to user interactions, and less like a static page in which any interaction causes a whole new page to be loaded and displayed.

Figure 6.9 summarizes the most important events defined by the browser’s native JSAPI and improved upon by jQuery. While some are triggered by user actions on DOM elements, others relate to the operation of the browser itself or to “pseudo-UI” events such as form submission, which may occur via clicking a Submit button, pressing the Enter key (in some browsers), or another JavaScript callback causing the form to be submitted. To attach a behavior to an event, simply provide a JavaScript function that will be called when the event *fires*. We say that this function, called a **callback** or *event handler*, is *bound* to that event on that DOM element. Although events are automatically triggered by the browser, you can also trigger them yourself: for example, `e.trigger(‘click’)` triggers the `click` event handler for element `e`. As we will see in Section 6.8, this ability is useful when testing: you can simulate user interaction and check that the correct changes are applied to the DOM in response to a UI event.

Browsers define built-in behavior for some events and elements: for example, clicking on a link visits the linked page. If such an element also has a programmer-supplied `click` handler, the handler runs first; if the handler returns a truthy value (Figure 6.2), the built-in behavior runs next, but if the handler returns a falsy value, the built-in behavior is suppressed. What if an element has *no* handler for a user-initiated event, as is the case for images? In that case, its parent element in the DOM tree is given the chance to respond to the event handler. For example, if you click on an `img` element inside a `div` and the `img` has no `click` handler, then the `div` will receive the `click` event. This process continues until some element handles the event or it “bubbles” all the way up to the top-level `window`, which may or may not have a built-in response depending on the event.

The less precise term Dynamic HTML was sometimes used in the past to refer to the effects of combining JavaScript-based DOM manipulation and CSS.

```
https://gist.github.com/d6d0e7d7554f93f952d0200d3121fa29
1 let MovieListFilter = {
2   filter_adult: function () {
3     // 'this' is *unwrapped* element that received event (checkbox)
4     if ($(this).is(':checked')) {
5       $('.adult').hide();
6     } else {
7       $('.adult').show();
8     };
9   },
10  setup: function() {
11    // construct checkbox with label
12    let labelAndCheckbox =
13      `(<label for="filter">Only movies suitable for children</label>' +
14      '<input type="checkbox" id="filter"/>)`;
15    labelAndCheckbox.insertBefore('#movies');
16    $('#filter').change(MovieListFilter.filter_adult);
17  }
18 }
$(MovieListFilter.setup); // run setup function when document ready
```

Figure 6.10: Using jQuery to add a “filter movies” checkbox to RottenPotatoes’ list of movies page; put this code in `app/assets/javascripts/movie_list_filter.js`. The text walks through the example in detail, and additional figures in the rest of the chapter generalize the techniques shown here. Our examples use jQuery’s DOM manipulation features rather than the browser’s built-in ones because the jQuery API is more consistent across different browsers than the official W3C DOM specification.

Our discussion of events and event handlers motivates the third common use of JavaScript’s `this` keyword (recall that Section 6.3 introduced the first two uses). When an event is handled, in the body of the event handler function, jQuery will arrange for `this` to refer to the element to which the handler is attached (which may not be the element that originally received the event, if the event “bubbled up” from a descendant). However, if you were programming *without* jQuery, the value of `this` in an event handler is the global object (`document.window`), and you have to examine the event’s data structure (usually passed as the final argument to the handler) to identify the element that handled the event. Since handling events is such a common idiom, and most of the time an event handler wants to inspect or manipulate the state of the element on which the event was triggered, jQuery is written to explicitly set `this` to that DOM element.

Putting all these pieces together, Figure 6.10 shows the client-side JavaScript to implement a checkbox that, when checked, will hide any movies with ratings other than G or PG. Our general strategy for JavaScript can be summarized as:

1. Identify the DOM elements we want to operate on, and make sure there is a convenient and unambiguous way of selecting them using `$()`.
2. Create the necessary JavaScript functions to manipulate the elements as needed. For this simple example we can just write them down, but as we’ll see in Section 6.8, for AJAX or more complex functions we will use TDD (Chapter 8) to develop the code.
3. Define a setup function that binds the appropriate JavaScript functions to the elements and performs any other necessary DOM manipulation.
4. Arrange to call the setup function once the document is loaded.

For Step 1, we modify our existing Rails movie list view to attach the CSS class `adult` to any table rows for movies rated other than G or PG. All we have to do is change line 12

of the Index template (Figure 4.5) as follows, thereby allowing us to write `$('.adult')` to select those rows:

<https://gist.github.com/c857168c2c367c4cd1649efb3c643c5c>

```
1 | <div class="row<%= ('adult' unless movie.rating =~ /G|PG$/) %>">
```

For Step 2, we provide the function `filter_adult`, which we will arrange to be called whenever the checkbox is checked or unchecked. As lines 4–8 of Figure 6.10 show, if the checkbox is checked, the adult movie rows are hidden; if unchecked, they are revealed. Recall from Figure 6.8 that `:checked` is one of jQuery’s built-in behaviors for checking the state of an element. Remember also that jQuery selectors such as `$('.adult')` generally return a collection of matching elements, and actions like `hide()` are applied to the whole collection.

Why does line 4 refer to `$(this)` rather than just `this`? The mechanism by which user interactions are dispatched to JavaScript functions is part of the browser’s JSAPI, so the value of `this` is the *browser’s* representation of the checkbox (the element that handled the event). In order to use the more powerful jQuery features such as `is(':checked')`, we have to “wrap” the native element as a jQuery element by calling `$` on it in order to give it these special powers. The first row of Figure 6.12 shows this usage of `$`.

For Step 3, we provide the `setup` function, which does two things. First, it creates a label and a checkbox (lines 12–14), using the `$` mechanism shown in the second row of Figure 6.12, and inserts them just before the `movies` table (line 15). Again, by creating a jQuery element we are able to call `insertBefore` on it, which is not part of the browser’s built-in JSAPI. Most jQuery functions such as `insertBefore` return the target object itself, allowing “chaining” of function calls as we’ve seen in Ruby.

Second, the `setup` function binds the `filter_adult` function to the checkbox’s `change` handler. You might have expected to bind to the checkbox’s `click` handler, but `change` is more robust because it’s an example of a “pseudo-UI” event: it fires whether the checkbox was changed by a mouse click, a keypress (for browsers that have keyboard navigation turned on, such as for users with disabilities that prevent use of a mouse), or even by other JavaScript code. The `submit` event on forms is similar: it’s better to bind to that event than to bind to the `click` handler on the form-submit button, in case the user submits the form by hitting the Enter key.

Why didn’t we just add the label and checkbox to the Rails view template? The reason is our design guideline of graceful degradation: by using JavaScript to create the checkbox, legacy browsers will not render the checkbox at all. If the checkbox was part of the view template, users of legacy browsers would still see the checkbox, but nothing would happen when they clicked on it.

Why does line 16 refer to `MovieListFilter.filter_adult`? Couldn’t it just refer to `filter_adult`? No, because that would imply that `filter_adult` is a variable name visible in the scope of the `setup` function, but in fact it’s not a variable name at all—it’s just a function-valued property of the object `MovieListFilter`, which *is* a (global) variable. It is good JavaScript practice to create one or a few global objects to “encapsulate” your functions as properties, rather than writing a bunch of functions and polluting the global namespace with their names.

The last step is Step 4, which is to arrange for the `setup` function to be called. For historical reasons, JavaScript code associated with a page can begin executing *before* the entire page has been loaded and the DOM fully parsed. This feature was more important for responsiveness when browsers and Internet connections were slower. Nonetheless, we usually want to wait until the page is finished loading and the entire DOM has been parsed,

<code>var m = new Movie();</code>	(Figure 6.6, line 13) In the body of the <code>Movie</code> function, <code>this</code> will be bound to a new object that will be returned from the function, so you can use <code>this.title</code> (for example) to set its properties. The new object's prototype will be the same as the function's prototype.
<code>pianist.full_title();</code>	(Figure 6.6, line 15) When <code>full_title</code> executes, <code>this</code> will be bound to the object that “owns” the function, in this case <code>pianist</code> .
<code>\$('#filter').change(MovieListFilter.filter_adult);</code>	(Figure 6.10, line 16) When <code>filter_adult</code> is called to handle a <code>change</code> event, <code>this</code> will refer to the element on which the handler was bound, in this case one of the element(s) matching the CSS selector <code>#filter</code> .

Figure 6.11: The three common uses of `this` introduced in Sections 6.3 and 6.6. See Fallacies and Pitfalls for more on the use and misuse of `this`.

Uses of <code>\$()</code> or <code>jQuery()</code> with example	Value/side effects, line number in Figure 6.10
<code>\$(sel)</code> <code>\$('.mov span')</code>	return collection of jQuery-wrapped elements selected by CSS3 selector <code>sel</code> (line 16)
<code>\$(elt)</code> <code>\$(this), \$(document),</code> <code>\$(document.getElementById('main'))</code>	When an element is returned by a JSAPI call such as <code>getElementById</code> or supplied to an event-handler callback, use this function to create a jQuery-wrapped version of the element, on which you can call the operations in Figure 6.8 (line 4)
<code>\$(HTML[, attrs])</code> <code>'<p>boldwords</p>',</code> <code>'', {</code> <code>src: '/rp.gif',</code> <code>click: handleClick }</code>	Returns a new jQuery-wrapped HTML element corresponding to the passed text, which must contain at least one HTML tag with angle brackets (otherwise jQuery will think you’re passing a CSS selector and calling it as in the previous table row). If a JavaScript object is passed for <code>attrs</code> , it is used to construct the element’s attributes. (Lines 13–14) The new element is not automatically inserted into the document; Figure 6.8 shows some methods for doing that, one of which is used in line 15.
<code>\$(func)</code> <code>\$(function () {...});</code>	Run the provided function once the document has finished loading and the DOM is ready to be manipulated. This is a shortcut for <code>\$(document).ready(func)</code> , which is itself a jQuery wrapper around the <code>onLoad()</code> handler of the browser’s built-in JSAPI. (line 19)

Figure 6.12: The four ways to invoke the overloaded function `jQuery()` or `$()` and the effects of each. All four are demonstrated in Figure 6.10.

or else we might be trying to bind callbacks on elements that don’t exist yet! Line 19 does this, adding `MovieListFilter.filter_adult` to the list of functions to be executed once the page is finished loading, as the last row of Figure 6.12 shows. Since you can call `$()` multiple times to run multiple setup functions, you can keep each file’s setup function together with that file’s functionality, as we’ve done here. To run this example place all the code from Figure 6.12 in `app/assets/javascripts/movie_list_filter.js`.

This was a dense example, but it illustrates the basic jQuery functionality you’ll need for many UI enhancements. The figures and tables in this section generalize the techniques introduced in the example, so it’s worth spending some time perusing them. In particular, Figure 6.12 summarizes the four different ways to use jQuery’s `$`, all of which we’ve now seen.

Finally, most of jQuery’s events are based on the built-in events recognized by browsers, but you can also define your own custom events and use `trigger` to trigger them, and many jQuery-based libraries do just that. For example, Bootstrap’s plugin for showing a **modal**

window defines a custom event `show` that is generated when a modal window is displayed and another custom event `shown` that is generated when that window is dismissed. Your own code can listen for these events in order to take actions before or after the modal is displayed. In your own code, you might enclose menus for month and day in a single outer element such as a `div`, and then define a custom `update` event on the `div` that checks that the month and day are compatible. You could then isolate the checking code in a separate event handler for `update`, and use `trigger` to call it from within the `change` handlers for the individual month and day menus. This is one way that custom handlers help DRY out your JavaScript code.

■ **Elaboration: JavaScript functions are closures**

Throughout these examples, it's easy to forget while writing an event handler that the time and place the handler code is defined is very different from the time and place (runtime scope) in which it will run when the event actually occurs. Fortunately, because JavaScript functions are true closures, wherever and whenever your event handler runs, it will have access to all of the same variables that were visible when the handler code was defined. This property—that a function can always see all the variables that were in scope at the time of its definition—will prove even more important for AJAX callbacks, which we discuss in the next section.



Summary of jQuery's DOM and event handlers:

- You can set or override how various HTML elements react to user input by binding JavaScript handlers or *callbacks* to specific events on specific elements. jQuery allows you to bind both “physical” user events such as mouse clicks and “logical” pseudo-events such as form submission. Figure 6.9 summarizes a subset of jQuery events.
- Inside an event handler, jQuery causes `this` to be bound to the *browser’s* DOM representation of the element that handled the event. We usually “wrap” the element to get `$(this)`, a “jQuery-wrapped” element that supports enhanced jQuery operations, such as `$(this).is(':checked')`.
- One of jQuery’s advanced features is the ability to apply transformations such as `show()` and `hide()` to a collection of elements (for example, a group of elements named by a single CSS selector) as well as a single element.
- For both DRYness and graceful degradation, the binding of event handlers to elements should occur in a setup function that is called when the document is loaded and ready; that way, legacy non-JavaScript browsers will not run the function at all. Passing a function to `$()` adds it to the list of setup functions that will be run once the document is finished loading.

Self-Check 6.6.1. Explain why calling `$(selector)` is equivalent to calling `$(window.document).find(selector)`.

◊ `document` is a property of the browser’s built-in global object (`window`) that refers to the browser’s representation of the root of the DOM. Wrapping the document element using `$`

gives it access to jQuery functions such as `find`, which locates all elements matching the selector that are in the subtree of its target; in this case, the target is the DOM root, so it will find any matching elements in the entire document. ■

Self-Check 6.6.2. In Self-Check 6.6.1, why did we need to write `$(document).find` rather than `document.find`?

◊ `document`, also known as `window.document`, is the browser's native representation of the `document` object. Since `find` is a jQuery function, we need to "wrap" `document` to give it special jQuery powers. ■

Self-Check 6.6.3. What would happen if we omitted the last line of Figure 6.10, which arranges to call the `setup` function?

◊ The browser would behave like a legacy browser without JavaScript. The checkbox wouldn't be drawn (since that happens in the `setup` function) and even if it were, nothing would happen when it was clicked, since the `setup` function binds our JavaScript handler for the checkbox's `change` event. ■

6.7 AJAX: Asynchronous JavaScript And XML

In 1998, Microsoft added a new function to the JavaScript global object defined by Internet Explorer 5. `XmlHttpRequest` (usually shortened to XHR) allowed JavaScript code to initiate HTTP requests to a server *without* loading a new page and use the server's response to modify the DOM of the current page. This new function, key to AJAX apps, allowed creating a rich interactive UI that more closely resembled a desktop application, as Google Maps powerfully demonstrated. Happily, you already know all the ingredients needed for "AJAX on Rails" programming:

1. Create a controller action or modify an existing one (Section 4.4) to handle the AJAX requests made by your JavaScript code. Rather than rendering an entire view, the action will render a partial (Section 5.1) to generate a chunk of HTML for insertion into the page.
2. Construct your RESTful URI in JavaScript and use XHR to send the HTTP request to a server. As you may have guessed, jQuery has helpful shortcuts for many common cases, so we will use jQuery's higher-level and more powerful functions rather than calling XHR directly.
3. Because JavaScript is by definition ***single-threaded***—it can only work on one task at a time until that task completes—the browser's UI would be "frozen" while JavaScript awaited a response from the server. Therefore XHR instead returns immediately and lets you provide an event handler callback (as you did for browser-only programming in Section 6.6) that will be triggered when the server responds or an error occurs.
4. When the response arrives at the browser, your callback is passed the response content. It can use jQuery's `replaceWith()` to replace an existing element entirely, `text()` or `html()` to update an element's content in place, or an animation such as `hide()` to hide or show elements, as Figure 6.8 showed. Because JavaScript functions are closures (like Ruby blocks), the callback has access to all the variables visible at the time the XHR call was made, even though it executes at a later time and in a different environment.

<https://gist.github.com/52a502b7240baf1945806d94624e41f4>

```

1 <p> <%= movie.description %> </p>
2
3 <%= link_to 'Edit Movie', edit_movie_path(movie), :class => 'btn btn-primary'
   %>
4 <%= link_to 'Close', '', :id => 'closeLink', :class => 'btn btn-secondary' %>
```

<https://gist.github.com/18c678c940549b1fa7468f0d2ff53401>

```

1 class MoviesController < ApplicationController
2   def show
3     id = params[:id] # retrieve movie ID from URI route
4     @movie = Movie.find(id) # look up movie by unique ID
5     render(:partial => 'movie', :object => @movie) if request.xhr?
6     # will render app/views/movies/show.<extension> by default
7   end
8 end
```

Figure 6.13: (a) Top: a simple partial that will be rendered and returned to the AJAX request. We give the “Close” link a unique element ID so we can conveniently bind a handler to it that will hide the popup. (b) Bottom: The controller action that renders the partial, obtained by a simple change to Figure 4.5: if the request is an AJAX request, line 5 performs a render and immediate return. The :object option makes @movie available to the partial as a local variable whose name matches the partial’s name, in this case `movie`. If `xhr?` is not true, the controller method will perform the default rendering action, which is to render the `show` view as usual.

Let’s illustrate how each step works for an AJAX feature in which clicking on a movie title shows the movie details in a floating window, rather than loading a separate page. Step 1 requires us to identify or create a new controller action that will handle the request. We will just use our existing `MoviesController#show` action, so we don’t need to define a new route. This design decision is defensible since the AJAX version of the action performs the same function as the original version, namely the RESTful “show” action. We will modify the `show` action so that if it’s responding to an AJAX request, it will render the simple partial in Figure 6.13(a) rather than an entire view. You could also define separate controller actions exclusively for AJAX, but that might be non-DRY if they duplicate the work of existing actions.

How does our controller action know whether `show` was called from JavaScript code or by a regular user-initiated HTTP request? Fortunately, every major JavaScript library and most browsers set an HTTP header `X-Requested-With: XMLHttpRequest` on all AJAX HTTP requests. The Rails helper method `xhr?`, defined on the controller instance’s `request` object representing the incoming HTTP request, checks for the presence of this header. Figure 6.13(b) shows the controller action that will render the partial.

Moving on to step 2, how should our JavaScript code construct and fire off the XHR request? We want the floating window to appear when we click on the link that has the movie name. As Section 6.6 explained, we can “hijack” the built-in behavior of an element by attaching an explicit JavaScript `click` handler to it. Of course, for graceful degradation, we should only hijack the link behavior if JavaScript is available. So following the same strategy as the example in Section 6.6, our `setup` function (lines 2–8 of Figure 6.14) binds the handler and creates a hidden `div` to display the floating window. Legacy browsers won’t run that function and will just get the default behavior of clicking on the link.

The actual click handler `getMovieInfo` must fire off the XHR request and provide a callback function that will be called with the returned data. For this we use jQuery’s `ajax` function, which takes an object whose properties specify the characteristics of the AJAX request, as lines 10–15 of Figure 6.14 show. Our example shows a subset of the properties you can specify in this object; one important property we don’t show is `data`, which can be

Hijax is sometimes humorously used to describe this technique.

Of course, `$.ajax` is just an alias for `jQuery.ajax`.

<https://gist.github.com/8101873022c2b3542451850446b85ba2>

```

1  var MoviePopup = {
2    setup: function() {
3      // add hidden 'div' to end of page to display popup:
4      let popupDiv = $('<div id="movieInfo"></div>');
5      popupDiv.hide().appendTo($('body'));
6      $(document).on('click', '#movies a', MoviePopup.getMovieInfo);
7    }
8    ,getMovieInfo: function() {
9      $.ajax({type: 'GET',
10        url: $(this).attr('href'),
11        timeout: 5000,
12        success: MoviePopup.showMovieInfo,
13        error: function(xhrObj, textStatus, exception) { alert('Error!'); }
14        // 'success' and 'error' functions will be passed 3 args
15      });
16      return(false);
17    }
18    ,showMovieInfo: function(data, requestStatus, xhrObject) {
19      // center a floater 1/2 as wide and 1/4 as tall as screen
20      let oneFourth = Math.ceil($(window).width() / 4);
21      $('#movieInfo').
22        css({'left': oneFourth, 'width': 2*oneFourth, 'top': 250}).
23        html(data).
24        show();
25      // make the Close link in the hidden element work
26      $('#closeLink').click(MoviePopup.hideMovieInfo);
27      return(false); // prevent default link action
28    }
29    ,hideMovieInfo: function() {
30      $('#movieInfo').hide();
31      return(false);
32    }
33  };
34  $(MoviePopup.setup);

```

Figure 6.14: The `ajax` function constructs and sends an XHR request with the given characteristics. `type` specifies the HTTP verb to use, `url` is the URL or URI for the request, `timeout` is the number of milliseconds to wait for a response before declaring failure, `success` specifies a function to call with the returned data, and `error` specifies a function to call if a timeout or other error occurs. Many more options to the `ajax` function are available, in particular for more robust error handling.

<https://gist.github.com/82017db9284e2d3f5daa4b3ff848e953>

```

1 #movieInfo {
2   padding: 2ex;
3   position: absolute;
4   border: 2px double grey;
5   background: wheat;
6 }
```

Figure 6.15: Adding this code to `app/assets/stylesheets/application.css` specifies that the “floating” window should be positioned at absolute coordinates rather than relative to its enclosing element, but as the text explains, we don’t know until runtime what those coordinates should be, so we use jQuery to dynamically modify `#movieInfo`’s CSS style properties when we are ready to display the floating window.

either a string of arguments to append to the URI (as in Figure 3.2) or a JavaScript object, in which case the object’s properties and their values will be serialized into a string that can be appended to the URI. As always, such arguments would then appear in the `params []` hash available to our Rails controller actions.

Looking at the rest of the code in Figure 6.14, getting the URI that is the target of the XHR request is easy: since the link we’re hijacking already links to the RESTful URI for showing movie details, we can query its `href` attribute, as line 10 shows. Lines 12–13 remind us that function-valued properties can specify either a named function, as `success` does, or an anonymous function, as `error` does. To keep the example simple, our error behavior is rudimentary: no matter what kind of error happens, including a timeout of 5000 ms (5 seconds), we just display an alert box. In case of success, we specify `showMovieInfo` as the callback.

Some interesting CSS trickery happens in lines 20 and 23 of Figure 6.14. Since our goal is to “float” the popup window, we can use CSS to specify its positioning as `absolute` by adding the markup in Figure 6.15. But without knowing the size of the browser window, we don’t know how large the floating window should be or where to place it. `showMovieInfo` computes the dimensions and coordinates of a floating `div` half as wide and one-fourth as tall as the browser window itself (line 20). It replaces the HTML contents of the `div` with the data returned from the server (line 22), centers the element horizontally over the main window and 250 pixels from the top edge (line 23), and finally shows the `div`, which up until now has been hidden (line 24).

There’s one last thing to do: the floated `div` has a “Close” link that should make it disappear, so line 26 binds a very simple `click` handler to it. Finally, `showMovieInfo` returns `false` (line 27). Why? Because the handler was called as the result of clicking on a link (`<a>`) element, we need to return `false` to suppress the default behavior associated with that action, namely following the link. (For the same reason, the “Close” link’s `click` handler returns `false` in line 31.)

With so many different functions to call for even a simple example, it can be hard to trace the flow of control when debugging. While you can always use `console.log(string)` to write messages to your browser’s JavaScript console window, it’s easy to forget to remove these in production, and as Chapter 8 describes, such “printf debugging” can be slow, inefficient and frustrating. In Section 6.8 we’ll introduce a better way by creating tests with Jasmine.

Lastly, there is one caveat we need to mention which could arise when you use JavaScript to dynamically create new elements at runtime, although it didn’t arise in this particular example. We know that `$('.myClass').on('click', func)` will bind `func` as the click handler for all current elements that match CSS class `myClass`. But if you then use JavaScript

to create new elements matching `myClass` *after* the initial page load and initial call to `on`, those elements won't have the handler bound to them, because `on` can only bind handlers to already-existing elements.

A common solution to this problem is to take advantage of a jQuery mechanism that allows an ancestor element to delegate event handling to a descendant, by using `on`'s polymorphism: `$(‘body’).on(‘click’, ‘.myClass’, func)` binds the HTML body element (which always exists) to the `click` event, but *delegates* the event to any descendant matching the selector `.myClass`. Since the delegation check is done each time an event is processed, new elements matching `.myClass` will “automagically” have `func` bound as their click handler when created.

Summary of AJAX:

- To create an AJAX interaction, figure out what elements will acquire new behaviors, what new elements may need to be constructed to support the interaction or display responses, and so on.
- An AJAX interaction will usually involve three pieces of code: the handler that initiates the request, the callback that receives the response, and the code in the `document.ready` function (setup function) to bind the handler. It's more readable to do each in a separate named function rather than providing anonymous functions.
- Just as we did in the example of Section 6.6, for graceful degradation, any page elements used *only* in AJAX interactions should be constructed in your setup function(s), rather than being included on the HTML page itself.
- Both interactive debuggers such as Firebug or the JavaScript consoles in Google Chrome and Safari and “printf debugging” using `console.log()` can help you find JavaScript problems, but a better way is through testing, which we show how to do in Section 6.8.

■ Elaboration: Event-driven programming

The programming model in which operations specify a completion callback rather than waiting for completion to occur is called **event-driven programming**. As you might conclude from the number of handlers and callbacks in this simple example, event-driven programs are considered harder to write and debug than **task-parallel** programs such as Rails apps, in which separate machinery in the app server effectively creates multiple copies of our app to handle multiple simultaneous users. Of course, behind the scenes, the operating system is switching among those tasks just as programmers do manually in JavaScript: when one user's “copy” of the app is blocked waiting for a response from the database, for example, another user's copy is allowed to make progress, and the first copy gets “called back” when the database response arrives. In this sense, event-driven and task-parallel programming are duals, and emerging standards such as WebWorkers²⁰ enable task parallelism in JavaScript by allowing different copies of a JavaScript program to run simultaneously on different operating system threads. However, JavaScript itself lacks concurrency abstractions such as Java's `synchronized` and inter-thread communication, so concurrency must be managed explicitly by the application.

What	RSpec/Ruby	Jasmine/JavaScript
Libraries	<code>rspec</code> , <code>rspec-rails</code> gems	<code>jasmine</code> gem, <code>jasmine-jquery</code> add-on
Setup	<code>rails generate rspec:install</code>	<code>rails generate jasmine:install</code>
Test files	<code>spec/models/</code> , <code>spec/controllers/</code> , <code>spec/helpers</code>	<code>spec/javascripts/</code>
Naming conventions	<code>spec/models/movie_spec.rb</code> contains tests for <code>app/models/movie.rb</code>	<code>spec/javascripts/movie_popup_spec.js</code> contains tests for <code>app/assets/javascripts/movie_popup.js</code> ; <code>spec/javascripts/moviePopupSpec.js</code> contains tests for <code>app/assets/javascripts/moviePopup.js</code>
Configuration file	<code>.rspec</code>	<code>spec/javascripts/support/jasmine.yml</code>
Run all tests	<code>rake spec</code>	<code>rake jasmine</code> , then visit <code>http://localhost:8888</code> ; or <code>rake jasmine:ci</code> to run once using Selenium/Webdriver and capture the output; or use <code>jasmine-headless-webkit</code> ²¹ to run from command line with no browser

Figure 6.16: Comparison of setting up and using Jasmine and RSpec. All paths are relative to the app root and all commands should be run from the app root. As you can see, the main difference is the use of `lower_snake_case` for filenames and method names in Ruby, versus `lowerCamelCase` in JavaScript.

Self-Check 6.7.1. In line 13 of Figure 6.14, why did we write `MoviePopup.showMovieInfo` instead of `MoviePopup.showMovieInfo()`?

- ◊ The former is the actual function, which is what `ajax` expects as its `success` property, whereas the latter is a *call* to the function. ■

Self-Check 6.7.2. In line 33 of Figure 6.14, why did we write `$(MoviePopup.setup)` rather than `$('MoviePopup.setup')` or `$(MoviePopup.setup())`?

- ◊ We need to pass the actual function to `$()`, not its name or the result of calling it. ■

Self-Check 6.7.3. Continuing Self-Check 6.7.2, if we had accidentally called `$('MoviePopup.setup')`, would the result be a syntax error or legal but unintended behavior?

- ◊ Recall that `$()` is overloaded, and when called with a string, it tries to interpret the string as HTML markup if it contains any angle brackets or a CSS selector otherwise. The latter applies in this case, so it would return an empty collection, since there are no elements whose tag is `MoviePopup` and whose CSS class is `setup`. ■

6.8 Testing JavaScript and AJAX

Even our simple AJAX example has many moving parts. In this section we show how to test it using Jasmine, an open-source JavaScript TDD framework developed by Pivotal Labs. Jasmine is designed to mimic RSpec and support the same TDD practices RSpec supports. The rest of this section assumes you've read Chapter 8 or are otherwise proficient with TDD and RSpec; as Figure 6.16 shows, we will reuse all those TDD concepts in Jasmine.

To start using Jasmine, add the `jasmine-rails` and `jasmine-jquery-rails` gems to the development and test groups in your Gemfile, and run `bundle` as usual, then run the



<https://gist.github.com/59575a6276a4c9db68ba3fc5a8e801da>

```
1 rails generate jasmine_rails:install
2 mkdir spec/javascripts/fixtures
3 git add spec/javascripts
```

Figure 6.17: Creating the Jasmine-related directories in your app. Line 1 creates a `spec/javascripts` directory where our tests will go, with subdirectories `support` and `helper` analogous to RSpec's setup. Line 2 adds a subdirectory for fixtures (Section 8.6). Line 3 adds these new JavaScript TDD files to your project.

commands in Figure 6.17 from your app's root directory. Create a simple example spec file `spec/javascripts/basic_check_spec.js` containing the following code:

<https://gist.github.com/63184132fe4e8f08b2b5d72f2e5c08af>

```
1 describe('Jasmine basic check', function() {
2   it('works', function() { expect(true).toBe(true); });
3 });
```

To run Jasmine tests, just start your app as usual with the `rails server` command, and once it's running, browse to the `specs` subdirectory of your app (so, for example, `http://localhost:3000/specs` if developing on your own computer) to run all the specs and see the results. From now on, when you change any code in `app/assets/javascripts` or tests in `spec/javascripts`, just reload the browser page to rerun all the tests.

Self-checking?

`rake spec:javascript` runs the Jasmine suite just once using the PhantomJS headless webkit and collects the output, making it suitable for use in continuous integration (Section 10.4).

Testing AJAX code must address two problems, and if you have read about TDD in Chapter 8, you're already familiar with the solutions to both. First, just as we did in Section 8.4, we must be able to "stub out the Internet" by intercepting AJAX calls, so that we can return "canned" AJAX responses and test our JavaScript code in isolation from the server. We will solve this problem using stubs. Second, our JavaScript code expects to find certain elements on the rendered page, but as we just saw, when running Jasmine tests the browser is viewing the Jasmine reporting page rather than our app. Happily, we can use fixtures to test JavaScript code that relies on the presence of certain DOM elements on the page, just as we used them in Section 8.6 to test Rails app code that relies on the presence of certain items in the database.

Figure 6.18 gives an overview of Jasmine for RSpec users. We will walk through five happy-path Jasmine specs for the popup-window functionality developed in Section 6.7. While these tests are hardly exhaustive even for the happy path, our goal is to illustrate Jasmine testing techniques generally and the use of Jasmine stubs and fixtures in AJAX testing specifically.

The basic structure of Jasmine test cases is immediately evident in Figure 6.20: like RSpec, Jasmine uses `it` to specify a single example and nestable `describe` blocks to group related sets of examples. Just as in RSpec, `describe` and `it` take a block of code as an argument, but whereas in Ruby code blocks are delimited by `do...end`, in JavaScript they are anonymous functions (functions without a name) of zero arguments. The punctuation sequence `});` is so prevalent because `describe` and `it` are JavaScript functions of two arguments, the second of which is a function of no arguments.

The `describe('setup')` examples check that the `MoviePopup.setup` function correctly creates the `#movieInfo` container but keeps it hidden from display. `toExist` and `toBeHidden` are expectation matchers provided by the Jasmine-jQuery add-on. Since Jasmine loads all your JavaScript files before running any examples, the call to `setup` (line 34 of Figure 6.14) occurs before our tests run; hence it's reasonable to test whether that function did its work.

The `describe('AJAX call to server')` examples are more interesting because

Structure of test cases

- `it("does something", function() {...})`
Specifies a single test (spec) by giving a descriptive name and a function that performs the test.
- `describe("behaviors", function(){...})`
Collects a related set of specs; the function body consists of calls to `it`, `beforeEach`, and `afterEach`. `describes` can be nested.
- `beforeEach` and `afterEach`
Setup/teardown functions that are run before each `it` block within the same `describe` block.
As with RSpec, if `describes` are nested, all `beforeEach` are run from the outside in, and all `afterEach` from the inside out.

Expectations

An expectation in a spec takes the form `expect(object).expectation` or `expect(object).not.expectation`

Commonly used expectations built into Jasmine:

- `toEqual(val)`, `toBeTruthy()`, `toBeFalsy()`
Test for equality using `==`, or that an expression evaluates to Boolean true or false.
- `toBeSelected()`, `toBeChecked()`, `toBeDisabled()`,
`toHaveValue(stringValue)`
Expectations on input elements in forms.
- `toBeVisible()`, `toBeHidden()`
Hidden is true if the element has zero width and height, if it is a form input with `type="hidden"`, or if the element or one of its ancestors has the CSS property `display: none`.
- `toExist()`, `toHaveClass(class)`, `toHaveId(id)`, `toHaveAttr(attrName,attrValue)`
Tests various attributes and characteristics of an element.
- `toHaveText(stringOrRegexp)`, `toContainText(string)`
Tests if the element's text exactly matches the given string or regexp, or contains the given substring.

Figure 6.18: A partial summary of a *small subset* of commonly used features in Jasmine and Jasmine-jQuery, following the structure of Figures 8.10 and 8.11 and extracted from the complete Jasmine documentation²⁴ and Jasmine jQuery add-on²⁵ documentation.

Stubs (Spies)

- `spyOn(obj, 'func')`
Creates and returns a spy (mock) of an existing function, which must be a function-valued property of `obj` named by `func`. The spy *replaces* the existing function.
- `calls` is a property of a spy that tracks calls that have been made to it, and the array `args[]` of the arguments of each call.

The following modifiers can be called on a spy to control its behavior:

- `and.returnValue(value)`
- `and.throwError(exception)`
- `and.callThrough()`
- `and.callFake(func)`

`func` must be a function of zero arguments, though it has access to the arguments with which the spy was called via `spy.calls.mostRecent().args[]`, and can call other functions using these arguments.

Fixtures and factories (requires `jasmine-jquery`)

- `sandbox({class: 'myClass', id: 'myId'})`
Creates an empty `div` with the given HTML attributes, if any; default is an empty `div` with no CSS class and an ID of `sandbox`. An alternative way to create the argument to `setFixtures` that avoids putting literal HTML strings into your test code.
- `loadFixtures("file.html")`
Load HTML content from in `spec/javascripts/fixtures/file.html` and put it inside a `div` with ID `jasmine-fixtures`, which is cleaned out between test cases.
- `setFixtures(HTMLcontent)`
Create a fixture directly instead of loading it from a file. `HTMLcontent` can be a literal string of HTML such as `<p class="foo">text</p>` or a jQuery-wrapped element such as `$('<p class="foo">text</p>')`.
- `getJSONFixture("file.json")`
Returns the JSON object in `spec/javascripts/fixtures/file.json`. Useful for storing mock data to simulate the result of an AJAX call without having to put literal JSON objects into your test code.

Figure 6.19: Continuation of Figure 6.18 describing stubs (*spies* in Jasmine) and fixtures.

<https://gist.github.com/b7c08e54899a87178d606f7c74c4fc77>

```

1 | describe('MoviePopup', function() {
2 |   describe('setup', function() {
3 |     it('adds popup Div to main page', function() {
4 |       expect($('#movieInfo')).toExist();
5 |     });
6 |     it('hides the popup Div', function() {
7 |       expect($('#movieInfo')).toBeHidden();
8 |     });
9 |   });
10 |  describe('clicking on movie link', function() {
11 |    beforeEach(function() { loadFixtures('movie_row.html'); });
12 |    it('calls correct URL', function() {
13 |      spyOn($, 'ajax');
14 |      $('#movies a').trigger('click');
15 |      expect($.ajax.calls.mostRecent().args[0]['url']).toEqual('/movies/1');
16 |    });
17 |    describe('when successful server call', function() {
18 |      beforeEach(function() {
19 |        let htmlResponse = readFixtures('movie_info.html');
20 |        spyOn($, 'ajax').and.callFake(function.ajaxArgs) {
21 |          ajaxArgs.success(htmlResponse, '200');
22 |        });
23 |        $('#movies a').trigger('click');
24 |      });
25 |      it('makes #movieInfo visible', function() {
26 |        expect($('#movieInfo')).toBeVisible();
27 |      });
28 |      it('places movie title in #movieInfo', function() {
29 |        expect($('#movieInfo').text()).toContain('Casablanca');
30 |      });
31 |    });
32 |  });
33 |});
```

Figure 6.20: Five happy-path Jasmine specs for the AJAX code developed in Section 6.7. Lines 2–9 check whether the `MoviePopup.setup` function correctly sets up the floating div that will be used to display movie info. Lines 10–32 check the behavior of the AJAX code without actually calling the RottenPotatoes server by stubbing around the AJAX call.

<https://gist.github.com/a08c97b3b1b4990c613ce25985572b40>

```

1 | <div id="movies">
2 |   <div class="row">
3 |     <div class="col-8"><a href="/movies/1">Casablanca</a></div>
4 |     <div class="col-2">PG</div>
5 |     <div class="col-2">1943-01-23</div>
6 |   </div>
7 | </div>
```

Figure 6.21: This HTML fixture mimics a row of the `#movies` table generated by the RottenPotatoes list-of-movies view (Figure 4.5). Note that we omit the table header from the fixture, since the spec doesn't require it to be present. This fixture would go in `spec/javascripts/fixtures/movie_row.html`. You can generate such fixtures by copy-and-pasting HTML code from “View Source” in the browser, or for source that was generated dynamically by JavaScript (such as the “Hide adult movies” checkbox), by inspecting `$('#movieInfo').html()` in the JavaScript console. Fallacies and Pitfalls describes a way to prevent such fixtures from getting out of sync if you change your app’s views.

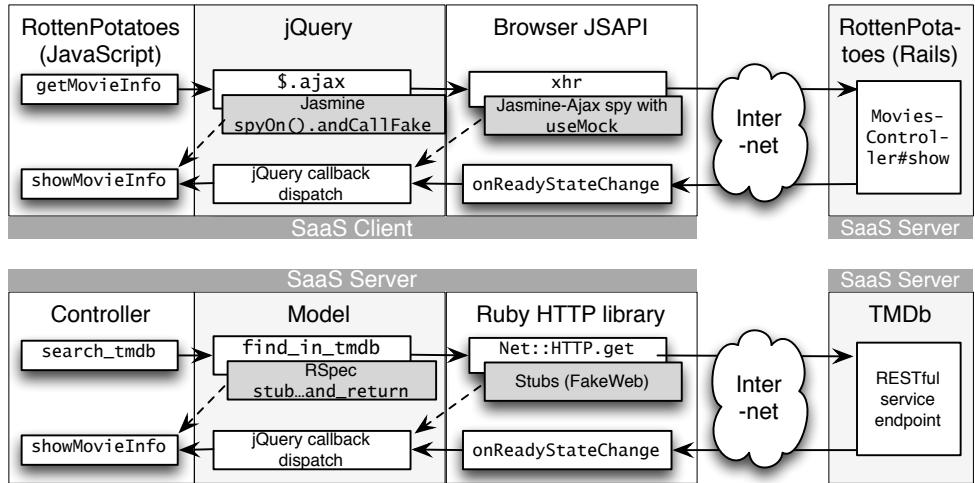


Figure 6.22: Top: Normally, our `getMovieInfo` function calls jQuery's `ajax`, which calls `xhr` in the browser's JSAPI, which sends the request to the server. The server's reply triggers callback logic in the browser's JSAPI, which calls an internal jQuery method that eventually calls our `showMovieInfo` callback. If we stub the `ajax` function, we can cause `showMovieInfo` to be called immediately; we can also stub "farther away" by stubbing `xhr` (using the Jasmine-Ajax plugin), causing the jQuery internal dispatcher to be called immediately. Bottom: Graphical representation of the discussion in Section 8.4.

they use stubs and fixtures to isolate our client-side AJAX code from the server with which it communicates. Figure 6.19 summarizes the stubs and fixtures available in Jasmine and Jasmine-jQuery. Like RSpec, Jasmine allows us to run test setup and teardown code using `beforeEach` and `afterEach`. In this set of examples, our setup code loads the HTML fixture shown in Figure 6.21, to mimic the environment the `getMovieInfo` handler would see if it was called after movie list was displayed. The fixtures functionality is provided by Jasmine-jQuery; each fixture is loaded inside of `div#jasmine-fixtures`, which is inside of `div#jasmine_content` on the main Jasmine page, and all the fixtures are cleared out after each spec to preserve test independence.

The first example (line 12 of Figure 6.20) checks that the AJAX call uses the correct movie URL derived from the table. To do this, it uses Jasmine's `spyOn` to stub out the `$.ajax` function. Like RSpec's `stub`, this call *replaces* any existing function of the same name, so when we manually `trigger` the click action on the (only) `a` element in the `#movies` table, if all is working well we should expect our spy function to have been called. Because in JavaScript it's common for functions to be the values of object properties, `spyOn` takes two arguments, an object (\$) and the name of the function-valued property of that object on which to spy ('`ajax`').

Line 15 looks complex, but it's straightforward. Each Jasmine spy remembers the arguments passed to it in each of its calls, e.g. `calls.mostRecent()`, and as you recall from the explanation in Section 6.7, a *real* call to the AJAX function takes a single object (lines 9–15 of Figure 6.14) whose `url` property is the URL to which the AJAX call should go. Line 15 of the spec is simply checking the value of this URL. In effect, it's testing whether `$(this).attr('href')` is the correct JavaScript code to extract the AJAX URL from the table.

Figure 6.22 shows the similarity between the challenges of stubbing the Internet for test-

<https://gist.github.com/39925bd69d7d6e12af94e4b9bac5f5b2>

```
1 <p>Casablanca is a classic and iconic film starring
2 Humphrey Bogart and Ingrid Bergman.</p>
3 <a href="" id="closeLink">Close</a>
```

Figure 6.23: This HTML fixture mimics the ajax response from the movies controller show action; it goes in `spec/javascripts/fixtures/movie_info.html`.

<https://gist.github.com/5bb00dbb9e829d245228982474b8dbca>

```
1 describe('element sanitizer', function() {
2   it('removes IMG tags from evil HTML', function() {
3     setFixtures(sandbox({class: 'myTestClass'}));
4     $('.myTestClass').text("Evil HTML! <img src='http://evil.com/xss'>");
5     $('.myTestClass').sanitize();
6     expect($('.myTestClass').text()).not.toContain('<img');
7   });
8 });
```

Figure 6.24: Jasmine-jQuery's `sandbox` method creates a new HTML `div` with the given attributes; its `id` defaults to `sandbox` if not given. Lines 4–5 use the `sandbox`-created element. The `sandbox` can be used to temporarily contain elements constructed in a factory-like way without “polluting” the test code with HTML markup.

ing AJAX and stubbing the Internet for testing code in a Service-Oriented Architecture (Section 8.4). As you can see, in both scenarios, the decision of where to stub depends on how much of the stack we want to exercise in our tests.

Line 19 reads in a fixture that will take the place of the ajax response from the movies controller show action, see Figure 6.23. In lines 20–22 we see the use fo the `callFake` function to not only intercept an AJAX call, but also to fake a successful response using the fixture. This and the triggering of the AJAX call (line 23) is repeated for each of the following two tests which check that both the `#movieInfo` popup is visible (line 26) and that it contains text from the movie description (line 29).

This concise introduction, along with the summary tables in this section, should get you started using BDD for your JavaScript code. The best sources of complete documentation for these tools are the Jasmine documentation²⁶ and the Jasmine jQuery add-on²⁷ documentation.

Summary of Jasmine BDD for JavaScript:

- Like RSpec, Jasmine specs are anonymous functions accompanied by a descriptive string. They are introduced by the Jasmine function `it`, can be grouped with (nested) `describe` blocks that have associated `beforeEach` and `afterEach` (test setup and teardown) calls.
- `spyOn` can be used to stub an existing method by replacing it with a spy. The spy's behavior can be controlled with functions like `and.callThrough`, `and.returnValue`, and so on, as Figure 6.19 shows.
- Jasmine-jQuery's HTML fixtures can provide both the “before” content for triggering an AJAX request and the “after” content for testing the results of a successful or failed AJAX request.

■ *Elaboration: Fixtures or factories?*

As Section 8.6 explains, in Rails apps it's often preferable to use a factory to create necessary test doubles “in place” rather than specifying fixtures. So why do we describe the use of fixtures rather than factories for AJAX testing? One reason is that the tradeoff is different in JavaScript. In the Rails app, fixtures are loaded into the database before tests are run, and various ActiveRecord methods such as `find` may behave differently when different fixtures are present; therefore fixtures may break test Independence. Factories are an appealing alternative in Rails because gems such as FactoryBot make it easy to instantiate test doubles “just in time” in each test that needs them. In Jasmine, to substitute an HTML “factory” for HTML fixtures, we would use `$('')` to create inline HTML elements, but many developers view this as undesirable because mixing HTML markup with JavaScript test code makes the latter hard to read. Jasmine-jQuery provides some simple support for using factories without excessively polluting your test code with HTML markup, as Figure 6.24 shows, but in general we see that fixtures for AJAX testing avoid some of the pitfalls of fixtures for Rails testing. They do, however, introduce a pitfall of their own—the possibility of getting “out of sync” with the app’s views. See Fallacies and Pitfalls for a discussion of this pitfall and its solution.

Self-Check 6.8.1. *Jasmine-jQuery also supports `toContain` and `toContainText` to check if a string of text or HTML occurs within an element. In line 7 of Figure 6.20, why would it be incorrect to substitute `.not().toContain('<div id="movieInfo"></div>')` for `toBeHidden()`?*

◊ A hidden element is not visible, but it still contains the text or HTML associated with the element. Hence `toContain`-style matchers can be used to test the *content* of an element but not its *visibility*. In addition, there are many ways for an element to be hidden—its CSS could include `display:none`, it could have zero width and height, or its ancestor could be hidden—and the `toBeHidden()` matcher checks all of these. ■

Self-Check 6.8.2. *Like RSpec, Jasmine supports `and.returnValue()` for returning a canned value from a stub. In Figure 6.20, why did we have to write `and.callFake` to pass `ajaxArgs` to a function as the result of stubbing `ajax`, rather than simply writing `and.returnValue(ajaxArgs)`?*

◊ Remember that AJAX calls are asynchronous. It's *not* the case that the `$.ajax` call returns data from the server: normally, it returns immediately, and sometime later, your callback is called with the data from the server. `and.callFake` simulates this behavior. ■

6.9 CHIPS: AJAX Enhancements to RottenPotatoes



CHIPS 6.9: AJAX Enhancements to RottenPotatoes

<https://github.com/saasbook/hw-javascript-ajax>

Write and deploy code, unit tests, and integration tests for new AJAX-enabled features in RottenPotatoes, while observing best practices of Unobtrusive JavaScript.

6.10 Single-Page Apps and JSON APIs

Google Maps was an early example of the emerging category called client-side single-page apps (SPAs). In a SPA, after the initial page load from the server, all interaction appears to the

user to occur without any page reloads. While we won’t develop a full SPA in this section, we will show the techniques necessary to do so.

Even SPAs usually have a significant server-side component. However, whereas client-side (in-browser) code *must* use JavaScript, server code can use a variety of languages and frameworks, as best serves the need. You might think that using JavaScript for both client and server code results in significant simplification; indeed, you can think of the Node and Express JavaScript libraries Express as analogous to Rack and Rails, respectively. But in fact, the three fundamental challenges of splitting an app between a client and a server benefit relatively little by using the same language:

1. The first is the need to identify and carefully manage important application state between the client and server. RESTful design advises us to do this by associating all important state with some type of resource, and defining and exposing a set of RESTful operations whose semantics are clear and well-circumscribed.
2. The second is the need to communicate data between the client and the server. Since the client and server are completely separate processes, in general such data must be **serialized**. For the vast majority of SPAs, JSON (JavaScript Object Notation) is used for this purpose.
3. The third is the need to manage dependencies among JavaScript libraries, analogously to what Bundler (Section 2.6) does for Ruby. This challenge only applies once the total amount of JavaScript code (whether on the client or on the server) exceeds a certain level of complexity.

This section addresses the second and third challenges, since the approach to the first challenge is not fundamentally different for SPAs than for the server-side apps you’ve been working with.

You’ve now seen at least three different ways to represent structured data: XML, YAML (Sections 4.2 and 8.6), and JSON. These three standards, and **many others**, address the problem of data **serialization** (also called *marshalling* or *deflating*)—translating a program’s internal data structures into a representation that can be “resurrected” later. Deserialization (unmarshalling, inflating) is often performed by a different program, possibly written in a different language or at the other end of a network connection, so the serialization format must be portable. Early SPAs used XML, but JSON has now eclipsed all other standards for serializing SaaS data between clients and servers.

How is JSON used in SPAs? Principally, the client requests some “raw” data that the server returns in JSON format, and the client uses that data to construct or modify DOM elements. To do this in Rails, we must solve four problems:

1. Since all requests from client to server use HTTP, how does the client indicate that a particular HTTP request “wants” a JSON response, rather than (for example) rendering an HTML view?
2. How do we get the server to generate JSON in response to such requests?
3. Since JSON objects are just properly-formatted strings, how does the client convert such strings into “real” JavaScript objects once the response is received, so that it can use the data to modify the DOM or take other action?

Ruby 1.9 added its alternate hash notation
`{foo: 'bar'}`,
equivalent to
`{:foo=>'bar'}`, to
mimic JSON.

<https://gist.github.com/2a6f7d0f83b3f7ddd42fcf9bbf4ef19e>

```

1 | Review.first.to_json
2 | # => "{\"created_at\":\"2012-10-01T20:44:42Z\", \"id\":1, \"movie_id\":1,
3 |   \"moviegoer_id\":2,\"potatoes\":3,\"updated_at\":\"2013-07-28T18:01:35Z\"}"

```

Figure 6.25: Rails’ built-in `to_json` can serialize simple ActiveRecord objects by calling itself recursively on each attribute of the model. As you can see, it doesn’t traverse associations—the review’s `movie_id` and `moviegoer_id` are serialized to integers, not to the `Movie` and `Moviegoer` objects to which the integer foreign keys refer. You can effect more sophisticated serialization by overriding `to_json` in your ActiveRecord models.

- When testing AJAX requests that expect JSON responses, how can we use “stub out the server” to test these behaviors in isolation, as we did in Section 6.8?

jQuery makes the first problem easy. To make an AJAX call that expects a JSON-encoded response, we just ensure that the argument object passed to `$.ajax` includes a `dataType` property whose value is the string `json`, as Figure 6.26 shows. (Recall that `$.ajax` ultimately calls `XmlHttpRequest` in the browser’s underlying JS API, which means that when the HTTP response is received, the browser *will not* automatically try to re-render the entire page.) Of course, the presence of this property doesn’t by itself cause the server to emit JSON—we address that next. But it does have one important effect: when the server returns data, the jQuery callback that receives the response will pass that data to `JSON.parse`, which converts a string of JSON into the corresponding JavaScript object(s), addressing the third problem above. This function is provided by the `JSON` object, which is part of the standard built-in collection of JavaScript objects.

How do we get the server to emit JSON rather than rendering a view? Rails controller actions can call `render :json=>object`, which sends a JSON representation of an object back to the client as the single response from the controller action. Like rendering a template, you are only allowed a single call to `render` per action, so all the response data for a given controller action must be packed into a single JSON object. This is not a limitation in practice, since in general most calls to a RESTful API expect a single JSON object as a result. `render :json` works by calling `to_json` on `object` to create the string to send back to the client. The default implementation of `to_json` can serialize simple ActiveRecord objects, as Figure 6.25 shows, but you can override it for your own models and other classes.

Finally, good unit-testing discipline requires us to be able to test the client-side JavaScript code without calling the server every time. Recall from Section 8.4 how stubbing the Internet to isolate tests from external services can be done either “near the client” or “far from the client.” In Section 6.8 we stubbed “near the client” by stubbing `$.ajax` and forcing it to immediately call the `success` function rather than allowing it to proceed with the external HTTP request. Another way to stub near the client for SPAs is Jasmine-jQuery’s fixture mechanism, which allows us to specify JSON fixtures as well as HTML fixtures, as Figure 6.27 shows. You can make a call to the actual server, capture the response as a JSON fixture, and use the fixture as a “canned” response in Jasmine tests, similar to how we stubbed `find_in_tmdb` in Section 8.4 to return a value immediately rather than allowing it to make a real HTTP request.

An alternative, which would more thoroughly exercise the code that handles the actual AJAX server responses, is to stub at the network level. Just as Webmock lets you provide “canned” XML or HTML responses based on the arguments of an XHR call, `jasmine-ajax`²⁸, a Jasmine extension from Pivotal Labs, lets you provide “canned” XML, HTML or JSON responses to AJAX XHR calls that are used instead of allowing the XHR call

<https://gist.github.com/499c2b3ad7ad5ba067e358462b7266e1>

```

1 let MoviePopupJson = {
2   // 'setup' function omitted for brevity
3   getMovieInfo: function() {
4     $.ajax({type: 'GET',
5        dataType: 'json',
6        url: $(this).attr('href'),
7        success: MoviePopupJson.showMovieInfo
8        // 'timeout' and 'error' functions omitted for brevity
9      });
10    return(false);
11  }
12 ,showMovieInfo: function(jsonData, requestStatus, xhrObject) {
13   // center a floater 1/2 as wide and 1/4 as tall as screen
14   let oneFourth = Math.ceil($(window).width() / 4);
15   $('#movieInfo').
16     css({'left': oneFourth, 'width': 2*oneFourth, 'top': 250}).
17     html($('

' + jsonData.description + '

'),
18       $('Close').
19         show());
20   // make the Close link in the hidden element work
21   $('#closeLink').click(MoviePopupJson.hideMovieInfo);
22   return(false); // prevent default link action
23 }
24 // hideMovieInfo omitted for brevity
25 };

```

Figure 6.26: This version of `MoviePopup` expects a JSON rather than HTML response (line 5), so the `success` function uses the returned JSON data structure to create new HTML elements inside the `popup` div (lines 17–19; observe that jQuery DOM-manipulation functions such as `append` can take multiple arguments of distinct pieces of HTML to create).

The functions omitted for brevity are the same as in Figure 6.14.

<https://gist.github.com/4a3fdcd704492c52eb5a3714bf284c5>

```

1 describe('MoviePopupJson', function() {
2   describe('successful AJAX call', function() {
3     beforeEach(function() {
4       loadFixtures('movie_row.html');
5       let jsonResponse = getJSONFixture('movie_info.json');
6       spyOn($, 'ajax').and.callFake(function.ajaxArgs) {
7         ajaxArgs.success(jsonResponse, '200');
8       });
9       $('#movies a').trigger('click');
10    });
11    // 'it' clauses are same as in movie_popup_spec.js
12  });
13 });

```

Figure 6.27: Jasmine-jQuery expects to find fixture files containing .json data in `spec/javascripts/fixtures/json`. After executing line 5, `jsonResponse` will contain the actual JavaScript object (not the raw JSON string!) that will get passed to the `success` handler.

to proceed. You can then spy on the handler functions `success`, `failure`, `timeout`, and so on passed to `$.ajax` to make sure the correct handler is called depending on the server's response.

Summary of Single-Page Apps:

- Whereas JavaScript-enhanced traditional SaaS apps will typically render complete chunks of HTML (for example, using partials) that the client will simply “plug into” the current HTML page, SPAs will usually receive structured data from one or more services and use that data to synthesize new content or modify existing content on the page.
- JSON’s simplicity and its natural fit with JavaScript are rapidly making it the preferred format for interchanging structured data in SPAs. Rails can serialize simple ActiveRecord models to JSON with `render :json=> object`, but you can override ActiveRecord’s `to_json` method to serialize arbitrarily complex data structures.
- Setting the `dataType` property to "json" in an `$.ajax` call tells jQuery to automatically deserialize the server’s response data into a JSON object.
- A spy that returns a JSON fixture can be used to simulate a server’s response in testing a SPA, allowing Jasmine tests to be isolated from the remote server(s) the SPA relies on.

■ Elaboration: Managing JavaScript dependencies

Early in the book we emphasized the importance of managing library versions in complex apps. For Ruby, the `gem` tool installs or uninstalls specific versions of Gems (libraries), and Bundler tracks and resolves dependencies among the set of gems needed by your app. The Node package manager `npm` is analogous to `gem`, and various tools are vying for the function of Bundler. Yarn, itself an `npm` package, manages `npm`-installed JavaScript libraries and tracks versions and dependencies just as Bundler does for Ruby gems. Webpack not only manages JavaScript libraries but also manages CSS stylesheets and does much of the work of the Rails asset pipeline (Section 6.2). Part of the motivation for Webpack is that ECMAScript version 6 (“ES6”) defines *modules* as a new JavaScript abstraction: a module is a namespace that can export specific symbols for use by other namespaces, with *live bindings* that ensure that when a module changes the value of such an exported symbol, the change is immediately visible in other modules that imported the symbol. Webpack attempts to construct a dependency graph of JavaScript modules that respects live bindings. Our view in early 2021 is that unless your client-side SPA code becomes comparable in complexity to your server code, Webpack (and the gem `webpacker`, which integrates it with Rails apps) may introduce complexity that does not pay back its own cost. But as these tools become more streamlined, expect to see JavaScript library management become a standard part of multi-language SaaS frameworks.

Self-Check 6.10.1. In Figure 6.27 showing the use of a JSON fixture, why do we also still need the HTML fixture to be loaded in line 4?

◊ Line 9 tries to trigger the click handler for an element matching `#movies a`, and if we don’t load the HTML fixture representing a row of the movies table, no such element will exist. (Indeed, the `MoviePopupJson.setup` function tries to bind a click handler on this element,

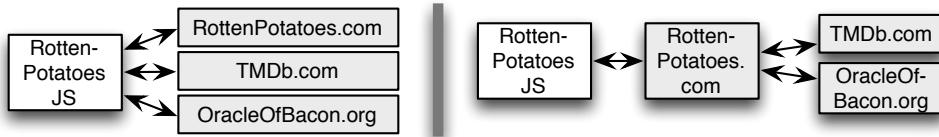


Figure 6.28: Architecture of in-browser SPAs that retrieve assets from multiple distinct services. Left: If the JavaScript code was served from RottenPotatoes.com, the default *same-origin policy* that browsers implement for JavaScript will forbid the code from making AJAX calls to servers in other domains. The *cross-origin resource sharing* (CORS) specification relaxes this restriction but is only supported by very recent browsers. Right: in the traditional SPA architecture, a single server serves the JavaScript code and interacts with other remote services. This arrangement respects the same-origin policy and also allows the main server to do additional work on behalf of the client if needed.

so that would also fail.) This is an example of using both an HTML fixture to simulate the user clicking on a page element and a JSON fixture to simulate a successful response from the server in response to that click. ■

■ ***Elaboration: Same-origin policy***

You can also arrange for your SPA to communicate with a RESTful server façade (Section 11.6), as Figure 6.28 shows. You might do this if your SPA relies on content from multiple sites: for security, JavaScript browser apps are bound by a ***same origin policy***, which says that a JavaScript app can only make AJAX requests to the same origin (scheme, host name, and port number, as described in Section 3.2) from which the app itself was served.

6.11 Fallacies and Pitfalls



Fallacy: AJAX will surely improve my app's responsiveness because more action happens right in the browser.

In a carefully-engineered app, judicious use of JavaScript and AJAX have the *potential* to improve responsiveness and usability. However, many factors also work against this goal. Your JavaScript code must be loaded from the server; Internet connection speeds today range from less than 1 Mbps (mobile phones in poorly-connected areas) to 1000 Mbps (high-speed wired networks). The JavaScript code must then be parsed and executed, and because JavaScript is single-threaded, JavaScript performance benefits from *faster* CPUs but not *more* CPUs. For both laptops and smartphones, this leads to order-of-magnitude differences in JavaScript performance between “premium” models (recent Apple iPhones, full-featured laptops) and “economy” models (US\$20 smartphones common in developing areas, older premium phones, entry-level Chromebooks) (Osmani 2019). That cute JavaScript animation may simply make your site frustratingly sluggish on less powerful or less well-connected devices. The true test of usability is not how the site behaves on your computer, but how it behaves on the devices used by the majority of your customers. The variation among those devices should make you cautious about the gratuitous overuse of JavaScript.



Pitfall: Disregarding the credibility of a source of information about JavaScript.

When JavaScript was embedded into browsers, it immediately became the one programming language that literally anyone with a browser could code in, and their creations could be deployed on the Web to boot. One unfortunate side effect of this rapid explosion in use is that there is a *lot* of bad JavaScript code out there, and trying to find good, sound advice online for “how to do X in JavaScript” can be like wading through a cesspit. Carefully vet the sources of such advice! The Mozilla Developer Network’s JavaScript documentation²⁹ is an excellent comprehensive reference for the language itself. *JavaScript The Right Way*³⁰ is primarily a curated set of links to descriptions of JavaScript best practices. And as always, don’t copy-paste code into your app if you don’t know where it’s been!



Pitfall: Creating a site that fails without JavaScript rather than being enhanced by it.

For reasons of accessibility by people with disabilities, security, and cross-browser compatibility, a well-designed site should work *better* if JavaScript is available, but *acceptably* otherwise. For example, GitHub’s pages for browsing code repos work well without JavaScript but work more smoothly and quickly with JavaScript. Try the site both ways for a great example of progressive enhancement. Tests also run faster without JavaScript: having a site for which JavaScript is optional means you can do the majority of your integration testing in the faster “headless browser” mode of Cucumber and Capybara.



Pitfall: Silent JavaScript failures in production code.

 When an unexpected exception occurs in your Rails code, you know it right away, as we’ve already seen: your app displays an ugly error page, or if you’ve been careful, a service like Hoptoad immediately contacts you to report the error, as we describe in Chapter 12). But JavaScript problems manifest as silent failures—the user clicks a control or loads a page, and nothing happens. These problems are especially pernicious because if they occur while an AJAX request is in progress, the `success` callback will never get called. So be warned: jQuery provides shortcuts for common uses of `$.ajax()` such as `$.get(url,data,callback)`, `$.post(url,data,callback)`, `$.load(url_and_selector)`, and `$.getJSON(url,data,callback)`, but all of these fail silently if anything goes wrong, whereas `$.ajax()` allows you to specify additional callbacks to be called in case of errors.



Pitfall: Silent JavaScript failures in tests.

The “silent failure” pitfall also arises when using Jasmine: if there are syntax errors in any of your JavaScript files or specs, when you reload the browser page that runs your Jasmine specs, you may see a blank page with no hint as to where the errors are. We suggest using Doug Crockford’s JSLint³¹ tool, which not only finds syntax errors but also points out bad habits and the use of JavaScript mechanisms that Crockford and others consider misfeatures.

Similarly, you may accidentally load HTML fixtures that result in illegal HTML. For example, you might accidentally create a fixture containing an element whose ID duplicates an existing element, or a fixture containing improperly-nested elements or HTML syntax errors. Since fixtures are loaded into an actual page when tests are run, the results of an ill-formed page may be unpredictable or result in silent failures.



Pitfall: Providing only expensive server operations and relying on JavaScript to do the rest.

If JavaScript is so powerful, why not write substantially all of the app logic in it, using the server as just a thin API to a database? For one thing, as we'll see in Chapter 12, successful scaling requires *reducing* the load on the database, and unless the APIs exposed to your JavaScript client code are carefully thought out, there's a risk of making needlessly complex database queries so that client-side JavaScript code can pick out the data it needs for each view. Second, whereas you have nearly complete control of performance (and therefore of the user experience) on the server side, you have nearly none on the client side. Because of wide variation in browser types, Internet connection speeds, and other factors beyond your control, JavaScript performance on each user's browser is largely out of your hands, making it difficult to provide consistent performance for the user experience.



Pitfall: Incorrect use of `this` in JavaScript functions.

The value of `this` in the body of a JavaScript function is the source of much grief and confusion for programmers new to the language. In particular, after seeing a couple of examples, new programmers don't realize that the value of `this` for a particular function is not dependent on how that function is written, but on how it is called, so different calls to the same function can result in different bindings for `this`. A complete discussion of why `this` works as it does is beyond the scope of this introduction, but the To Learn More section offers some pointers for those interested in delving deeper, which will take you into the realm of how JavaScript is influenced by its ancestors Scheme and Self.

Until you understand the issue more deeply, you can make your own code safe by following the common cases we outlined, which Figure 6.11 summarizes.



Pitfall: JavaScript—the bad parts.

The `++` operator was invented by [Ken] Thompson for pointer arithmetic. We now know that pointer arithmetic is bad, and we don't do it anymore; it's been implicated in buffer-overrun attacks and other evil stuff. The last popular language to include the `++` operator is C++, a language so bad it was named after this operator.

—Douglas Crockford, *Programming and Your Brain*, keynote at USENIX WebApps'12 conference

The entrepreneurial boom in which JavaScript was born was a time of ridiculous schedule pressures: LiveScript was designed, implemented, and released in a product in 10 days. As a result, the language has some widely-regarded misfeatures and pitfalls that some have compared to “gotchas” in the C language, so we urge you to use Doug Crockford's JSLint³² tool to warn you of both potential pitfalls and opportunities to beautify your JavaScript code.

Some specific pitfalls to avoid include the following:

1. The interpreter helpfully tries to insert semicolons it believes you forgot, but sometimes its guesses are wrong and result in drastic and unexpected changes in code behavior, such as the following example:



<https://gist.github.com/4491c97822d4b7bec7c886905b4e64d5>

```

1 // good: returns new object
2 return {
3   ok: true;
4 };
5 // bad: returns undefined, because JavaScript
6 // inserts "missing semicolon" after return
7 return
8 {
9   ok: true;
10 }

```

One good workaround is to adopt a consistent coding style designed to make “punctuation errors” quickly visible, such as the coding style recommended for Node.js package developers³³.

- Despite a syntax that suggests block scope—for example, the body of a for-loop inside a function gets its own set of curly braces inside which additional `var` declarations can appear—all variables declared with `var` in a function are visible *everywhere* throughout that function, including to any nested functions. Hence, in a common construction such as `for (var m in movieList)`, the scope of `m` is the entire function in which the for-loop appears, not just the body of the for-loop itself. The same is true for variables declared with `var` inside the loop body. This behavior, called *function scope*, was invented in Algol 60. Keeping functions short (remember SOFA from Section 9.5?) helps avoid the pitfall of block vs. function scope.
- An `Array` is really just a object whose keys are nonnegative integers. In some JavaScript implementations, retrieving an item from a linear array is marginally faster than retrieving an item from a hash, but not enough to matter in most cases. The pitfall is that if you try to index an array with a number that is negative or not an integer, a string-valued key will be created. That is, `a[2.1]` becomes `a["2.1"]`.
- The comparison operators `==` and `!=` perform type conversions automatically, so `'5'==5.0` is true. The operators `===` and `!==` perform comparisons without doing any conversions. This is potentially confusing because Ruby also has a `==` (“three-equal”) operator that does something quite different.
- Equality for arrays and hashes is based on identity and not value, so `[1,2,3]==[1,2,3]` is false. Unlike Ruby, in which the `Array` class can define its own `==` operator, in JavaScript you must work around these built-in behaviors, because `==` is part of the language.
- Strings are immutable, so methods like `toUpperCase()` always return a new object. Hence write `s=s.toUpperCase()` if you want to replace the value of an existing variable.
- If you call a function with more arguments than its definition specifies, the extra arguments are ignored; if you call it with fewer, the unassigned arguments are `undefined`. In either case, the array `arguments[]` (within the function’s scope) gives access to all arguments that were actually passed.
- String literals behave differently from strings created with `new String` if you try to create new properties on them, as the code excerpt below shows. The reason

is that JavaScript creates a temporary “wrapper object” around `fake` to respond to `fake.newprop=1`, performs the assignment, then immediately destroys the wrapper object, leaving the “real” `fake` without any `newprop` property. You can set extra properties on strings if you create them explicitly with `new`. But better yet, don’t set properties on built-in types: define your own prototype object and use composition rather than inheritance (Chapter 11) to make a string one of its properties, then set the other properties as you see fit. (This restriction applies equally to numbers and Booleans for the same reasons, but it doesn’t apply to arrays because, as we mentioned earlier, they are just a special case of hashes.)

<https://gist.github.com/0458f7f6c0921e7002342ce5f0d177d9>

```

1 | real = new String("foo");
2 | fake = "foo";
3 | real.newprop = 1;
4 | real.newprop      // => 1
5 | fake.newprop = 1; // BAD: silently fails since 'fake' isn't true object
6 | fake.newprop      // => undefined

```

6.12 Concluding Remarks: JavaScript Past, Present and Future

JavaScript’s privileged position as the client-side language of the Web has focused a lot of energy on it. Just-in-time compilation (JIT) techniques and other advanced language engineering features are being brought to bear on the language, closing the performance gap with other interpreted and even some compiled languages. Over half a dozen **JavaScript engine** implementations and one compiler (Google’s Closure) are available as of this writing, most of them open source, and vendors such as Microsoft, Apple, Google, and others compete on the performance of their browsers’ JavaScript interpreters. As early as 2011, JavaScript was fast enough to use to rewrite large parts of the Palm webOS operating system. We can expect this trend to continue, because JavaScript is one of the first languages to receive attention when new hardware becomes available that could be useful for user-facing apps.

We saw over and over again in studying Ruby and Rails that productivity often goes hand in hand with conciseness. JavaScript’s syntax is sometimes awkward, in part because JavaScript was always functional at heart (recall that its creator originally wanted to use Scheme as the browser scripting language) and in part because its large community of developers accustomed to class-oriented languages sometimes had difficulty embracing JavaScript’s alternative model of prototype-based inheritance.

ECMAScript version 6 (ES6) attempts to address this by providing new keywords such as `class` that look more familiar to such developers, but it’s important to remember that no new mechanisms or abilities were added to the language in this case. The new keywords are syntactic sugar that make prototype-based inheritance look and behave more like traditional classes. Indeed, it is possible to create a **Source-to-source compiler**, sometimes called a *transpiler*, that consumes ES6 and emits pure JavaScript, as some early browsers’ implementations of ES6 did.

JavaScript’s single-threaded execution model, which some feel hampers productivity because it requires event-driven programming, seems unlikely to change anytime soon. Some bemoan the adoption of JavaScript-based server-side frameworks such as Node, a JavaScript library that provides event-driven versions of the same POSIX (Unix-like) operating system



facilities used by task-parallel code. Rails core committer Yehuda Katz summarized the opinions of many experienced programmers: when things happen in a deterministic order, such as server-side code handling a controller action in a SaaS app, a sequential and blocking model is easier to program; when things happen in an unpredictable order, such as reacting to user-initiated user interface events, the asynchronous model makes more sense. Your authors firmly believe that the future of software is “cloud+client” apps, and our view is that it’s more important to choose the right language or framework for each job than to obsess about whether a single language or framework will become dominant for both the client and cloud parts of the app.

We covered only a small part of the language-independent DOM representation using its JavaScript API. The DOM representation itself has a rich set of data structures and traversal methods, and APIs are available for all major languages, such as the `dom4j`³⁴ library for Java and the `Nokogiri`³⁵ gem for Ruby.

Here are additional useful resources for mastering JavaScript and jQuery:

- A great presentation by Google JavaScript guru Miško Hevery: *How JavaScript works: introduction to JavaScript and Browser DOM*³⁶
- Yehuda Katz³⁷ is an active core committer to both Rails and jQuery, among other high-profile projects. His programmer-oriented blog posts discuss tips and techniques ranging from the practical to the esoteric for both Ruby and JavaScript. In particular, he has a nice post on the subtle difference between Ruby blocks and JavaScript anonymous functions³⁸ and another on why `this` works the way it does in JavaScript functions³⁹.
- jQuery is an extremely powerful library whose potential we barely tapped. Combined with Bootstrap, it may provide the facilities needed to enhance server-centric apps whose client-side part is not complex enough to justify the use of a full client-side framework such as React. *jQuery: Novice to Ninja* (Castledine and Sharkie 2012) is an excellent reference with many examples that go far beyond our introduction.
- *JavaScript: The Good Parts* (Crockford 2008), by the creator of the JSLint⁴⁰ tool, is a highly opinionated, intellectually rigorous exposition of JavaScript, focusing uncompromisingly on the disciplined use of its good features while candidly exposing the pitfalls of its design flaws. This book is “must” reading if you plan to write entire JavaScript apps comparable to Google Docs.
- The ProgrammableWeb⁴¹ site lists hundreds of service APIs, both RESTful and non-RESTful and serving both XML and JSON data, that you may find useful for SPAs and mashups. Some are completely open and require no authentication; others require a developer key which may be free or non-free.

E. Castledine and C. Sharkie. *jQuery: Novice to Ninja, 2nd Edition - New Kicks and Tricks*. SitePoint Books, 2012.

D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 2008.

A. Osmani. The Cost of JavaScript in 2019. In *PerfMatters Conference 2019*, June 2019. URL <https://v8.dev/blog/cost-of-javascript-2019>.

P. Seibel. *Coders at Work: Reflections on the Craft of Programming*. Apress, 2009. ISBN 1430219483.

Notes

¹<http://github.com/jasmine/jasmine>
²<https://vuejs.org/v2/guide/comparison.html>
³<http://jquery.org>
⁴<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
⁵<http://developers.google.com/closure>
⁶<http://yui.github.io/yuicompressor>
⁷<http://jsonlint.com>
⁸http://guides.rubyonrails.org/asset_pipeline.html
⁹<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>
¹⁰<http://www.w3.org/DOM>
¹¹<http://api.jquery.com>
¹²<http://api.jquery.com>
¹³<https://www.w3.org/TR/html52/>
¹⁴<https://html.spec.whatwg.org/multipage/input.html#the-input-element>
¹⁵<https://www.w3.org/TR/html-aria/>
¹⁶https://www.w3.org/TR/wai-aria-1.1/#roles_categorization
¹⁷<https://www.w3.org/TR/WCAG21/>
¹⁸<https://www.deque.com/axe/axe-for-web/>
¹⁹https://www.w3.org/TR/wai-aria-practices-1.1/#aria_ex
²⁰http://en.wikipedia.org/wiki/Web_Workers
²¹<http://johnbintz.github.com/jasmine-headless-webkit/>
²²<http://github.com/jasmine/jasmine>
²³<http://github.com/velesin/jasmine-jquery>
²⁴<http://github.com/jasmine/jasmine>
²⁵<http://github.com/velesin/jasmine-jquery>
²⁶<http://github.com/jasmine/jasmine>
²⁷<http://github.com/velesin/jasmine-jquery>
²⁸<https://github.com/jasmine/jasmine-ajax>
²⁹<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
³⁰<https://jstherightway.org/>
³¹<http://jslint.com>
³²<http://jslint.com>
³³<https://www.npmjs.com/package/node-style-guide>
³⁴<http://dom4j.sourceforge.net>
³⁵<http://nokogiri.org>
³⁶<http://misko.hevery.com/2010/07/14/how-javascript-works/>
³⁷<http://yehudakatz.com>
³⁸<http://yehudakatz.com/2012/01/10/javascript-needs-blocks/>
³⁹<http://yehudakatz.com/2011/08/11/understanding-javascript-function-invocation-and-this>
⁴⁰<http://jslint.com>
⁴¹<http://programmableweb.com>

Part II

**Agile Software
Development**

7

Requirements: Behavior-Driven Design and User Stories

Niklaus Wirth (1934–) received the Turing Award in 1984 for his pioneering contributions to structured programming, in which structured control flow constructs (`if/then/else`) and loops (`while` and `for`) improve the clarity and quality of code. Wirth also developed a sequence of innovative programming languages that embodied these concepts, including Algol-W, Euler, Modula, and Pascal.



Clearly, programming courses should teach methods of design and construction, and the selected examples should be such that a gradual development can be nicely demonstrated.

—Niklaus Wirth, “Program Development by Stepwise Refinement,” *CACM* 14(5), May 1971

7.1	Behavior-Driven Design and User Stories	206
7.2	SMART User Stories	209
7.3	Lo-Fi User Interface Sketches and Storyboards	211
7.4	Points and Velocity	213
7.5	Agile Cost Estimation	216
7.6	Cucumber: From User Stories to Acceptance Tests	217
7.7	CHIPS: Intro to BDD and Cucumber	220
7.8	Explicit vs. Implicit and Imperative vs. Declarative Scenarios	220
7.9	The Plan-And-Document Perspective on Documentation	223
7.10	Fallacies and Pitfalls	230
7.11	Concluding Remarks: Pros and Cons of BDD	233

Prerequisites and Concepts

Concepts:

The big concepts of this chapter are requirements elicitation, cost estimation, project scheduling, and monitoring progress.

The embodiment of these concepts for the Agile lifecycle, which follows *Behavior-Driven Development (BDD)*, are:

- *User stories* to elicit functional requirements.
- *Low-fidelity (Lo-Fi)* user interfaces and *storyboards* to elicit UI requirements.
- *Points* to turn user stories into cost estimates.
- *Velocity* to measure and estimate schedule.
- Using the tool *Cucumber* to transform user stories into acceptance tests.
- Using the tool *Pivotal Tracker* to track project progress, to calculate velocity, and to estimate time to milestones.

For the Plan-and-Document lifecycle, you will become familiar with the same concepts in a quite different format:

- Requirements elicitation via *interviewing, scenarios, and use cases*, requirements documentation via a *Software Requirements Specification (SRS)*, and requirements fulfillment using *requirements traceability*.
- Cost estimation based on project manager experience or formulas such as *CO-COMO*, scheduling and monitoring progress using *PERT charts*, and change management using *version control systems* for documentation and schedule as well as the code.
- *Risk analysis* and management to increase chances of project being successful.

Both lifecycles illustrate the difference between *functional* versus *non-functional requirements* and *explicit* versus *implicit requirements*.

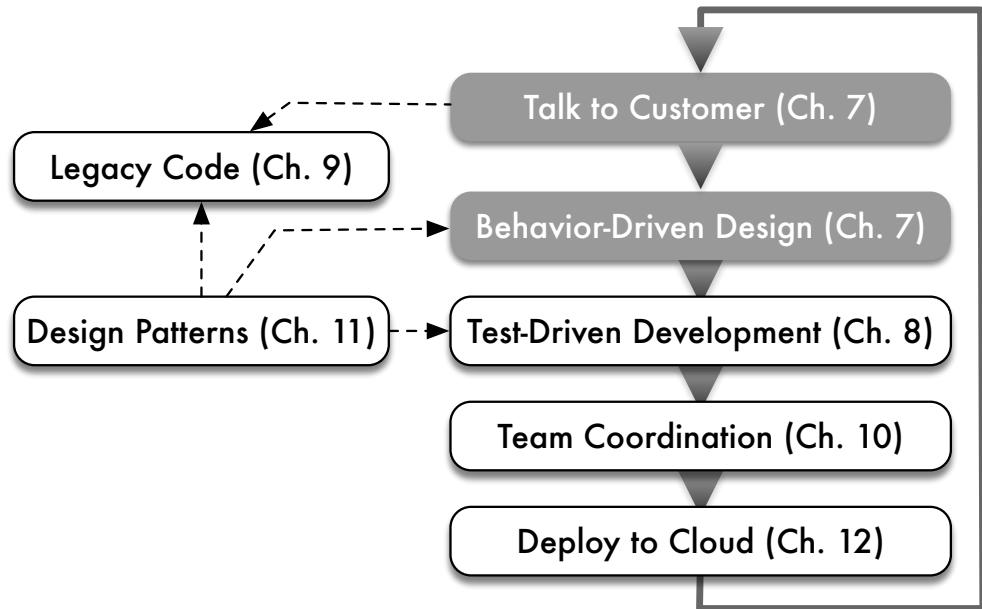


Figure 7.1: An iteration of the Agile software lifecycle and its relationship to the chapters in this book. This chapter emphasizes talking to customers as part of Behavior-Driven Design.

7.1 Behavior-Driven Design and User Stories

Behavior-Driven Design is Test-Driven Development done correctly.

—Anonymous

Software projects fail because they don't do what customers want; or because they are late; or because they are over budget; or because they are hard to maintain and evolve; or all of the above.

The Agile lifecycle was invented to attack these problems for many common types of software. Figure 7.1 shows one iteration of the Agile lifecycle from Chapter 1, highlighting the portion covered in this chapter. As we saw in Chapter 1, the Agile lifecycle involves:

Agile stakeholders include users, customers, developers, maintenance programmers, operators, project management,

- Working closely and continuously with stakeholders to develop requirements and tests.
- Maintaining a working prototype while deploying new features typically every two weeks—called an **iteration**—and checking in with stakeholders to decide what to add next and to validate that the current system is what they really want. Having a working prototype and prioritizing features reduces the chances of a project being late or over budget, or perhaps increasing the likelihood that the stakeholders are satisfied with the current system once the budget is exhausted!

Unlike a plan-and-document lifecycle in Chapter 1, Agile development does not switch phases (and people) over time from development mode to maintenance mode. With Agile, you are basically in maintenance mode as soon as you've implemented the first set of features. This approach helps make the project easier to maintain and evolve.

We start the Agile lifecycle with ***Behavior-Driven Design (BDD)***. BDD asks questions about the behavior of an application *before and during development* so that the stakeholders are less likely to miscommunicate. Requirements are written down as in plan-and-document, but unlike plan-and-document, requirements are continuously refined to ensure the resulting software meets the stakeholders' desires. That is, using the terms from Chapter 1, the goal of BDD requirements is ***validation*** (build the right thing), not just ***verification*** (build the thing right).

The BDD version of requirements is ***user stories***, which describe how the application is expected to be used. They are lightweight versions of requirements that are better suited to Agile. User stories help stakeholders plan and prioritize development. Thus, like plan-and-document, you start with requirements, but in BDD user stories take the place of design documents in plan-and-document.

By concentrating on the *behavior* of the application versus its *implementation*, it is easier to reduce misunderstandings between stakeholders. As we shall see in the next chapter, BDD is closely tied to Test-Driven Development (TDD), which *does* test implementation. In practice they work together hand-in-hand, but for pedagogical reasons we introduce them sequentially.

User stories came from the Human Computer Interface (HCI) community. They developed them using 3-inch by 5-inch index cards or “3-by-5 cards,” or in countries where metric paper sizes are used, A7 cards of 74 mm by 105 mm. (We'll see other examples of paper and pencil technology from the HCI community shortly.) These cards contain one to three sentences written in everyday nontechnical language written jointly by the customers and developers. The rationale is that paper cards are nonthreatening and easy to rearrange, thereby enhancing brainstorming and prioritizing. The general guidelines for the user stories themselves is that they must be testable, be small enough to implement in one iteration, and have business value. Section 7.2 gives more detailed guidance for good user stories.

Note that individual developers working by themselves without customer interaction don't need these 3-by-5 cards, but this “lone wolf” developer doesn't match the Agile philosophy of working closely and continuously with the customer.

We will use the RottenPotatoes app from Chapters 3 and 4 as the running example in this chapter and the next one. We start with the stakeholders for this simple app:

- The operators of RottenPotatoes, and
- The movie fans who are end-users of RottenPotatoes.

We'll introduce a new feature in Section 7.6, but to help understand all the moving parts, we'll start with a user story for an existing feature of RottenPotatoes so that we can understand the relationship of all the components in a simpler setting. The user story we picked is to add movies to the RottenPotatoes database:

<https://gist.github.com/1b759799a29f3b3e56686b32c6509ec1>

```

1 | Feature: Add a movie to RottenPotatoes
2 |   As a movie fan
3 |   So that I can share a movie with other movie fans
4 |   I want to add a movie to RottenPotatoes database
5 | Scenario: Add a movie
6 |   Given I am on the RottenPotatoes home page
7 |   When I follow "Add new movie"
8 |   Then I should be on the Create New Movie page
9 |   When I fill in "Title" with "Hamilton"
10 |  And I select "PG-13" from "Rating"
11 |  And I select "July 4, 2020" as the "Released On" date
12 |  And I press "Save Changes"
13 |  Then I should be on the RottenPotatoes home page
14 |  And I should see "Hamilton"
```

This user story format was developed by the startup company Connextra and is named after them; sadly, this startup is no longer with us. The format is:

<https://gist.github.com/f2be5cdde6a9d6c116a9877fb93aa0b9>

```

1 | Feature name
2 |   As a [kind of stakeholder],
3 |   So that [I can achieve some goal],
4 |   I want to [do some task]
```

This format identifies the stakeholder since different stakeholders may describe the desired behavior differently. For example, users may want links to information sources to make it easier to find the information, while operators may want links to trailers so that they can get an income stream from the advertisers. All three clauses have to be present in the Connextra format, but they do not have to be in this order.

Summary of BDD and User Stories

- BDD emphasizes working with stakeholders to define the behavior of the system being developed. Stakeholders include nearly everyone: customers, developers, managers, operators,
- **User stories**, a device borrowed from the HCI community, make it easy for non-technical stakeholders to help create requirements.
- 3 × 5 **cards** (or A7-size cards), each with a user story of one to three sentences, are a simple and nonthreatening technology that lets *all* stakeholders brainstorm and prioritize features.
- The Connextra format of user stories captures the stakeholder, the stakeholder's goal for the user story, and the task at hand.

■ Elaboration: User Stories and Case Analysis

User stories represent a lightweight approach to **use-case analysis**, a term traditionally used in software engineering to describe a similar process. A full use case analysis would include the use case name; actor(s); goals of the action; summary of the use case; preconditions (state of the world before the action); steps occurring in the scenario (both the actions performed by the user and the system's responses); related use cases; and postconditions (state of the world after the action). A **use case diagram** is a type of UML diagram (see Chapter 11) with stick figures standing in for the actors, and can be used to generalize or extend use cases or to include a use case by reference. For example, if we have a use case for “user logs in” and another use case for “logged-in user views her account summary”, the latter could include the former by reference, since a precondition to the second use case is that the user has logged in.

Self-Check 7.1.1. *True or False: User stories on 3x5 cards in BDD play the same role as design requirements in plan-and-document.*

◊ True. ■

7.2 SMART User Stories

What makes a good user story versus a bad one? The SMART acronym offers concrete and (hopefully) memorable guidelines: Specific, Measurable, Achievable, Relevant, and Timeboxed.

- **Specific.** Here is an example of a vague feature paired with a specific version:

<https://gist.github.com/c1d72e393df67e390453bc030d2caa37>

1	Feature: User can search for a movie (vague)
2	Feature: User can search for a movie by title (specific)

- **Measurable.** Adding Measurable to Specific means that each story should be testable, which implies that there are known expected results for some good inputs. Here is an example of an unmeasurable feature versus its measurable counterpart:

<https://gist.github.com/ae608d98061eebb1348f988b67ae5563>

1	Feature: RottenPotatoes should have good response time (unmeasurable)
2	Feature: When adding a movie, 99% of Add Movie pages
3	should appear within 3 seconds (measurable)

Only the second case can be tested to see if the system fulfills the requirement.

- **Achievable.** Ideally, you implement the user story in one Agile iteration. If you are getting less than one story per iteration, then they are too big and you need to subdivide these stories into smaller ones. As mentioned above, the tool **Pivotal Tracker** measures **Velocity**, which is the rate of completing stories of varying difficulty.
- **Relevant.** A user story must have business value to one or more stakeholders. To drill down to the real business value, one technique is to keep asking “Why.” Using as an example a ticket-selling app for a regional theater, suppose the proposal is to add a Facebook linking feature. Here are the “Five Whys” in action with their recursive questions and answers:



1. Why add the Facebook feature? As box office manager, I think more people will go with friends and enjoy the show more.
2. Why does it matter if they enjoy the show more? I think we will sell more tickets.
3. Why do you want to sell more tickets? Because then the theater makes more money.
4. Why does the theater want to make more money? We want to make more money so that we don't go out of business.
5. Why does it matter that the theater is in business next year? If not, I have no job.

(We're pretty sure the business value is now apparent to at least one stakeholder!)

- *Timeboxed*. Timeboxing means that you stop developing a story once you've exceeded the time budget. Either you give up, divide the user story into smaller ones, or reschedule what is left according to a new estimate. If dividing looks like it won't help, then you go back to the customers to find the highest value part of the story that you can do quickly.

The reason for a time budget per user story is that it is extremely easy to underestimate the length of a software project. Without careful accounting of each iteration, the whole project could be late, and thus fail. Learning to budget a software project is a critical skill, and exceeding a story budget and then refactoring it is one way to acquire that skill.

One important concept expands upon the R of SMART. The ***minimum viable product*** (MVP) is a subset of the full set of features that when completed has business value in the real world. Not only are the stories Relevant, but the combination of all of them makes the software product viable in the marketplace. Obviously, you can't start selling the product if it's not viable, so it makes sense to give priority to the stories that will let the product be shipped. The Epic or a Release point of Pivotal Tracker can help identify the stories of the MVP.

Summary of SMART User Stories

- The *SMART* acronym captures the desirable features of a good user story: Specific, Measurable, Achievable, Relevant, and Timeboxed.
- The ***Five Whys*** are a technique to help you drill down to uncover the real business relevance of a user story.

Self-Check 7.2.1. Which *SMART* guideline(s) does the feature below violate?
<https://gist.github.com/5eee5f0d00ca47c288c854a7a56fa60b>

1 | Feature: RottenPotatoes should have a good User Interface

- ◊ It is not Specific, not Measurable, not Achievable (within 1 iteration), and not Timeboxed. While business Relevant, this feature goes just one for five. ■

Self-Check 7.2.2. Rewrite this feature to make it *SMART*.

<https://gist.github.com/f284bab41b84680a27572cb1f6881fee>

1 | Feature: I want to see a sorted list of movies sold.

◊ Here is one SMART revision of this user story:

<https://gist.github.com/4058581d653f1e429c46e816db6c792c>

1 | Feature: As a customer, I want to see the top 10 movies sold,
2 | listed by price, so that I can buy the cheapest ones first.

■ Given user stories as the work product from eliciting requirements of customers, we can introduce a metric and tool to measure productivity.

7.3 Lo-Fi User Interface Sketches and Storyboards

We usually need to specify a user interface (UI) when adding a new feature since many SaaS applications interact with end users. Thus, part of the BDD task is often to propose a UI to match the user stories. If a user story says a user needs to login, then we need a mockup of a page that has the login. Alas, building software prototypes of user interfaces can intimidate stakeholders from suggesting improvements—just the opposite of the effect we need at this early point of the design.

What we want is the UI equivalent of 3x5 cards; engaging to the nontechnical stakeholder and encouraging trial and error, which means it must be easy to change or even discard. Just as the HCI community advocates 3x5 cards for user stories, they recommend using kindergarten tools for UI mockups: crayons, construction paper, and scissors. They call this low-tech approach to user interfaces **lo-fi (low-fidelity) UI** and the paper prototypes *sketches*. Ideally, you make sketches for all the user stories that involve a UI. It may seem tedious, but eventually you are going to have to specify all the UI details when using HTML to make the real UI, and it's a lot easier to get it right with pencil and paper than with code.

A lo-fi sketch shows what the UI looks like at one instant in time. However, we also need to show how the sketches work together as a user interacts with a page. Filmmakers face a similar challenge with scenes of a movie. Their solution, which they call **storyboarding**, is to go through the entire film as if it was a comic book, with drawings for every scene. Instead of a linear sequence of images like in a movie, the storyboard for a UI is typically a tree or graph of screens driven by different user choices.

To make a storyboard, you must think about all the user interactions with a web app:

- Pages or sections of pages,
- Forms and buttons, and
- Popups.

Figure 7.2 shows a storyboard composed of lo-fi sketches for adding a new movie to RottenPotatoes. The storyboard includes indications of what the user clicks to cause the transitions between sketches. After drawing the sketches and storyboards, you are ready to write HTML. Chapter 3 showed how Erb markup becomes HTML, and how the `class` and `id` attributes of HTML elements can be used to attach styling information to them via Cascading Style Sheets (CSS). The key to the lo-fi approach is to get a good overall structure from your sketches, and do minimal CSS (if any) to get the view to look more or less like

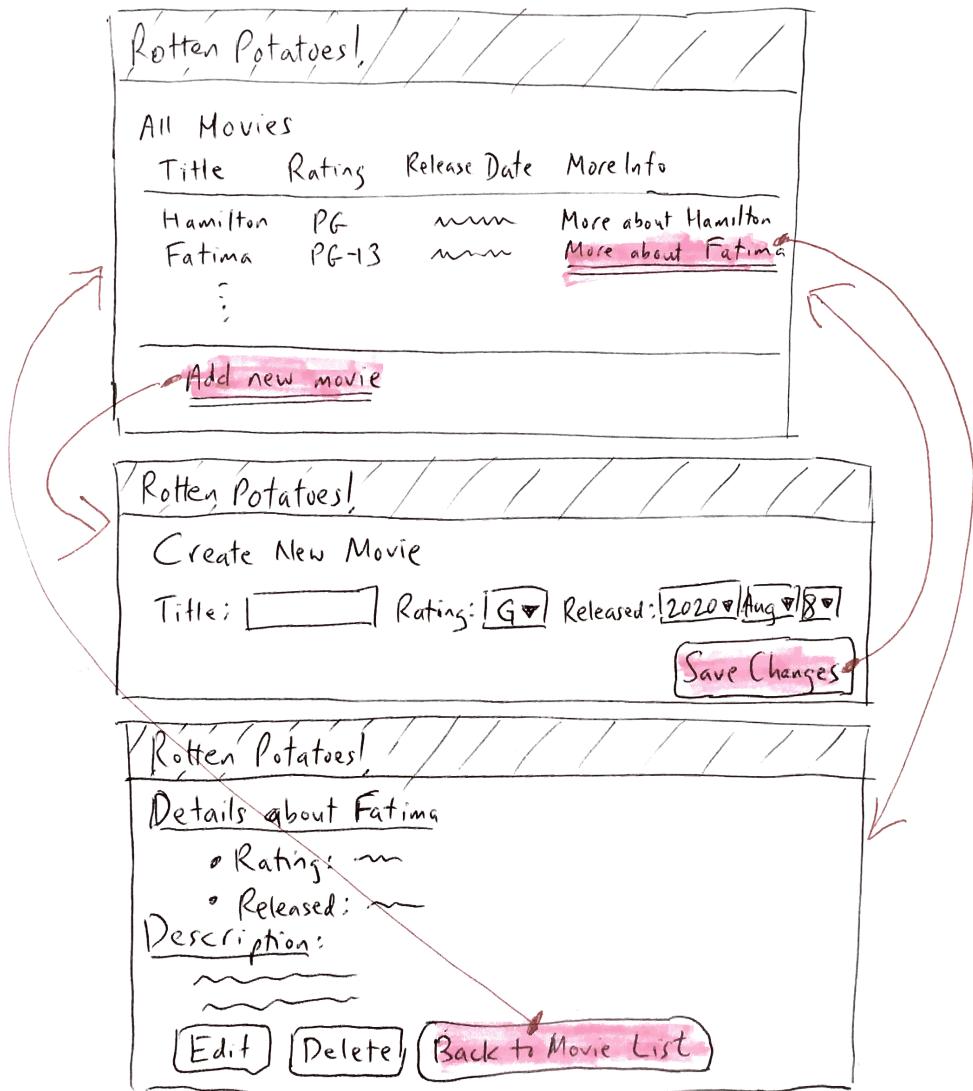


Figure 7.2: A storyboard illustrating two simple user stories about RottenPotatoes. Top: RottenPotatoes home page, listing all movies; the starting point for both stories. Middle: Screen that should appear when adding a movie to RottenPotatoes. Bottom: Screen that should appear when viewing details of an existing movie. The red-shaded words on each page are clickable links or buttons; the arrows show what screen should be displayed next if that link or button is clicked.

your sketch. Remember that the common parts of the page layout—banners, structural `divs`, and so on—can go into `views/layouts/application.html.erb`.

Start the process by looking at the lo-fi UI sketches and split them into “blocks” of the layout. Use HTML `divs` for obvious layout sections. There is no need to make it pretty until after you have everything working. Adding CSS styling, images, and so on is the fun part, but make it look good *after* it works. One reason we have repeatedly advocated for the use of CSS frameworks such as Bootstrap in this book is that they facilitate producing an acceptable-looking prototype quickly.

Summary: Borrowing from the HCI community once again, ***lo-fi sketches*** are low cost ways to explore the user interface of a user story. Paper and pencil makes them easy to change or discard, which once again can involve all stakeholders. ***Storyboards*** capture the interaction between different pages depending on what the user does. It is much less effort to experiment in this low cost medium before using HTML and CSS to create the pages you want.

Self-Check 7.3.1. True or False: *The purpose of lo-fi UI sketches and storyboards is to debug the UI before you program it.*

- ◊ True. ■

7.4 Points and Velocity

One way to measure the productivity of a team would be simply to count the number of user stories completed per iteration, and then calculate the average. The average would then be used to decide how many stories to try to implement each iteration.

The problem with this measure is that some stories are much harder than others, leading to mispredictions. The simple solution is to give each user story an integer number of *points* reflecting its perceived difficulty. The “value” of a point—the approximate expected number of coding hours it represents—is completely up to the team, and will likely differ across teams, but the point scale should have two important properties. First, everyone on the team should be in rough agreement on how much a “point” is worth. Second, more points should represent not only more effort, but more uncertainty. For example, your team might start with a simple 3-point scale in which 1 point represents approximately a 3-hour work session. Your team might be good at estimating the effort required to complete a 1 or 2 point story this way, but can you confidently estimate that a 3-point story will really take 9 hours of work? For this reason, your team should also set a threshold above which a story *must* be broken down into smaller tasks before estimating its difficulty, until each task is sufficiently well understood that it can be estimated with high confidence. Therefore, in our simple suggested introductory scheme, you might decide that any story estimated at higher than 3 points must be subdivided into stories that everyone agrees are 3 points or less.

A practical way to estimate points that also builds the team’s *collective ownership* (knowledge of different parts of the project being diffused around the team) is known as ***planning poker***. During an Iteration Planning Meeting at the beginning of an iteration, the team first prioritizes the stories according to the stated desires of the customer (or the Product Owner speaking for the customer). Each story is discussed in turn: the Project Manager reads and reviews the story to ensure everyone understands what the story requires, then each team member places a card face-down marked with the number of points they think that story

Fibonacci scale With more experience, the Fibonacci scale is commonly used: 1, 2, 3, 5, and 8. (Each new number is sum of previous two.) However, at places like Pivotal Labs, 8 is extremely rare.

should be worth. An even easier variation is to have everyone simultaneously stick out 1 to 5 fingers, in the style of the children’s game Rock–Paper–Scissors. There should be a card (or hand gesture) that means “I don’t know” and another that means “This story is too complicated and should be broken down.” The team then discusses differences in the votes to reach consensus, and they vote again, possibly after subdividing the story. An inability to reach consensus may indicate a story that isn’t SMART.

When should a story get more points? Some stories may require information-gathering, such as becoming familiar with other parts of the codebase or doing some scouting to determine which files or classes in the app will be affected by the proposed feature. A major source of uncertainty, such as figuring out how to integrate a new technology or library, should get its own **spike**: a short investigation into a technique or problem that the team wants explored before sitting down to do serious coding. An example would be a spike on incorporating recommendations into an app, in which a developer or pair investigates different algorithms and different libraries that could be used, possibly using a scratch branch of the code (which we discuss in Section 10.2) to do some basic testing and exploration. After a spike is done, the spike code must be thrown away: The spike’s purpose is to help you determine what approach you want to follow, and now that you know, you should write it correctly.

The **backlog** is the collection of stories that have been prioritized and assigned points in this way, but have not yet been started in this iteration. The team then begins *delivering the backlog*, that is, working on the stories in priority order during the iteration. (Section 10.4 presents best practices for coordinating this work.) At the end of the iteration, the team computes the total number of *points* completed, rather than the number of stories. The moving average of this total is called the team’s **velocity**.

Velocity measures work rate based on the team’s self-evaluation. As long as the team rates user stories consistently, it doesn’t matter whether the team is completing 5 points or 10 points per iteration. The purpose of velocity is to give all stakeholders an idea how many iterations it will take a team to add the desired set of features, which helps set reasonable expectations and reduces chances of disappointment. Points and velocity are often used as the basis of a **burn down chart**, which shows work to be done (points) on the vertical axis and time along the horizontal axis. The slope of the downward-pointing line is the team’s velocity, and the line’s intersection with the x-axis represents the prediction for when the work will be done.

Figure 7.3 shows the UI of Pivotal Tracker¹, a Web-based tool that allows a team to enter and prioritize user stories with point values, assign stories to developers, attach design documents such as lo-fi mockups to stories, and perhaps most importantly, track points and velocity as stories are delivered, optionally generating a variety of analytics including burn down charts. Tracker also provides an Icebox panel, which contains unprioritized stories. They can stay “on ice” indefinitely, but when you’re ready to start working on them, just drag them to the Current or Backlog panels. Tracker provides a way to enter a spike and prioritize it relative to other stories, so the team knows that certain stories cannot be completed until the spike is done. Similarly, one story can be marked as being *blocked* by another, indicating a dependency that must be taken into account when arranging the backlog in priority order. Finally, since complex stories representing a single “feature” should be broken down into smaller stories, Tracker provides Epics as a way of grouping related stories and tracks how many total points are still needed to complete the epic, regardless of how the stories are ordered in the backlog. The idea is to give software engineers the big picture of where the application is in the development process with regard to big features.



Tracker intro Pivotal Labs has produced an excellent 3-minute video intro² to using Tracker.

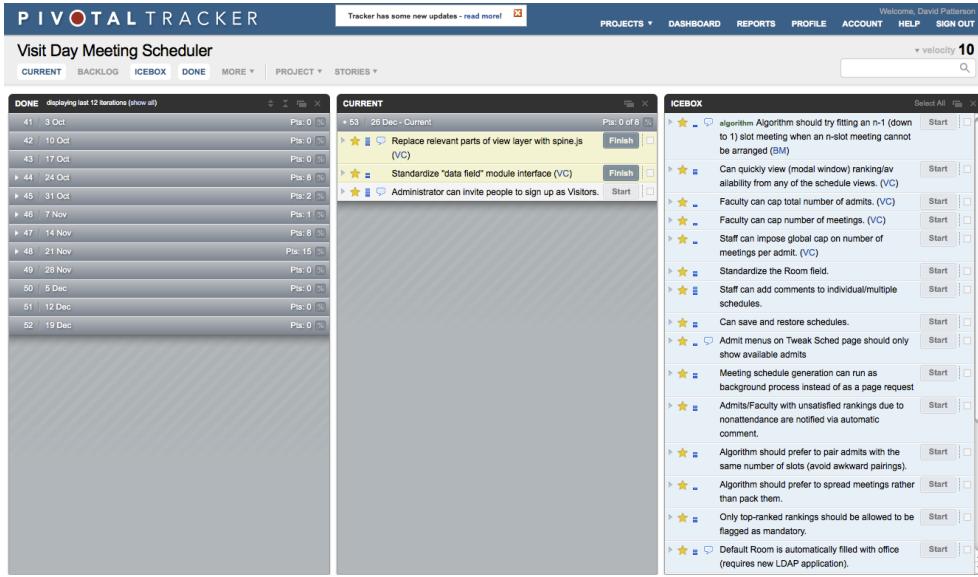


Figure 7.3: Screen image of the UI of the Pivotal Tracker service.

Because Tracker calculates velocity based on points completed, it groups the remaining prioritized stories in the backlog into iterations based on the assumption that velocity will remain approximately constant. These estimates can be useful in setting customer expectations as to when a particular feature will be delivered. Tracker even allows the insertion of Release markers, which bound the stories that must all be delivered before a particular feature is completely working and ready to announce to the customer. (We will have more to say about releases in Section 12.4.) This approach is in sharp contrast to management by schedule, in which a manager picks a release date and the team is expected to work hard to meet the deadline.

Some teams use GitHub Issues³ to track stories as a to-do list, or a project management tool such as Trello⁴ to put virtual story cards on a virtual wall. These simpler tools are fine to start out with, but they lack the ability to track points and velocity, and therefore to supply estimates of story completion time in more complicated projects. In addition, Tracker is a good place to centralize information about the project—design notes, architecture diagrams, lo-fi sketches, and so on—because you can associate it with individual user stories and even link out to documents in Google Docs or other places. Every GitHub repository also includes a Wiki, which allows team members to jointly edit a document and add files. Whatever tool you choose, the important thing is to keep all documentation about the project accessible from one place that the whole team can agree on, and which will remain stable even as members of the team come and go.

Summary of points and velocity:

- To help the team manage each iteration and to predict how long the team will take to implement new features, the team assigns points to rate difficulty of user stories and tracks the team's **velocity**, or average points per iteration.
- Techniques such as planning poker, in which team members simultaneously "vote" on the difficulty of a story and then discuss discrepancies, are a quick and practical way to estimate points while diffusing knowledge of the project throughout the team.
- Pivotal Tracker provides a service that helps prioritize and keep track of user stories and their status, calculates velocity, and predicts software development time based on the team's history.

Self-Check 7.4.1. *True or False: When comparing two teams, the one with the higher velocity is more productive.*

◊ False: Since each team assigns points to user stories, you cannot use velocity to compare different teams. However, you could look over time for a given team to see if there were iterations that were significantly less or more productive. ■

Self-Check 7.4.2. *True or False: When you don't know how to approach a given user story, just give it 3 points.*

◊ False: A user story should not be so complex that you don't have an approach to implementing it. If they are, you should go back to your stakeholders to refactor the user story into a set of simpler tasks that you do know how to approach. ■

7.5 Agile Cost Estimation

Given that the Agile Manifesto values customer collaboration over contract negotiation, it is unsurprising that it does *not* follow the plan-and-document approach of making a cost estimate and schedule for a given set of features as part of a bid to win a contract, as we shall see in Section 7.9). This section describes the process at Pivotal Labs, which relies upon Agile development (Burkes 2012).

Because Pivotal does Agile, Pivotal never commits to delivering features X, Y, and Z by date D. Pivotal commits to providing a certain amount of resources to work in the most efficient way possible up to date D. Along the way, Pivotal needs the client to work with the project team to define priorities, and let Tracker's velocity guide the decisions as to which features actually make it into the release on date D.

A potential client first gets in contact with the Agile team. If it looks like a good fit for the Agile team, they do a 30 to 60 minute phone call telling the potential client what an engagement looks like, how it's different from other "outsourcing" agencies, what type of time commitment it will require on the customer's part, and so on. This first call makes clear that the Agile team works on a time and materials basis, not on a fixed bid basis, as is usually the case with plan-and-document processes. The Agile team gets them to describe at a high level what they want developed, what their current development process looks like, what their current staffing is, and so on.

Pivotal Labs is a software consultancy that teaches clients the Agile lifecycle while collaborating with them to develop a specific software product.

If the client is comfortable with what they heard, and the Agile team thinks it still sounds like a good fit, the client visits for what Pivotal calls a “scoping.” A scoping is a roughly 90 minute conversation with a potential client, preferably in person. The Agile team asks the client to bring the person responsible for the product, a lead developer if they have one, a designer if they have one, any existing designs for what they want built, and so on. Basically, the client representatives bring whatever they think can clarify exactly what they want done, and the Agile team brings two engineers to the scoping.

During the scoping, the Agile team asks the client to describe what they want done in detail, and they ask a series of questions designed to identify unknowns, risks, external integrations, and so forth. Essentially, the Agile team wants to identify anything that would add uncertainty to the estimate that the Agile team will deliver. If the Agile team gets a client with a very clear definition of what they want to build, a finished design, no external integrations, and so on, the Agile team can produce a fairly tightly-scoped estimate, such as “20 to 22 weeks.” On the other hand, if they don’t have clear product definition, lots of external integrations, or other uncertainty, the Agile team’s estimate will have a greater range, such as “18 to 26 weeks.” If you use pair programming (see Section 2.2), as Pivotal Labs does, the cost estimates would be in “pair weeks.”

After the client leaves the scoping, the Pivots (engineers) involved will stay behind for another 15 to 30 minutes, and agree on an estimate in terms of weeks. They deliver their findings, which include the estimate, identification of risks, and so on, to the sales staff, who then turn that into a proposal email to the client.

Because the Agile team does time and materials only, it’s easy to turn estimated weeks into an estimated range of expense.

Summary: Following the Agile Manifesto’s emphasis on customer cooperation over contracts, an Agile team’s notion of “cost estimation” is therefore more about advising the client on what team size can provide the maximum efficiency, following Brooks’s Law that there is a point of diminishing returns on team size (see Section 7.10). The Agile team’s goal in the scoping process is to identify that point, then ramp the team up to that size over time. Agile companies bid costs for time and materials based on short discussions with external customers. As we shall see in Section 7.9, this approach is in sharp contrast with companies that follow plan-and-document processes, which promise customers a set of features for an agreed upon cost by an agreed upon date.

Self-Check 7.5.1. *True or False: As practitioners of Agile Development, Pivotal Labs does not use contracts.*

- ◊ False. Pivotal certainly offers customers a contract that they sign, but it is primarily a promise to pay Pivotal for its best effort to make the customer happy over a limited range of time. ■

With the already helpful role of user stories for measuring progress behind us, we introduce a tool that lets user stories play yet another important role.

7.6 Cucumber: From User Stories to Acceptance Tests

Remarkably enough, the tool **Cucumber** turns customer-understandable user stories into **acceptance tests**, which ensure the customer is satisfied, and **integration tests**, which

<https://gist.github.com/1b759799a29f3b3e56686b32c6509ec1>

```

1 Feature: Add a movie to RottenPotatoes
2   As a movie fan
3   So that I can share a movie with other movie fans
4   I want to add a movie to RottenPotatoes database
5 Scenario: Add a movie
6   Given I am on the RottenPotatoes home page
7   When I follow "Add new movie"
8   Then I should be on the Create New Movie page
9   When I fill in "Title" with "Hamilton"
10  And I select "PG-13" from "Rating"
11  And I select "July 4, 2020" as the "Released On" date
12  And I press "Save Changes"
13  Then I should be on the RottenPotatoes home page
14  And I should see "Hamilton"

```

Figure 7.4: A Cucumber scenario associated with the adding a movie feature for RottenPotatoes.

ensure that the interfaces between modules have consistent assumptions and communicate correctly. (Chapter 1 describes types of testing.) The key is that Cucumber meets halfway between the customer and the developer: user stories don't look like code, so they are clear to the customer and can be used to reach agreement, but they also aren't completely free-form. This section explains how Cucumber accomplishes this minor miracle.

In the Cucumber context we will use the term **user story** to refer to a single **feature** with one or more **scenarios** that show different ways a feature is used. The keywords **Feature** and **Scenario** identify the respective components. Each scenario is in turn composed of a sequence of typically 3 to 8 **steps**. Expanding a user story into a set of scenarios also helps developers enumerate the various user-visible conditions that will be tested to ensure the feature works. For example, consider a fictitious e-commerce site that wants developers to implement the feature *Customer can use “guest checkout” to make a purchase without creating an account*. This might be broken down into several scenarios:

- Customer can complete a purchase as guest
- Customer cannot do a guest purchase if the order includes a gift
- Multiple guest checkouts associated with same email address group the orders into the same account

Notice in particular the third scenario, which might arise from conversation with the customer while discussing the feature: “Well, what happens if the same email address appears on multiple orders but that user has no account? Should we associate those orders with the same customer internally?” Indeed, such discussions are vital to fleshing out ambiguities in the customer’s desired features.

Figure 7.4 is an example user story, showing a feature with one scenario of adding the movie *Hamilton*; the scenario has nine steps. (We show just a single scenario in this example, but features usually have many scenarios.) Although stilted writing, this format that Cucumber can act upon is still easy for the nontechnical customer to understand, providing a common representation of the story on which the customer and team can now collaborate—a founding principle of Agile and BDD.

Each step of a scenario starts with its own keyword. Steps that start with **Given** usually set up some preconditions, such as navigating to a page. Steps that start with **When** typically

Cucumber keywords

Given, When, Then, And, and But have different names just for the benefit of human readers, but they are all aliases to the same method.

use one of Cucumber’s built-in web steps to simulate the user pressing a button, for example. Steps that start with `Then` will usually check to see if some condition is true. The conjunction `And` allows more complicated versions of `Given`, `When`, or `Then` phrases. The only other keyword you see in this format is `But`.

A separate set of files defines the Ruby code that tests these steps. These are called *step definitions*. How does Cucumber match each step of a scenario with the correct step definitions? The trick is that Cucumber uses regular expressions or **regexes** (Chapter 2) to match the phrases in the scenario steps to the step definitions themselves. For example, below is a string from a step definition in the scenario for RottenPotatoes:

<https://gist.github.com/8873bc7dc65d123f752e37fdb701ea6e>

```
1 | Given /^(?:|I )am on (.+)$/
```

This regex can match the text “I am on the RottenPotatoes home page” on line 6 of Figure 7.4. The regex also captures the string after the phrase “am on ” until the end of the line (“the RottenPotatoes home page”). The body of the step definition contains Ruby code that tests the step, likely using captured strings such as the one above. Thus, most step definitions are typically used by many different steps. You can think of step definitions as method definitions, and the steps of the scenarios are analogous to method calls.

We then need a tool that will act as a user and pretend to use the feature under different scenarios. In the Rails world, this tool is called *Capybara*, and Cucumber integrates seamlessly with it. Capybara “pretends to be a user” by taking actions in a simulated web browser, for example, clicking on a link or button. Capybara can interact with the app to receive pages, parse the HTML, and submit forms as a user would. In the rest of this chapter and its associated CHIPS, you will write your own steps to describe the app’s behavior, then connect the steps to step definitions that actually stimulate the app to instantiate the behaviors—the core of Behavior-Driven Design.

Finally, the simple scenario above only describes one particular **happy path** of the feature in question, but it is also important to agree with the customer on what should happen when things go wrong. For example, if the user leaves the movie title blank, we would probably want to redisplay the Create New Movie page, but perhaps with an error message informing the user of what went wrong. This “sad path” would get its own scenario in the feature file and its own storyboard, since describing what happens when things go wrong is part of the overall feature.



Summary of Cucumber Introduction

- **Cucumber** lets you use a stylized, restricted form of English to describe a set of user stories, or **scenarios**, that collectively describe a feature.
- The steps of a Cucumber scenario use the keyword `Given` to describe the current state, `When` to identify user actions, and `Then` to describe the intended consequences of those actions.
- Each scenario step is matched to a step definition using **regular expressions**. A typical step definition uses the browser simulator *Capybara* to simulate a user’s actions corresponding to that step, or interrogates the app to check if the desired consequences of the user’s actions have occurred.

■ *Elaboration: Stubbing the web*

The way we use Cucumber and Capybara in this chapter doesn't allow us to test JavaScript code, which is covered in Chapter 6. With appropriate options, Cucumber can control Webdriver, which actually fires up a *real* browser and "remote controls" it to make it do what the stories say, including all JavaScript code. For this chapter, we will use Capybara's "headless browser simulator" mode, which is much faster and is appropriate for testing everything except JavaScript.

Self-Check 7.6.1. *Given that Cucumber step definitions are just Ruby code, in principle we could just write the entire scenario in Ruby, rather than writing steps in stilted English and looking up the step definition for each step. Why do you think Cucumber has remained popular despite this fact?*

- ◊ The customer can (probably) read the Cucumber scenario steps and understand the description of what the app is supposed to do, and can determine whether they agree with that description. Most customers would find it much more difficult to read Ruby code. Thus the scenarios provide a common ground on which the technical team and customer can meet. ■

7.7 CHIPS: Intro to BDD and Cucumber



CHIPS 7.7: BDD With Cucumber

<https://github.com/saasbook/hw-bdd-cucumber>

In this exercise you'll write Cucumber scenarios to both test existing features and drive the creation of new features in the RottenPotatoes app.

7.8 Explicit vs. Implicit and Imperative vs. Declarative Scenarios

Now that we have seen user stories and Cucumber in action, we are ready to cover two important testing topics that involve contrasting perspectives.

The first is *explicit versus implicit requirements*. A large part of the formal specification in plan-and-document is requirements, which in BDD are user stories developed by the stakeholders. Using the terminology from Chapter 1, they typically correspond to acceptance tests. Implicit requirements are the logical consequence of explicit requirements, and typically correspond to what Chapter 1 calls integration tests. An example of an implicit requirement in RottenPotatoes might be that by default movies should be listed in chronological order by release date.

The good news is that you can use Cucumber to kill two birds with one stone—create acceptance tests *and* integration tests—if you write user stories for both explicit and implicit requirements. (The next chapter shows how to use another tool for unit testing.)

The second contrasting perspective is *imperative versus declarative scenarios*. The example scenario in Figure 7.4 above is imperative, in that you are specifying a logical sequence of user actions: filling in a form, clicking on buttons, and so on. Imperative scenarios tend to have complicated When statements with lots of And steps. While such scenarios are useful

in ensuring that the details of the UI match the customer's expectations, it quickly becomes tedious and non-DRY to write most scenarios this way.

To see why, suppose we want to write a feature that specifies that movies should appear in alphabetical order on the list of movies page. For example, "Zorro" should appear after "Apocalypse Now", even if "Zorro" was added first. As Figure 7.5(a) shows, it would be the height of tedium to express this scenario naively, because it mostly repeats lines from our existing "add movie" scenario—not very DRY. Cucumber is supposed to be about *behavior* rather than implementation—focusing on *what* is being done—yet in this poorly-written scenario, only line 18 mentions the behavior of interest!

An alternative approach is to think of using the step definitions to make a *domain language* (which is different from a formal **Domain Specific Language (DSL)**) for your application. A domain language is informal but uses terms and concepts specific to your application, rather than generic terms and concepts related to the implementation of the user interface. Steps written in a domain language are typically more declarative than imperative in that they describe the state of the world rather than the sequence of steps to get to that state and they are less dependent on the details of the user interface. Figure 7.5(b) shows what a declarative version of the above scenario might look like using a domain language for RottenPotatoes. The declarative version is obviously shorter, easier to maintain, and easier to understand since the text describes the state of the app in a natural form: "I am on the RottenPotatoes home page sorted by title."

The good news is that, as Figure 7.5(c) shows, you can *reuse* existing imperative steps to implement such scenarios. This is a very powerful form of reuse, and as your app evolves, you will find yourself reusing steps from your first few imperative scenarios to create more concise and descriptive declarative scenarios. Declarative, domain-language-oriented scenarios focus the attention on the feature being described rather than the low-level steps you need to set up and perform the test.



Summary:

- We can use Cucumber for both acceptance and integration testing if we write user stories for both explicit and implicit requirements. Declarative scenarios are simpler, less verbose, and more maintainable than imperative scenarios.
- As you get more experienced, the vast majority of your user stories should be in a domain language that you have created for your app via your step definitions, and the stories should worry less about user interface details. The exception is for the specific stories where there is business value (customer need) in expressing the details of the user interface.

<https://gist.github.com/2aa894c7bad7c813ecae447b6a97b059>

```

1 Feature: movies should appear in alphabetical order, not added order
2
3 Scenario: view movie list after adding 2 movies (imperative and non-DRY)
4
5   Given I am on the RottenPotatoes home page
6   When I follow "Add new movie"
7   Then I should be on the Create New Movie page
8   When I fill in "Title" with "Zorro"
9   And I select "PG" from "Rating"
10  And I press "Save Changes"
11  Then I should be on the RottenPotatoes home page
12  When I follow "Add new movie"
13  Then I should be on the Create New Movie page
14  When I fill in "Title" with "Apocalypse Now"
15  And I select "R" from "Rating"
16  And I press "Save Changes"
17  Then I should be on the RottenPotatoes home page
18  Then I should see "Apocalypse Now" before "Zorro" on the RottenPotatoes home
     page sorted by title

```

<https://gist.github.com/548b8f5a3a62a5d6f2b600d0ae91ae30>

```

1 Feature: movies should appear in alphabetical order, not added order
2
3 Scenario: view movie list after adding movie (declarative and DRY)
4
5   Given I have added "Zorro" with rating "PG-13"
6   And I have added "Apocalypse Now" with rating "R"
7   Then I should see "Apocalypse Now" before "Zorro" on the RottenPotatoes
     home page sorted by title

```

<https://gist.github.com/068e25c2428df4dbf731133e444f1926>

```

1 Given /I have added "(.*)" with rating "(.*)" / do |title, rating|
2   steps %Q{
3     Given I am on the Create New Movie page
4     When I fill in "Title" with "#{title}"
5     And I select "#{rating}" from "Rating"
6     And I press "Save Changes"
7   }
8 end
9
10 Then /I should see "(.*)" before "(.*)" on (.*) / do |string1, string2, path|
11   steps %Q{Given I am on #{path}}
12   regexp = /#{string1}.*#{string2}/m # /m means match across newlines
13   expect(page.body).to match(regexp)
14 end

```

Figure 7.5: Top (a): A repetitive, non-DRY scenario for checking the alphabetical ordering of movies in the list. Middle (b): A DRYer, more declarative expression of the same scenario. Bottom (c): Adding this code to `movie_steps.rb` creates new step definitions matching lines 5–7 of the declarative scenario by reusing existing steps. (Recall from Figure 2.2 that `%Q` is an alternative syntax for double-quoting a string.) We will learn about `expect`, which appears in line 13, in the next chapter.

■ Elaboration: The BDD ecosystem

There is enormous momentum, especially in the Ruby community where testable, beautiful, and self-documenting code is highly valued, to document and promote best practices for BDD. Good scenarios serve as both documentation of the app designers' intent and executable acceptance and integration tests; they therefore deserve the same attention to beauty as the code itself. For example, this free screencast from RailsCasts⁵ describes *scenario outlines*, a way to DRY out a repetitive set of happy or sad paths whose expected outcomes differ based on how a form is filled in, similar to the contrast between our happy and sad paths above. The Cucumber wiki⁶ is a good place to start, but as with all programming, you'll learn BDD best by doing it often, making mistakes, and revising and beautifying your code and scenarios as you learn from your mistakes.

Self-Check 7.8.1. *True or False: Explicit requirements are usually defined with imperative scenarios and implicit requirements are usually defined with declarative scenarios.*

- ◊ False. These are two independent classifications; both requirements can use either type of scenario. ■

7.9 The Plan-And-Document Perspective on Documentation

As is well known to software engineers (but not to the general public), by far the largest class of [software] problems arises from errors made in the eliciting, recording, and analysis of requirements.

—Daniel Jackson, Martyn Thomas, and Lynette Millett (Editors), *Software for Dependable Systems: Sufficient Evidence?*, 2007

Recall that the hope for plan-and-document methods was to make software engineering as predictable in budget and schedule as civil engineering. Remarkably, user stories, points, and velocity correspond to *seven* major tasks of the plan-and-document methodologies. They include:

1. Requirements Elicitation
2. Requirements Documentation
3. Cost Estimation
4. Scheduling and Monitoring Progress

It's worth recalling that *novel* civil engineering projects, or modifications after the fact, often suffer cost and time overruns as well. Civil engineers tend to have better success predicting project outcomes when the project is similar to others that have been successfully completed in the past.

These are done up front for the Waterfall model and at the beginning of each major iteration for the Spiral and RUP models. As requirements change over time, these items above imply other tasks:

5. Change Management for Requirements, Cost, and Schedule
6. Ensuring Implementation Matches Requirement Features

Finally, since accuracy of the budget estimate and the schedule is vital to the success of the plan-and-document process, there is another task not found in BDD:

7. Risk Analysis and Management

The hope is that by imagining all the risks to the budget and schedule in advance, the project can make plans to avoid or overcome them.

As we shall see in Chapter 10, the plan-and-document processes assume that each project has a manager. While the whole team may participate in requirements elicitation and risk analysis and help document them, it is up to the project manager to estimate costs, make and maintain the schedule, and decide which risks to address and how to overcome or avoid them.

Advice for project managers comes from all corners, from practitioners who offer guidelines and rules of thumb based on their experience to researchers who have measured many projects to come up with formulas for estimating budget and schedule. There are also tools to help. Despite this helpful advice and tools, the project statistics from Chapter 1 (Johnson 1995, 2009)—that 40% to 50% of projects exceed the budget and schedule by factors of 1.7 to 3.0, and that 20% to 30% of projects are cancelled or abandoned—document the difficulty of making accurate budgets and schedules.

We now give quick overviews of these seven tasks so that you can be familiar with what is done in plan-and-document processes to give you a head start if you need to use them in the future. These overviews help explain the inspiration for the Agile Manifesto. If you are unclear on how to successfully perform these tasks, it may be due more to their inherent difficulties rather than to brevity.

1. Requirements Elicitation. Like User Stories, requirements elicitation involves participation by all stakeholders, using one of several techniques. The first is *interviewing*, where stakeholders answer predefined questions or just have informal discussions. Note that one goal is to understand the social and organization environment to see how tasks are *really* done versus the official story. Another technique is to cooperatively create **scenarios**, which can start with an initial assumption of the state of the system, show the flow of the system for a happy case and a sad case, list what else is going on in the system, and then the state of the system at the end of the scenario. Related to scenarios and user stories, a third technique is to create **use cases**, which are lists of steps between a person and a system to achieve a goal (see the elaboration in Section 7.1).

In addition to **functional requirements** such as those listed above, **non-functional requirements** include performance goals, dependability goals, and so on.

2. Requirements Documentation. Once elicited, the next step is to document the requirements in a **Software Requirements Specification (SRS)**. Figure 7.6 gives an outline for an SRS based on IEEE Standard 830-1998. A SRS for a patient management system⁷ is 14 pages long, but they are often hundreds of pages.

Part of the process is to check the SRS for:

- Validity—are all these requirements really necessary?
- Consistency—do requirements conflict?
- Completeness—are all requirements and constraints included?
- Feasibility—can the requirements really be implemented?

Techniques to test for these four characteristics include having stakeholders—developers, customers, testers, and so on—proofread the document, trying to build a prototype that includes the basic features, and generating test cases that check the requirements.

Table of Contents

- 1. Introduction
 - 1.1 Purpose
 - 1.2 Scope
 - 1.3 Definitions, acronyms, and abbreviations
 - 1.4 References
 - 1.5 Overview
- 2. Overall description
 - 2.1 Product perspective
 - 2.2 Product functions
 - 2.3 User characteristics
 - 2.4 Constraints
 - 2.5 Assumptions and dependencies
- 3. Specific requirements
 - 3.1 External interface requirements
 - 3.1.1 User interfaces
 - 3.1.2 Hardware interfaces
 - 3.1.3 Software interfaces
 - 3.1.4 Communication interfaces
 - 3.2 System features
 - 3.2.1 System feature 1
 - 3.2.1.1 Introduction/purpose of feature
 - 3.2.1.2 Stimulus/response sequence
 - 3.2.1.3 Associated function requirements
 - 3.2.1.3.1 Functional requirement 1
 - ...
 - 3.2.1.3.n Functional requirement n
 - 3.2.2 System feature 2
 - ...
 - 3.2.m System feature m
 - 3.3 Performance requirements
 - 3.4 Design constraints
 - 3.5 Software system attributes
 - 3.6 Other requirements

Figure 7.6: A table of contents for the IEEE Standard 830-1998 recommended practice for Software Requirements Specifications. We show Section 3 organized by feature, but the standard offers many other ways to organize Section 3: by mode, user class, object, stimulus, functional hierarchy, or even mixing multiple organizations.

A project may find it useful to have two types of SRS: a high-level SRS that is for management and marketing and a detailed SRS for the project development team. The former is presumably a subset of the latter. For example, the high-level SRS might leave out the functional requirements that correspond to 3.2.1.3 in Figure 7.6.

■ ***Elaboration: Formal specification languages***

Formal specification languages such as Alloy or Z allow the project manager to write executable requirements, which makes it easier to validate the implementation. Not surprisingly, the cost is both a more difficult document to write and usually a much longer requirements document to read. The advantage is both precision in the specification and the potential to automatically generate tests cases or even use formal methods for verification of correctness (see Section 8.10).

3. Cost Estimation. The project manager then decomposes the SRS into the tasks to implement it, and then estimates the number of weeks to complete each task. The advice is to decompose no finer than one week. Just as a user story with more than seven points should be divided into smaller user stories, any task with an estimate of more than eight weeks should be further divided into smaller tasks.

The total effort is traditionally measured in person-months, perhaps in homage to Brooks's classic software engineering book *The Mythical Man-Month* (Brooks 1995). Managers use salaries and overhead rates to convert person-months into an actual budget.

The cost estimate is likely done twice: once to bid a contract, and once again after the contract is won. The second estimate is done after the software architecture is designed, so that the tasks as well as the effort per task can be more easily and accurately identified.

The project manager surely wants the second estimate to be no larger than the first, since that is what the customer will pay. One suggestion is to add a safety margin by multiplying your original estimate by 1.3 to 1.5 to try to handle estimation inaccuracy or unforeseen events. Another is to make three estimates: a best case, expected case, and worst case, and then use that information to make your best guess.

The two approaches to estimating are experiential or quantitative. The first assumes the project managers have significant experience either at the company or in the industry, and they rely on that experience to make accurate estimates. It certainly increases confidence when the project is similar to tasks that the organization has already successfully completed.

The quantitative or algorithmic approach is to estimate the programming effort of the tasks in a technical measure such as lines of code (LOC), and then divide by a productivity measure like LOC per person-month to yield person-months per task. The project manager can get help from others to get estimates on LOC, and like velocity, can look at the historical record of the organization's productivity to calculate person-months.

Since cost estimates for software projects have such a dismal record, there has been considerable effort on improving the quantitative approach by collecting information about completed projects and finding models that predict the outcomes (Boehm and Valerdi 2008). The next step in sophistication follows this formula:

$$\text{Effort} = \text{Organizational Factors} \times \text{Code Size}^{\text{Size Penalty}} \times \text{Product Factors} \quad (7.1)$$

Constructive Cost Model (COCOMO) is the basis of this 1981 formula. Its 1995 successor is called COCOMO II.

where Organizational Factors include practices for this type of product, Code Size is measured as before, Size Penalty reflects that effort is not linear in code size, and Product Factors include experience of development team with this type of product, dependability requirements, platform difficulty, and so on. Example constants from real projects are 2.94 for

Organizational Factors; Size Penalty between 1.10 and 1.24; and Product Factors between 0.9 and 1.4.

While these estimates are quantitative, they certainly depend on the project manager's subjective picks for Code Size, Size Penalty, and Product Factors.

The successor to the COCOMO formula above asks the project manager to pick many more parameters. COCOMO II adds three more formulas to adjust estimates for 1) developing prototypes, 2) accounting for the amount of code reuse, and 3) a post-detailed-architecture estimate. This last formula expands Size Penalty by adding a normalized product of 5 independent factors and replaces Product Factors by a product of 17 independent factors.

The British Computer Society Survey of more than 1000 projects mentioned in Chapter 1 found that 92% of project managers made their estimates using experience instead of formulas (Taylor 2000).

As no more than 20% to 30% of projects meet their budget and schedule, what happens to the rest? Another 20% to 30% of the projects are indeed cancelled or abandoned, but the remaining 40% to 50% are still valuable to the customer even if late. Customers and providers typically then negotiate a new contract to deliver the product with a more limited set of features by a near-term date.

■ *Elaboration: Function points*

Function points are an alternative measure to LOC that can lead to estimates that are more accurate. They are based on the function inputs, outputs, external queries, input files, output files, and the complexity of each. The corresponding productivity measure is then function points per person-month.

4. Scheduling and Monitoring Progress. Given the SRS has been broken into tasks whose effort has been estimated, the next step is to use a scheduling tool that shows which tasks can be performed in parallel and which have dependencies so they must be performed sequentially. The format is typically a box and arrow diagram such as a **PERT chart**, which can identify the **critical path** or minimum time for project. For example, in Figure 7.7, the shortest possible path from step 1 (the starting state) to step 11 (software release) *must* traverse the nodes 3, 5, 9, and 10. The project manager places the graph in a table with rows associated with the people on the project, and then assigns people to tasks.

Once again, this process is typically done twice, once when bidding the contract, and once after the contract is won and the detailed architecture design is complete. Safety margins are again used to ensure that the first schedule, which is when the customer expects the product to be released, is not longer than the second version.

Similar to calculating velocity, the project manager can see if the project is behind by comparing the predicted expenditures and time for tasks to the actual expenditures and progress to date. A way to make project status clear to all stakeholders is to add intermediate milestones to the schedule, which lets everyone see if the project is on schedule and on budget.

5. Change Management for Requirements, Cost, and Schedule. As stated many times in this book, customers are likely to ask for changes to the requirements as the project evolves for many reasons, including a better understanding of what is wanted after trying a prototype, changing market conditions for the project, and so on. The challenge for the project manager is keeping the requirements documents, the schedule, and cost predictions up-to-date as the project changes. Thus, version control systems are needed for evolving documents as well as for programs, so the norm should be checking in the revised documentation along with the

PERT stands for Program Evaluation and Review Technique, which was invented by the US Navy in the 1950s for its nuclear submarine program.

Requirements Creep is the term developers use to describe the dreaded increase in requirements over time.

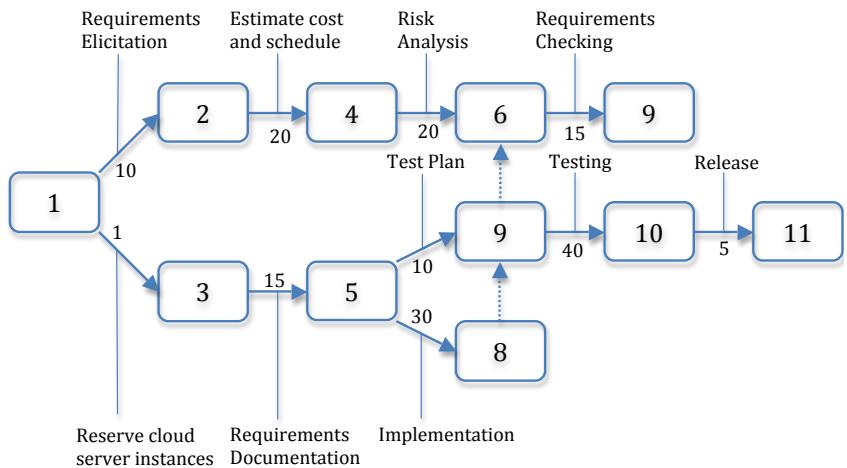


Figure 7.7: Numbered nodes represent milestones and labeled lines represent tasks, with arrowheads representing dependencies. Diverging lines from a node represent concurrent tasks. The numbers on the other side of lines represent the time allocated for the task. Dotted lines indicate dependencies that need no resources, so they have no time allocated for the task.

revised code.

6. Ensuring Implementation Matches Requirement Features. The Agile process consolidates these many major tasks into three tightly coupled ones: User Stories, acceptance tests in Cucumber, and the code that comes from the BDD/TDD process. Thus, there is little confusion in the relationship between particular stories, tests, and code.

However, plan-and-document methodologies involve many more mechanisms without tight integration. Thus, we need tools that allow the project manager to check to see if the implementation matches the requirements. The relationship between features in requirements and what is implemented is called **requirements traceability**. Tools that implement traceability essentially offer cross-references between a portion of the design, the portion of the code that implements the feature, code reviews that checked it, and the tests that validate it.

If there is both a high-level SRS and a detailed SRS, *forward traceability* refers to the traditional path from requirements to implementation, while *backwards traceability* is the mapping from a detailed requirement back to a high-level requirement.

7. Risk Analysis and Management. In an effort to improve the accuracy of cost estimation and scheduling, plan-and-document methodologies have borrowed risk analysis from the business school. The philosophy is that by taking the time up front to identify potential risks to the budget and schedule, a project can either do extra work to reduce the risk of changes, or change the plan to avoid risks. Ideally, risk identification and management occurs over the first third of a project. It does not bode well if they are identified late in the development cycle.

Risks are classified as technical, organizational, or business. An example of a technical risk might be that the relational database chosen cannot scale to the workload the project needs. An organizational risk might be that many members of the team are unfamiliar with J2EE, which the project depends upon. A business risk could be that by the time the project is complete, the product is not competitive in the market. Examples of actions to overcome these risks would be to acquire a more scalable database, send team members to a J2EE workshop, and do a competitive survey of existing products, including their current features and plans for improvements.

The approach to identify risks is to ask everyone for their worst-case scenarios. The project manager puts them into a “risk table,” in which each risk is assigned a probability of happening between 0% and 100%, and an impact on a numeric scale of 1 to 4, representing negligible, marginal, critical, and catastrophic. One can then sort the risk table by the product of the probability and impact of each risk.

There are many more potential risks than projects can afford to address, so the advice is to address the top 20% of the risks, in the hope that they represent 80% of the potential risks to the budget and schedule. Trying to address all potential risks could lead to an effort that is larger than the original software project! Risk reduction is a major reason for iteration in both the Spiral and RUP models. Iterations and prototypes should reduce risks associated with a project.

Section 7.5 mentions asking the customers about risks for the project as part of the cost estimation in Agile, but the difference is that this information is used to decide the range of the cost estimate rather than becoming a significant part of the project itself.

Tasks	In Plan-and-Document	In Agile
Requirements Documentation	Software Requirements Specification such as IEEE Standard 830-1998	User stories, Cucumber, Points, Velocity
Requirements Elicitation	Interviews, Scenarios, Use Cases	
Change Management for Requirements, Schedule, and Budget	Version Control for Documentation and Code	
Ensuring Requirements Features	Traceability to link features to tests, reviews, and code	
Scheduling and Monitoring	Early in project, contracted delivery date based on cost estimation, using PERT charts. Milestones to monitor progress	Evaluate to pick range of effort for time and materials contract
Cost Estimation	Early in project, contracted cost based on manager experience or estimates of task size combined with productivity metrics	
Risk Management	Early in project, identify risks to budget and schedule, and take actions to overcome or avoid them	

Figure 7.8: The relationship between the requirements related tasks of Plan-and-Document versus Agile methodologies.

Summary The hope of the original efforts in software engineering was to make software development as predictable in quality, cost, and schedule as building a bridge. Perhaps because less than a sixth of software projects are completed on time and on budget with full functionality, the plan-and-document process has many steps to try to achieve this difficult goal. Agile does not try to predict cost and schedule at the start of the project, instead relying on working with customers on frequent iterations and agreeing on a range of time for the best effort to achieve the customer's goals. Rating user stories on difficulty and recording the points actually completed per iteration increases the chances of more realistic estimates. Figure 7.8 shows the resulting different tasks given the differing perspectives of these two philosophies.

Self-Check 7.9.1. Name three plan-and-document techniques that help with requirements elicitation.

- ◊ Interviewing, Scenarios, and Use Cases. ■

7.10 Fallacies and Pitfalls



Pitfall: Customers who confuse mock-ups with completed features.

As a developer, this pitfall may seem ridiculous to you. But nontechnical customers sometimes have difficulty distinguishing a highly polished digital mock-up from a working feature! The solution is simple: use paper-and-pencil techniques such as hand-drawn sketches and storyboards to reach agreement with the customer—there can be no doubt that such Lo-Fi mockups represent *proposed* rather than implemented functionality.



Pitfall: Adding cool features that do not make the product more successful.

Agile development was inspired in part by the frustration of software developers building what they thought was cool code that customers dropped. The temptation is strong to add a feature that you think would be great, but it can also be disappointing when your work is discarded. User stories help all stakeholders prioritize development and reduce chances of wasted effort on features that only developers love.



Pitfall: Sketches without storyboards.

Sketches are static; interactions with a SaaS app occur as a sequence of actions over time. You and the customer must agree not only on the general content of the Lo-Fi UI sketches, but on what happens when they interact with the page. “Animating” the Lo-Fi sketches—“OK, you clicked on that button, here’s what you see; is that what you expected?”—goes a long way towards ironing out misunderstandings *before* the stories are turned into tests and code.



Pitfall: Tracking tasks rather than stories.

A story is the customer’s view of how a feature should work, such as “Box office manager can generate a report of today’s sales.” Tasks such as “Add Excel export code in Report model” is a developer-facing task that, while it may be part of implementing a story, is not itself something that results in customer value. Use project management and effort estimation tools to track stories, not tasks. Tracker allows multiple specific tasks to be part of a story, but expressing the task itself as a story also entails the further risk that you’ll use the tool as a to-do list, simply checking off tasks when they’re done, rather than tracking the lifecycle of a story and allowing you to improve your skill at estimating project effort.



Pitfall: Using Cucumber solely as a test-automation tool rather than as a common middle ground for all stakeholders.

If you look at `web_steps.rb`, you’ll quickly notice that low-level, imperative Cucumber steps such as “When I press Cancel” are merely a thin wrapper around Capybara’s “headless browser” API, and you might wonder (as some of the authors’ students have) why you should use Cucumber at all. But Cucumber’s real value is in creating documentation that nontechnical stakeholders and developers can agree on *and* that serves as the basis for automating acceptance and integration tests, which is why the Cucumber features and steps for a mature app should evolve towards a “mini-language” appropriate for that app. For example, an app for scheduling vacations for hospital nurses would have scenarios that make heavy use of domain-specific terms such as *shift*, *seniority*, *holiday*, *overtime*, and so on, rather than focusing on the low-level interactions between the user and each view.



Pitfall: Relying too heavily on integration-level scenarios for your tests.

Scenarios are comforting to write and satisfying to run (when they pass) because they closely mimic what a real user would do. Indeed, that is why Cucumber tests have value both as validation—you built the right thing, because the test instantiates a user story created in collaboration with the customer—and verification—you built the thing right, because the test passes. However, one thing such tests *don’t* reveal is whether your code is well factored—whether the different subsystems exercised in the scenario are easily testable, let alone whether each has been thoroughly tested. Unit and module level tests, which are the subject of Chapter 8, are more likely to tell you about the design of your code. Of course, over-reliance on unit and module level tests is just as bad, as the corresponding Pitfall at the

end of Chapter 8 reminds us!



Fallacy: The feature is “done” if it works correctly in production, even if the scenario doesn’t pass.

When a test fails, it’s trying to tell you something. Sometimes it’s straightforward—there’s a bug in your code. Other times it’s more subtle: your code fulfills the requirements of the user story, but for some reason, it is unusually difficult to test. Either way, the outcome bears investigation. Without an integration test you can trust, it will be hard to detect if future changes cause your existing code to break.



Pitfall: Trying to predict what you need before you need it.

Part of the magic of Behavior-Driven Design (and Test-Driven Development in the next chapter) is that you write the tests *before* you write the code you need, and then you write code needed to pass the tests. This top-down approach again makes it more likely for your efforts to be useful, which is harder to do when you’re predicting what you think you’ll need. This observation has also been called the YAGNI principle—You Ain’t Gonna Need It.



Pitfall: Careless use of negative expectations.

Beware of overusing *Then I should not see....* Because it tests a negative condition, you might not be able to tell if the output is what you intended—you can only tell what the output *isn’t*. Many, many outputs don’t match, so that is not likely to be a good test. For example, if you were testing for the *absence* of “Welcome, Dave!” but you accidentally wrote *Then I should not see* “Greetings, Dave!”, the scenario will pass even if the app incorrectly emits “Welcome, Dave!”. Always include positive expectations such as *Then I should see...* to check results.



Pitfall: Careless use of positive expectations.

Even if you use positive expectations such as *Then I should see...*, what if the string you’re looking for occurs multiple times on the page? For example, if the logged-in user’s name is Emma and your scenario is checking whether Jane Austen’s book *Emma* was correctly added to the shopping cart, a scenario step *Then I should see “Emma”* might pass even if the cart isn’t working. To avoid this pitfall, use Capybara’s `within` helper, which constrains the scope of matchers such as *I should see* to the element(s) matching a given CSS selector, as in `Then I should see 'Emma' within 'div#shopping_cart'`, and use unambiguous HTML `id` or `class` attributes for page elements you want to name in your scenarios. The Capybara documentation⁸ lists all the matchers and helpers.



Pitfall: Delivering a story as “done” when only the happy path is tested.

As should be clear by now, a story is only a candidate for delivery when both the happy path and the most important sad paths have been tested. Of course, as Chapter 8 describes, there are many more ways for something to work incorrectly than to work correctly, and sad-path tests are not intended to be a substitute for finer-grained test coverage. But from the user’s point of view, correct app behavior when the user accidentally does the wrong thing is just as important as correct behavior when they do the right thing.

7.11 Concluding Remarks: Pros and Cons of BDD

In software, we rarely have meaningful requirements. Even if we do, the only measure of success that matters is whether our solution solves the customer's shifting idea of what their problem is.

—Jeff Atwood, *Is Software Development Like Manufacturing?*, 2006

The advantage of user stories and BDD is creating a common language shared by all stakeholders, especially the nontechnical customers. BDD is perfect for projects where the requirements are poorly understood or rapidly changing, which is often the case. User stories also make it easy to break projects into small increments or iterations, which makes it easier to estimate how much work remains. The use of 3x5 cards and paper mockups of user interfaces keeps the nontechnical customers involved in the design and prioritization of features, which increases the chances of the software meeting the customer's needs. Iterations drive the refinement of this software development process. Moreover, BDD and Cucumber naturally leads to writing tests *before* coding, shifting the validation and development effort from debugging to testing.

Comparing user stories, Cucumber, points, and velocity to the plan-and-document processes makes it clear that BDD plays many important roles in the Agile process:

1. Requirements elicitation
2. Requirements documentation
3. Acceptance tests
4. Traceability between features and implementation
5. Scheduling and monitoring of project progress

Google places these posters inside restrooms to remind developers of the importance of testing. Used with permission.



The downside of user stories and BDD is that it may be difficult or too expensive to have continuous contact with the customer throughout the development process, as some customers may not want to participate. This approach may also not scale to very large software development projects or to safety critical applications. Perhaps plan-and-document is a better match in both situations.

Another potential downside of BDD is that the project could satisfy customers but not result in a good software architecture, which is an important foundation for maintaining the code. Chapter 11 discusses design patterns, which should be part of your software development toolkit. Recognizing which pattern matches the circumstances and refactoring code when necessary (see Chapter 9) reduces the chances of BDD producing poor software architectures.

All this being said, there is enormous momentum in the Ruby community (which places high value on testable, beautiful and self-documenting code) to document and promote best practices for specifying behavior both as a way to document the intent of the app's developers and to provide executable acceptance tests. The Cucumber wiki⁹ is a good place to start.

BDD may not seem initially the natural way to develop software; the strong temptation is to just start hacking code. However, once you have learned BDD and had success at it, for most developers there is no going back. Your authors remind you that good tools, while

sometimes intimidating to learn, repay the effort many times over in the long run. Whenever possible in the future, we believe you'll follow the BDD path to writing beautiful code.

You may find the following resources useful for more depth on the topics in this chapter:

- Want to see paper prototyping and storyboards in action? First read this excellent article with examples¹⁰ of paper prototyping, then watch this video of paper storyboarding¹¹ for a web-based email app.
- The Cucumber wiki¹² has links to documentation, tutorials, examples, screencasts, best practices, and lots more on Cucumber.
- *The Cucumber Book* (Wynne and Hellesøy 2012), co-authored by the tool's creator and one of its earliest adopters, includes detailed information and examples using Cucumber, excellent discussions of best practices for BDD, and additional Cucumber uses such as testing RESTful service automation.

B. W. Boehm and R. Valerdi. Achievements and challenges in COCOMO-based software resource estimation. *IEEE Software*, 25(5):74–83, Sept 2008.

F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading, MA, Anniversary edition, 1995. ISBN 0201835959.

D. Burkes. Personal communication, December 2012.

J. Johnson. The CHAOS report. Technical report, The Standish Group, Boston, Massachusetts, 1995. URL <http://blog.standishgroup.com/>.

J. Johnson. The CHAOS report. Technical report, The Standish Group, Boston, Massachusetts, 2009. URL <http://blog.standishgroup.com/>.

A. Taylor. IT projects sink or swim. *BCS Review*, Jan. 2000. URL <http://archive.bcs.org/bulletin/jan00/article1.htm>.

M. Wynne and A. Hellesøy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012. ISBN 1934356808.

Notes

¹<https://pivotaltracker.com>

²<http://www.youtube.com/watch?v=mTYcHg51sWY>

³<https://github.com>

⁴<https://trello.com>

⁵<http://railscasts.com/episodes/159-more-on-cucumber>

⁶<http://cukes.info>

⁷<http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/CaseStudies/MHCPMS/SupportingDocs/MHCPMSCaseStudy.pdf>

⁸<http://rubydoc.info/github/jnicklas/capybara/>

⁹<http://cukes.info>

¹⁰<https://alistapart.com/article/paperprototyping/>

¹¹<http://www.youtube.com/watch?v=GrV2SZuRPv0>

¹²<http://cukes.info>

8

Software Testing: Test-Driven Development

Sir Maurice Wilkes
(1913–2010) received the 1967 Turing Award for designing and building EDSAC in 1949, one of the first stored-program computers.



It was on one of my journeys between the EDSAC room and the punching equipment that “hesitating at the angles of stairs” the realization came over me with full force that a good part of the remainder of my life was going to be spent finding errors in my own programs.

—Maurice Wilkes, Memoirs of a Computer Pioneer, 1985

8.1	FIRST, TDD, and Red–Green–Refactor	238
8.2	Anatomy of a Test Case: Arrange, Act, Assert	240
8.3	Isolating Code: Doubles and Seams	243
8.4	Stubbing the Internet	248
8.5	CHIPS: Intro to RSpec on Rails	249
8.6	Fixtures and Factories	249
8.7	Coverage Concepts and Types of Tests	255
8.8	Other Testing Approaches and Terminology	258
8.9	CHIPS: The Acceptance Test/Unit Test Cycle	260
8.10	The Plan-And-Document Perspective on Testing	260
8.11	Fallacies and Pitfalls	264
8.12	Concluding Remarks: TDD vs. Conventional Debugging	266

Prerequisites and Concepts

The big concepts of this chapter are test creation, test coverage, and levels of testing.

Concepts:

Agile and Plan-and-Document differ sharply in their approaches to testing: when test writing starts, the order in which different kinds of tests are written, and even who does the testing.

Testing in the Agile lifecycle, which follows ***Test-Driven Development (TDD)***, is largely the responsibility of the developers rather than a separate Quality Assurance team. Agile testing follows these steps:

- Starting from the acceptance and integration tests derived from ***User stories***, write failing ***unit tests*** that test the nonexistent code you wish you had.
- Write just enough code to pass one such unit test and look for opportunities to ***refactor*** the code before continuing with the next test. Since many test frameworks display failing test output in red and passing test output in green, the iterative sequence of this step and the previous one is called ***Red–Green–Refactor***.
- To isolate the behavior of the code you're testing from the behavior of other classes or methods on which it depends, use ***test doubles***: "stunt doubles" that stand in for real objects in tests, but whose behavior you can closely control. Test doubles are examples of ***seams***, or places where you can change program behavior during testing without changing the source code itself.
- Construct your tests so that they are Fast, Independent, Repeatable, Self-checking, and Timely (FIRST).
- Capture and inspect ***code coverage*** metrics to help determine which parts of your code need more testing.

For the Plan-and-Document lifecycle, you use some of the same concepts in a quite different order and even with different people:

- The program manager assigns programming tasks based on the SRS, so ***unit testing*** starts after coding. ***Quality-Assurance (QA)*** testers take over from the developers to perform the higher level tests.
- ***Top-down***, ***Bottom-up***, and ***Sandwich*** are options on how to combine the resulting code to perform ***integration testing***. The testing plan and results are documented, such as by following IEEE Standard 829-2008.
- After integration testing, the QA team performs a ***system test*** before releasing it to the customer. Testing stops when a specified level of coverage is reached, such as "95% statement coverage."
- An alternative to testing, used for small critical software, is ***formal methods***. They use formal specifications of correct program behavior that are automatically verified by theorem provers or by exhaustive state search, both of which can go beyond what conventional testing can do.

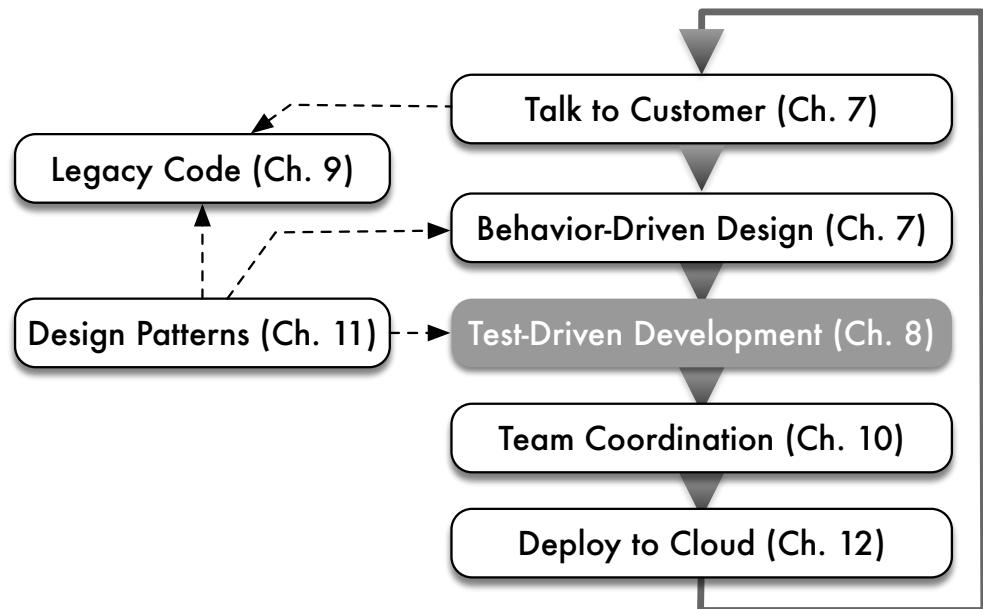


Figure 8.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter emphasizes unit testing as part of Test-Driven Development.

8.1 FIRST, TDD, and Red–Green–Refactor

Chapter 1 introduced the Agile lifecycle and distinguished two aspects of software assurance: validation (“Did you build the right thing?”) and verification (“Did you build the thing right?”). In this chapter, we focus on verification—building the thing right—via software testing as part of the Agile lifecycle. Figure 8.1 highlights the portion of the Agile lifecycle covered in this chapter.

Although testing is only one technique used for verification, we focus on it because its role is often misunderstood, and as a result it doesn’t get as much attention as other parts of the software lifecycle. In addition, as we will see, approaching software construction from a test-centric perspective often improves the software’s readability and maintainability. In other words, *testable code tends to be clear code, and vice versa*. This insight may take a while to sink in if you are new to TDD, because practicing TDD may feel alien to you. We ask you again to be patient and have faith in the process!

In Agile development, developers do not “toss their code over the wall” to the **Quality Assurance (QA)** team, nor do QA engineers extensively exercise the software manually and file bug reports. Instead, Agile developers bear far more responsibility for testing their own code and participating in reviews, while Agile QA responsibilities focus on improving the testing tools infrastructure, helping developers make their code more testable, and verifying that customer-reported bugs are reproducible, as we’ll discuss further in Chapter 10. Furthermore, in the vast majority of tests you will write, the test code itself can determine whether the code being tested works or not, without requiring a human to manually check test output or interact with the software.

Even though Agile developers are expected to write their own tests, and those tests are

expected to be automated, there is often a role for some manual testing. For example, **user acceptance testing** observes actual users (or QA engineers acting as “typical” users) using the product to determine whether you “built the right thing,” and operational acceptance testing may manually try additional scenarios to ensure you “built the thing right.” Both can uncover bugs that were previously undetected, some of which can then have automated tests created for them. And some visual aspects of the design, such as whether particular elements on the page render in a visually appealing way, require manual inspection. But in general, modern software quality assurance is the shared responsibility of a whole team following good processes, rather than compartmentalized in a separate group.

In this section we introduce two key ideas that underpin **Test-driven development** (TDD): Red–Green–Refactor and making tests FIRST. TDD advocates the use of tests to *drive* the development of code. When TDD is used to create new code, as in this chapter, it is sometimes referred to as *test-first development*. The basic TDD workflow, repeated for each created test, is known as Red–Green–Refactor and proceeds as follows.

When TDD is used to extend or modify legacy code, as in Chapter 9, new tests may be created for code that already exists.

1. Before you write any code, write a test for *one* aspect of the behavior you *expect* the new code will have. Since the code being tested doesn’t exist yet, writing the test forces you to think about how you *wish* the code would behave and interact with its collaborators if it did exist. We call this “exercising the code you wish you had.”
2. **Red** step: Run the test, and verify that it fails because you haven’t yet implemented the code necessary to make it pass (that is, the code you wish you had).
3. **Green** step: Write the *simplest possible* code that causes *this* test to pass without breaking any existing tests.
4. **Refactor** step: Look for opportunities to **refactor** either your code or your tests—changing the code’s structure to eliminate redundancy or repetition that may have arisen as a result of adding the new code. The tests ensure that your refactoring doesn’t introduce bugs.

How do you know when you have completed all necessary tests? If you are using BDD (Chapter 7) to drive your application development, the new code being written is presumably necessary to make one or more Cucumber scenario steps pass. When all steps in a scenario pass, you’re done.

Although TDD may feel strange at first, it tends to result in code that is not only well tested, but also more modular and easier to read than code developed separately from tests. While TDD is certainly not the only way to achieve those goals, it is difficult to end up with seriously deficient code if TDD is used correctly.

What about the tests themselves? Five principles for creating good tests are summarized by the acronym FIRST: **F**ast, **I**ndependent, **R**epeatable, **S**elf-checking, and **T**imely.

- **Fast:** it should be easy and quick to run the subset of test cases relevant to your current coding task, to avoid interfering with your train of thought.
- **Independent:** The order in which tests run shouldn’t matter. More precisely, if no test relies on preconditions created by other tests, we can prioritize running only a subset of tests that cover recent code changes.

Y2K bug in action This photo was taken on Jan. 3, 2000. (Wikimedia Commons)



- Repeatable: test behavior should not depend on external factors such as today's date or on "magic constants" that will break the tests if their values change, as occurred with many 1960s programs when the year 2000 arrived due to the **Y2K problem**.
- Self-checking: each test should be able to determine on its own whether it passed or failed, rather than relying on humans to check its output.
- Timely: tests should be created or updated at the same time as the code being tested. As we'll see, with test-driven development the tests are written *immediately before* the code.

Summary

- Besides ensuring software correctness, another reason to use test-centric software construction is that it often improves readability and maintainability: *testable code tends to be clear code, and vice versa*.
- Software QA is a shared responsibility of the whole team: developers write most of their own tests, with QA staff improving the testing environment and handling some kinds of testing that are hard to automate.
- The TDD cycle of Red–Green–Refactor begins with writing a test that fails because the *subject code* being tested doesn't exist yet (Red), then adding the minimum code necessary to pass just that one example (Green), and finally DRYing out and cleaning up the test code (Refactor).
- Tests should be **Fast** to run, with results **Independent** of the order in which they are run, thus **Repeatably** giving the same result. Each test should be **Self-checking** (the test code itself knows whether the test passed or failed), and should be developed in a **Timely** way with respect to the code it tests. Indeed, TDD suggests developing the tests *before* writing the code.

Self-Check 8.1.1. Suppose step 1 in your Cucumber scenario is passing, but step 2 is failing because the code needed is not yet written. If you are practicing strict BDD and TDD, explain why you will necessarily go through one or more cycles of Red–Green–Refactor before step 2 passes.

◊ If the code for step 2 does not yet exist, strict TDD says you should develop that code by *first* writing a focused test for one aspect of the code's behavior, watching that test fail, *then* writing the code to make it pass. ■

8.2 Anatomy of a Test Case: Arrange, Act, Assert

The ISTQB (International Software Testing Qualifications Board) defines² *test object* as the thing being tested and SUT as a test object that is a system, but the xUnit terminology is more widely used among Agile developers.

We begin with a few definitions. Following the terminology in the fairly widely used xUnit Test Patterns¹ (Meszaros 2007), we refer to the object being tested as the **system under test** (SUT), whether that "object" is a single method, a group of methods, or even the entire application. That is, SUT is defined from the point of view of the test. The goal of a single **test case** for some SUT is to check that some specific behavior happens (for example, the return value from a function matches an expected result) or doesn't happen (for example, passing

On a value	Example in RSpec
Value equality	<code>expect(x).to eq('Ruby');</code>
Boolean	<code>expect(x).to beTruthy # i.e., non-false/non-nil</code>
Regular expression	<code>expect(s).to match(/YourRegexpHere/)</code>
Object properties	<code>expect(a).to beAKindOf(Array)</code> <code>expect(a).to respondTo(:[])</code>
On a block	Example in RSpec
Exception	<code>expect { Math.sqrt(-1) }.to raiseError(Math::DomainError)</code>
Side effect	<code>expect { Review.first.destroy }.to change { Review.count }.by(-1)</code>

Figure 8.2: A few examples of the kinds of assertions allowed by RSpec. One can invert the sense of any assertion (“Expect `x` not to equal 50”), as in the first line of the table, but as Chapter 7 warned, negative assertions should be used with caution, since there are many ways for a program *not* to satisfy a particular condition while still not behaving correctly. The use of braces rather than parentheses for the Exception example shows that `expect` can take either an expression, like `x`, or a callable block.

an empty string to a string comparison function doesn’t result in an error or exception). A collection of test cases is called a **test suite**. A code base usually has several test suites, corresponding to different kinds of tests, as we describe later in Section 8.7.

In this section we focus on **unit tests**, the finest-grained test cases, for which the SUT is a single method. In particular, if the method being tested does not call any other methods to help do its job, we say it is a **leaf method**. Since even a leaf method may have multiple testable behaviors, a single method may be the subject of multiple test cases.

A unit test is conceptually simple: call a method, and verify some aspect of its behavior. But even leaf methods usually require establishing some preconditions before exercising the code. For example, when testing a method that combines two lists into a single sorted list, we need to create the two lists that will be provided as input. We then exercise the SUT, and finally check whether the particular behavior we were looking for was correctly exhibited.

In general, then, every test case in a suite follows the same structure of *Arrange, Act, Assert*:

1. Arrange: create any necessary preconditions for the test case, such as setting values of variables that affect the behavior of the SUT.
2. Act: exercise the SUT.
3. Assert: verify that the result or behavior matches what was expected.

The general form of an **assertion** or expectation is “Expect `expression` to satisfy `predicate`”. An example of a simple predicate is an equality check: “Expect the return value of `Math.sqrt(49)` to equal 7”. As Figure 8.2 shows, other kinds of assertions deal with both inspecting output values and checking non-output-value-related behaviors.

The easiest unit tests to write are those for which the SUT is a method that is a **pure function**—one that has no side effects and whose return value is always the same for the same arguments. The only thing a test case needs to do is choose some inputs, call the method, and check the returned value. For example, consider a hypothetical method `leap?` that accepts an integer and returns a truthy value if and only if that integer corresponds to a leap year (that is, it is either a multiple of 400, or a multiple of 4 but not 100). So, for example, 2000 and 2004 are leap years, but 1900 is not. Since exhaustive testing (trying every possible input) is clearly infeasible, how do we choose which input values to use for our test cases?

Category	Test value
1. A number that is not a multiple of 4 or 100 (and therefore not a multiple of 400)	1973
2. A number that is a multiple of 4, but not of 100 (and therefore not a multiple of 400)	2008
3. A number that is not a multiple of 400, but is a multiple of 100	1900
4. A number that is a multiple of 400	2000

Figure 8.3: A set of categories that completely covers the space of possible nonnegative numeric inputs for a leap year detector, and a possible test value representative of each category. The calculation of a leap year depends only on which category a value is in, not the value itself.

A good guideline in such cases is to use input values that would cause the calculation performed in the method to follow different code paths. Given the above rule for leap years, by inspection we can deduce four categories, as Figure 8.3 shows.

Chapter 9 considers how to write tests after the fact for code you didn't write or can't easily inspect.

A “pure” TDD workflow for developing a function that detects leap years might therefore proceed as follows. Choose any value in category 1 above, and write a test case that asserts that the return value of `leap?` is falsy when called with that value. The test fails because `leap?` doesn’t yet exist, so write just enough of `leap?` to make that case pass. Next, choose a value in category 2, write the corresponding test, and when it fails, modify `leap?` so that now both tests pass. Continue until all categories are covered.

You might object that `leap?` is such a simple leaf method, and its functionality so well-circumscribed, that you might as well write the entire method at once along with the four test cases, rather than going through the motions of developing each test case incrementally. That’s not an unreasonable objection, and as with other Agile practices, “pure” TDD is an ideal to strive for even if you do not always follow it to the letter. But with methods that have more complex behaviors, TDD is a valuable way to proceed methodically.

Summary:

- The system under test (SUT), which may be as small as a single method or as large as the whole app, is the subject of a particular test case. A test suite is a full set of tests, which usually includes unit, integration, and perhaps other types of tests.
- The finest-grained tests are unit tests, which test the code in a single method. The simplest unit tests are those for pure leaf functions: deterministic, no side effects, no collaborators or helper methods called. Hence, it is worth structuring your code to expose as much functionality as possible in pure leaf functions.
- One way to select input values for unit tests is to test critical points (values that may influence the code path in the SUT) and values from each noncritical set (set within which the choice of any particular value does not affect control flow).
- Each unit test checks just one behavior, so, for example, each category of input values would get its own test case. Most testing frameworks provide some way to group together examples that test related behaviors and share common setup or teardown phases.
- Each test case follows the same structure: arrange (set up preconditions), act (stimulate the SUT), assert (verify the expected results). The assertion step makes each test self-checking, eliminating the need for a human programmer to inspect test results.
- Common assertions are checks on values (equality, betweenness, and so on) and checks on behavior (is an exception raised or not).

8.3 Isolating Code: Doubles and Seams

We can distinguish three characteristics (which may occur individually or together) that complicate unit tests:

- The SUT has one or more dependencies, such as other methods it calls to help do its work. Test cases should isolate the SUT from those dependencies.
- The SUT has side effects when executed; that is, it causes a change in application state visible outside the test code itself. Test cases should verify that the correct side effect occurred, which involves inspecting application state outside the SUT.
- The SUT is not a pure function, because its output depends not only on its input but other implicit factors, such as the time of day or a random event. Test cases should control the values of these factors to force the SUT to traverse predictable code paths.

As an example, consider testing a controller action. By design, as we have seen, controller actions shouldn't contain “business logic”—instead they manage communication with the model, calling model methods to do the real work and setting up variables to display information in the view. To make our example relevant to SaaS, consider a hypothetical SaaS app that allows the user to look up a movie in another service’s movie database, and display the movie info so the user can write a review. Here is how our hypothetical app works:

```
https://gist.github.com/ae42df0d8365b35af8b67f2698d74c3c
1 class MoviesController < ApplicationController
2   def review_movie
3     search_string = params[:search]
4     begin
5       matches = Movie.find_in_tmdb(search_string)
6       if matches.empty? # nothing was found
7         redirect_to review_movie_path, :alert => "No matches."
8       elsif matches.length == 1
9         @movie = matches[0]
10        render 'review_movie'
11      else # more than 1 match
12        @movies = matches
13        render 'select_movie'
14      end
15    rescue Movie::ConnectionError => err
16      redirect_to review_movie_path, :alert => "Error contacting TMDb: #{err.message}"
17    end
18  end
19 end
```

Figure 8.4: A simple controller method that tries to search a remote database for one or more matches to a movie title. The call to the remote service happens from within the `find_in_tmdb` class method.

1. The `Movie` model has a class (static) method `find_in_tmdb` that makes a call to the API of the external service The Movie Database (TMDb)³ and returns an array of `Movie` objects, which may be empty if there were no matches.
2. If there are no matches, the controller action should redirect the user back to the search page with an appropriate message.
3. If there is exactly one match, the controller should render a view that allows the user to enter a review for that movie.
4. If there is more than one match, the controller should render a different view that allows the user to specify which movie they want to review.
5. Because the model method relies on calling an external service, the call might fail if the service doesn't respond for some reason. In that case, we assume `Movie.find_in_tmdb` will raise the exception `Movie::ConnectionError`.

Figure 8.4 shows what the above controller action might look like.

How would we unit-test this controller action? The Arrange step consists of preparing `params` to hold some search string. The Act step consists of calling the controller action with that search string. But the Assert step depends on whether the call to `find_in_tmdb` returns an empty array, an array of exactly one match, an array containing more than one match, or raises an exception because of an error communicating with The Movie Database. Indeed, as items 2–5 in the list above show, there are really four test cases required here, and to test each of them, we essentially need to be able to control the *behavior of the call* to `find_in_tmdb`.

Michael Feathers (Feathers 2004) defines a *seam* as “a place where you can alter behavior in your program without editing in that place.” In our case, we want to alter (control) the behavior of `find_in_tmdb` but without changing the source code of the controller action. Recall that one ability afforded by metaprogramming is being able to modify code while a

<https://gist.github.com/94f973152cfb2080ed5875ab1685acaf>

```

1 describe MoviesController do
2   describe 'looking up movie' do
3     it 'redirects to search page if no match' do
4       allow(Movie).to receive(:find_in_tmdb).and_return( [] )
5       post 'review_movie', {'search_string' => 'I Am Big Bird'}
6       expect(response).to redirect_to(review_movie_path)
7     end
8   end
9 end

```

Figure 8.5: This RSpec example (test case) stubs `Movie.find_in_tmdb` to isolate the controller action from its collaborators for the purposes of unit testing.

program is running. In this case, the strategy would be to *temporarily* modify `find_in_tmdb` so that *instead of calling the real method, it calls a “fake” method whose behavior we control* and can change for each test case.

Such a construction is called a **method stub**, and is easy to implement in languages that support metaprogramming. The RSpec testing framework provides direct support for this, as Figure 8.5 shows: the Arrange part of a test now includes setting up a stub for the method, and specifying that when the stub is called, it should return an empty array, ensuring that `matches.empty?` in line 6 of Figure 8.4 will be true, causing line 7 to be executed next. As is typical for a testing framework, RSpec “un-registers” any stubs after each example (test case) is run, making the stub visible only to that test case and thereby keeping tests Independent. Later we will show how to group together sets of examples that rely on the same precondition setup, so that tests can be DRY as well.

Keeping in mind that every Ruby function call is a method call on an object, line 4 of Figure 8.5 can be read as follows: “Allow the `Movie` class (which is itself an object) to receive a call to its (class) method `find_in_tmdb`, and return an empty array as the return value of that call.” Note that it is *not an error* for `find_in_tmdb` not to be called: the stub setup only specifies what should happen *if* that method is called. If we wanted to express the test condition that the method *must* be called, we would replace `allow` with `expect`. In that case, line 4 would be both an Arrange step defining a stub and an Assert step specifying that the test should fail if the stub isn’t actually called. RSpec automatically verifies `expect...to receive` assertions at the end of each example, so the test wouldn’t need an extra line to check if the stub was called—simply using `expect` rather than `allow` to set up the stub distinguishes the two cases.

In this case, `receive()` creates a seam by overriding a method in place, without us having to edit the file containing the original method (although in this case, the original method doesn’t even exist yet). Seams are also important when it comes to adding new code to your application, but in the rest of this chapter we will see many more examples of seams in testing. Seams are useful in testing because they let us break dependencies between a piece of code we want to test and its collaborators, allowing the collaborators to behave differently under test than they would in real life.

The kind of seam we just described is called a **method stub** or simply *stub*, because it is a piece of code that replaces the real method’s code with a controllable or fixed behavior for testing purposes. A **mock object** or simply *mock* is a simplified “stunt double” of an object that can only mimic a few fixed behaviors of the object, such as returning fixed values for specific attributes. Mocks are useful when a real object would be complex to instantiate because it has other dependencies, yet only a few specific properties of the object

Spies are similar to stubs, but they allow a call to the real method to proceed while “recording” the arguments and return values for later inspection.

System under test (SUT)	Testing strategy
Pure leaf function—no side effects, no collaborator methods or classes, same inputs yield same outputs	Assert correct output results for critical values and for arbitrary values in noncritical regions
Relies on results from calling collaborator methods or invoking behaviors on collaborator objects	In Arrange phase, create doubles that “force” the desired behavior by returning prearranged values, raising an exception, and so on
Nondeterministic or time-dependent behavior	In code under test, isolate the nondeterminism in a method call that can be stubbed using a double in the Arrange phase
Has side effects	In Arrange phase, observe the relevant state before executing test code; in Assert phase, observe it again and check for side effect

Figure 8.6: Strategy to properly isolate the SUT when it is not a pure function, not a leaf method, or both.

are necessary for the SUT to work properly. The term **test double** generically covers these and a few other types of seams. Figure 8.6 summarizes typical strategies for using these doubles in various unit-testing scenarios, and Figure 8.7 shows examples of each strategy using RSpec.

Summary

- When testing a method that has external dependencies, for example calling other methods or consuming other objects, we use test doubles to “stand in” for the real methods or objects and allow the test to tightly control the SUT’s behavior.
- Test doubles are set up in the Arrange phase of a test case. Stubs are set up to control the return values from collaborator methods, while mocks are set up to mimic just those behaviors of the collaborator object used by the SUT.
- Depending on what behavior is being tested, a test case can specify whether a particular stub *must* be called, that is, if the stub not being called signals a bug.

<https://gist.github.com/7ba2377b49261a1d97c3109b99a4dbf0>

```
1 # 1. Pure leaf function: test critical values and noncritical regions
2 it 'occurs when multiple of 4 but not 100' do
3   expect(leap?(2008)).to be_truthy
4 end
5 it 'does not occur when multiple of 400' do
6   expect(leap?(2000)).to be_falsy
7 end
8
9 # 2. Using doubles for explicit dependencies such as collaborators
10 #   UI.background() calls Defcon.level() to determine display color
11 it 'colors the UI red if Defcon is 2 or lower' do
12   # Arrange: stub Defcon to return 2
13   allow(Defcon).to receive(:level).and_return(2)
14   expect(UI.background).to eq('red')           # Act and Assert
15 end
16
17 # 3. Has implicit dependencies such as time
18 it 'runs backups on Tuesdays' do
19   # Arrange: stub Date.today to return Tues 2020-02-04
20   allow(Date).to receive(:today).and_return(Time.local(2020,2,4))
21   expect(run_backups_today?).to be_truthy      # Act and Assert
22 end
23
24 # 4. Has side effects (verbose version)
25 it 'lowers Defcon level by 1' do
26   # Arrange: check previous value of state
27   before = Defcon.level()
28   post_alert("Hostile craft detected")        # Act
29   expect(Defcon.level()).to eq(before - 1)     # Asset
30 end
31
32 # Shortcut version passing a callable to `expect`
33 it 'lowers Defcon level by 1' do
34   expect { post_alert("Hostile craft detected") }.
35     to change { Defcon.level() }.by(-1)
36 end
```

Figure 8.7: RSpec examples corresponding to Figure 8.6.

■ Elaboration: Seams in other languages

In non-object-oriented languages such as C, seams are hard to create. Since all method calls are resolved at link time, usually the developer creates a library containing the “fake” (test double) version of a desired method, and carefully controls library link order to ensure the test-double version is used. Similarly, since C data structures are accessed by reading directly from memory rather than calling accessor methods, data structure seams (mocks) are usually created by using preprocessor directives such as `#ifdef TESTING` to compile the code differently for testing vs. production use.

In statically-typed OO languages like Java, since method calls are resolved at runtime, one way to create seams is to create a subclass of the class under test and override certain methods when compiling against the test harness. Mocking objects is also possible, though the mock object must satisfy the compiler’s expectations for a fully-implemented “real” object, even if the mock is doing only a small part of the work that a real object would. The JMock website⁴ shows some examples of inserting testing seams in Java.

In dynamic OO languages like Ruby that let you modify classes at runtime, we can create a seam almost anywhere and anytime. RSpec exploits this ability in allowing us to create just the specific mocks and stubs needed by each test, which makes tests easy to write.

Self-Check 8.3.1. *Name two likely violations of FIRST that arise when unit tests actually call an external service as part of testing.*

- ◊ The test may no longer be Fast, since it takes much longer to call an external service than to compute locally. The test may no longer be Repeatable, since circumstances beyond our control could affect its outcome, such as the temporary unavailability of the external service.

8.4 Stubbing the Internet

When testing a method that makes a call to an external service via an API, there are many reasons we almost certainly *don’t* want to make a real call to that API. One reason is abuse of the service’s terms. Several years ago, the CS department head at a major US university received a complaint from a web site that hosted academic papers, because a group of students had been working on a student project that repeatedly made “test” API calls against the real site. The site threatened to cut off the university’s access if the students continued this behavior. Another reason is that making real calls might prevent the test from being **Repeatable** depending on how the remote service responds, and would almost certainly prevent the test from being **Fast**.

In fact, when testing our own app, all that we really care about is whether the API calls *it would* make are correctly formed—analogous to checking a call to a method stub to make sure the arguments are correct. So the more general question is: Where should we stub external methods when testing an app that makes calls to an external service? In Figure 8.5 we chose to stub the model and mimic the results of the gem’s calls to TMDb, but a more robust integration testing approach would instead place the stub “closer” to the remote service. In particular, we could create fixtures—files containing the JSON content returned by actual calls to the service—and arrange to intercept calls to the remote service and return the contents of those fixture files instead. The Webmock⁵ gem does exactly this: it stubs out the entire Web except for particular URIs that return a canned response when accessed from a

Ruby program. (You can think of Webmock as `allow(...).to receive(...).and_return` for the whole Web.) There's even a companion gem VCR⁶ that automates getting a response from the real service, saving the response data in a fixture file, and then "replaying" the fixture when your tests cause the remote service to be "called" by intercepting low-level calls in the Ruby HTTP library.

From an integration-testing standpoint, Webmock is the most realistic way to test interactions with a remote service, because the stubbed behavior is "farthest away"—we are stubbing as late as possible in the flow of the request. Therefore, when creating Cucumber scenarios to test external service integration, Webmock is usually the appropriate choice. From a unit testing point of view (as we've adopted in this chapter) it's less compelling, since we are concerned with the correct behavior of specific class methods, and we don't mind stubbing "close by" in order to observe those behaviors in a controlled environment.

VCR (for **Videocassette Recorder**) was an analog-tape video-recording device popular in the 1980s but made obsolete by DVDs in the early 2000s. The `vcr` gem even uses the term *cassette* to refer to the stored server responses that are replayed during tests.

Summary:

- To create **Fast** and **Repeatable** test cases for code that communicates with an external service, we use stubs to mimic the service's behavior. `context` blocks can group specs that test different behaviors of the remote service, using `before` blocks to set up necessary stubs or other preconditions to simulate each behavior.
- The question of "where to stub" an external service depends on the purpose of the test. Stubbing "far away" using Webmock is more realistic and appropriate for functional or integration tests; stubbing "close by" in a gem or library that communicates with the remote service is often adequate for low-level unit tests.

Self-Check 8.4.1. Is "stubbing the Internet" in conflict with the advice of Chapter 7 that one should avoid mocks or stubs in full-system Cucumber scenarios?

◊ Full-system testing should avoid "faking" certain parts of it as we have done using seams in most of this chapter. However, if the "full system" includes interacting with outside services we don't control, such as the interaction with TMDb in this example, we do need a way to "fake" their behavior for testing. ■

8.5 CHIPS: Intro to RSpec on Rails



CHIPS 8.5: Intro to RSpec on Rails

<https://github.com/saasbook/hw-tdd-rspec>

In this assignment, you'll learn to use RSpec and other tools to support test-driven development.

8.6 Fixtures and Factories

Doubles are appropriate when you need a stand-in with a small amount of functionality to isolate the code under test from its dependencies. But suppose you were testing a new instance method of class `Movie` called `name_with_rating` that returns a nicely formatted string showing a movie's title and rating. Clearly, such a method would have to access the

`title` and `rating` attributes of a `Movie` instance. You could create a double that knows all that information, and pass that double:

<https://gist.github.com/040ab61ab111ebfe71e9c42ff9dc726b>

```
1 fake_movie = double('Movie')
2 allow(fake_movie).to receive(:title).and_return('Casablanca')
3 allow(fake_movie).to receive(:rating).and_return('PG')
4 expect(fake_movie.name_with_rating).to eq 'Casablanca (PG)'
```

But since the instance method being tested is part of the `Movie` class itself, it makes sense to use a real object here, since this isn't a case of isolating the test code from collaborator classes.

Where can we get a real `Movie` instance to use in such a test? Most testing frameworks for object-oriented languages support the use of **factories**—bits of code or declarative descriptions of objects designed to allow rapid creation of full-featured objects (rather than mocks) at testing time. The goal of a factory is to quickly create valid instances of a class using some default attributes that you can selectively override for testing. For example, if you were testing some code that allows a user to write a review for a movie, you might need a valid movie instance to pass to that code. In the above scenario of testing a title-and-rating formatter, you don't care what the movie's release date is, or who directed it; you just need a movie object that is valid and whose title and rating you do know. So you would ask the factory to produce a movie instance whose title and rating you specify, but whose other attributes you don't care about as long as they are valid values.

You might think this seems like more work than just creating a movie instance directly by calling its constructor. In our simple example, that may be true. But there are two cases in which factories really shine. The first is when the object to be created has many attributes that must be initialized at creation time, even though any particular test case may only care about the specific values of a few of them. For example, the app that manipulates `Movie` objects may have validations requiring a movie to have a valid release date or other fields meeting specific criteria, yet the test above doesn't care about the values of those other fields. In such cases, you can ask the factory to create an object in which certain attribute values are specified but others are filled in with valid defaults. The second case is when objects you need to create have has-many or belongs-to relationships with other objects, as Chapter 5 describes. For example, if a `Review` belongs to a `Movie`, and you are creating a set of tests to check various behaviors of a `Review`, you literally cannot create a valid `Review` instance without creating a `Movie` instance for it to belong to, even if the tests you are writing don't care about the movie itself. In this case, the `Review` factory can be configured so that creating a `Review` also creates a valid `Movie` to which it belongs. Again, you can either specify a particular `Movie` object you've created, or let the factory create one with valid default values. Then in your test you can simply ask for a `Review` object to be created, without having the details of the parent relationship clutter your test code.

The Ruby gem FactoryBot⁷ lets you define a factory for any kind of model in your app and create just the objects you need quickly for each test, selectively overriding only certain attributes, as Figure 8.8 shows.

In database-backed MVC apps, one other source of “real” objects for use in tests is **fixtures**—a set of objects whose existence is guaranteed and fixed, and can be assumed by all test cases. The term *fixture* comes from the manufacturing world: a test fixture is a device that holds or supports the item under test. Since all state in Rails SaaS apps is kept in the database, a fixture file defines a set of objects that is automatically loaded into the test database before tests are run, so you can use those objects in your tests without first setting them up. Rails

Don't confuse this use of the term “factory” with the Abstract Factory Pattern discussed in Chapter 11.



<https://gist.github.com/81860a06858f0233566e76324228b185>

```

1 # spec/factories/movie.rb
2
3 FactoryBot.define do
4   factory :movie do
5     title 'A Fake Title' # default values
6     rating 'PG'
7     release_date { 10.years.ago }
8   end
9 end

```

<https://gist.github.com/b12c0c67a9859877324ec85dc7dee37a>

```

1 # in spec/models/movie_spec.rb
2 describe Movie do
3   it 'should include rating and year in full name' do
4     # 'build' creates but doesn't save object; 'create' also saves it
5     movie = FactoryBot.build(:movie, :title => 'Milk', :rating => 'R')
6     expect(movie.name_with_rating).to eq 'Milk (R)'
7   end
8 end

```

Figure 8.8: Using factories rather than fixtures preserves Independence among tests. Frameworks such as FactoryBot (gem ‘factory_bot_rails’ in Gemfile) make factory creation easy.

looks for fixtures in a file containing objects expressed in **YAML** (a recursive acronym for YAML Ain’t Markup Language), as Figure 8.9 shows. Following convention over configuration, the fixtures for the Movie model are loaded from `spec/fixtures/movies.yml`, and are available to your specs via their symbolic names, as Figure 8.9 shows.

But unless used carefully, fixtures can interfere with tests being Independent, as every test now depends implicitly on the fixture state, so changing the fixtures might change the behavior of tests. In addition, although any given test probably relies on only one or two fixtures, the union of fixtures required by all tests can become unwieldy. Therefore, fixtures should be used very sparingly if at all, and primarily for truly fixed data that, in production, would not be expected to change while the app is running but needs to be present in order for it to work. For example, at deployment time the app might allow setting the timezone or language in which it operates and storing the preferences in the database, and many aspects of the app might rely on these values being set to a legal value. Having a fixture that “hardwires” some values suitable for testing is reasonable in this case. As a rule of thumb, *use factories for kinds of data that normally change while the app is running, and consider fixtures for data that doesn’t change but must be present for the app to work at all*.

Whether you use factories or fixtures, the test framework itself (in our case, RSpec) is responsible for restoring the state of the world to look “pristine” before the next test case runs, just as with doubles. Specifically, the database is completely erased, and any fixtures are then reloaded. Doing this *test teardown* before every single example keeps tests Independent.



```
https://gist.github.com/ab3223cfdfc9270c8391ea63f5f66571
```

```

1 # spec/fixtures/movies.yml
2 milk_movie:
3   id: 1
4   title: Milk
5   rating: R
6   release_date: 2008-11-26
7
8 documentary_movie:
9   id: 2
10  title: Food, Inc.
11  release_date: 2008-09-07

```

```
https://gist.github.com/697fb4ec76a1f1845a72bf1d2341b972
```

```

1 # spec/models/movie_spec.rb:
2
3 require 'rails_helper'
4
5 describe Movie do
6   fixtures :movies
7   it 'includes rating and year in full name' do
8     movie = movies(:milk_movie)
9     expect(movie.name_with_rating).to eq('Milk (R)')
10    end
11  end

```

Figure 8.9: Fixtures declared in YAML files (top) are automatically loaded into the test database before each spec is executed (bottom). After each example runs, the database is cleared out and the fixtures reloaded.

Summary

- When a test needs to operate on a real object rather than a mock, the real object can be created on the fly by a factory or preloaded as a fixture. But beware that fixtures can create subtle interdependencies between tests, breaking Independence, so best practice is to avoid them except for fixed data that must be present for the app to run at all.
- Tests are a form of internal documentation. RSpec exploits Ruby language features to let you write exceptionally readable test code. Like application code, test code is there for humans, not for the computer, so taking the time to make your tests readable not only deepens your understanding of them but also documents your thoughts more effectively for those who will work with the code after you've moved on.

Self-Check 8.6.1. Suppose a test suite contains a test that adds a model object to a table and then expects to find a certain number of model objects in the table as a result. Explain how the use of fixtures may affect the Independence of the tests in this suite, and how the use of factories can remedy this problem.

◊ If the fixtures file is ever changed so that the number of items initially populating that table changes, this test may suddenly start failing because its assumptions about the initial state of the table no longer hold. In contrast, a factory can be used to quickly create only those objects needed for each test or example group on demand, so no test needs to depend on any global “initial state” of the database. ■

Structure of test cases:

- `before(:each) do...end`
Set up preconditions executed before each spec (use `before(:all)` to do just once, at your own risk)
- `it 'does something' do...end`
A single example (test case) for one behavior
- `describe 'collection of behaviors' do...end`
Groups a set of related examples

Mocks and stubs:

- `m=double('movie')`
Creates a mock object with no predefined methods
- `allow(m).to receive(:rating).and_return('R')`
Replaces the existing `rating` method on `m`, or defines a new `rating` method if none exists, that returns the canned response '`R`'
- `m=double('movie', :rating=>'R')`
Shortcut that combines the 2 previous examples
- `allow(Movie).to receive(:find).and_return(@fake_movie)`
If `Movie.find` is called, `@fake_movie` will be returned; if not called, no error

Useful methods and objects for controller specs: Your specs must be in the `spec/controllers` subdirectory for these methods to be available.

- `post '/movies/create', {title: 'Milk', rating: 'R'}`
Causes a POST request to `/movies/create` and passes the given hash as the value of `params`.
`get, put, delete` also available.
- `expect(response).to render_template('show')`
Checks that the controller action renders the `show` template for this controller's model
- `expect(response).to redirect_to(controller: 'movies', action: 'new')`
Checks that the controller action redirects to `MoviesController#new` rather than rendering a view

Figure 8.10: Some of the most useful RSpec methods introduced in this chapter. See the `rspec.info` documentation site for details and additional methods not listed here.

Assertions on method calls: can also negate by using either `to_not` or `not_to` (whichever reads better) in place of `to`

- `expect(Movie).to receive(:find).exactly(2).times`
Stubs `Movie.find` and ensures it's called exactly twice. Omit `exactly` if you don't care how many calls; `at_least()` and `at_most()` also available
- `expect(Movie).to receive(:find).with('Milk', 'R')`
Checks that `Movie.find` is called with exactly 2 arguments having these values
- `expect(Movie).to receive(:find).with(anything(), anything())`
Checks that `Movie.find` is called with 2 arguments whose values aren't checked
- `expect(Movie).to receive(:find).with(hash_including title: 'Milk')`
Checks that `Movie.find` is called with 1 argument that must be a hash (or something that quacks like one) that includes the key `:title` with the value `'Milk'`
- `expect(Movie).to receive(:find).with(no_args())`
Checks that `Movie.find` is called with zero arguments

Matchers:

- `expect(greeting).to eq 'bonjour'`
Compares its argument for equality with receiver of assertion
- `expect(value).to be >= 7`
Compares its argument with the given value; syntactic sugar for `expect(value).to(be.>=(7))`
- `expect(num).to be_within(delta).of(value)`
Test whether a numeric expression is within a threshold of some numeric value (useful for floating-point calculations)
- `expect(str).to match(/regexp/)`
Assert that the string matches the given regexp
- `expect(result).to be_remarkable`
Asserts that calling `remarkable?` (note question mark) on `result` returns a non-nil value

Figure 8.11: Continuation of summary of useful RSpec methods introduced in this chapter.

<https://gist.github.com/811c0b0c8f91fbbb8a4233898450267a>

```

1 class MyClass
2   def foo(x,y,z)
3     if x
4       if (y && z) then bar(0) end
5     else
6       bar(1)
7     end
8   end
9   def bar(x) ; @w = x ; end
10 end

```

Figure 8.12: A simple code example to illustrate basic coverage concepts.

8.7 Coverage Concepts and Types of Tests

How much testing is enough? A poor but unfortunately widely-given answer is “As much as you can before the release deadline.” A very coarse-grained alternative is the *code-to-test ratio*, the number of non-comment lines of code divided by number of lines of tests of all types. In production systems, this ratio is usually less than 1, that is, there are more lines of test than lines of app code. The command `rake stats` issued in the root directory of a Rails app computes this ratio based on the number of lines of RSpec tests and Cucumber scenarios.

Another widely-used metric that is more conservative is “when the rate of new bug reports falls below some threshold.” This formulation acknowledges that while code can never be proven bug-free, bugs are getting harder to find. But a more precise way to approach the question is to combine such metrics with **code coverage**. Since the goal of testing is to exercise the subject code in at least the same ways it would be exercised in production, what fraction of those possibilities is actually exercised by the test suite? Surprisingly, measuring coverage is not as straightforward as you might suspect. Figure 8.12 shows a simple fragment of code that we will use to illustrate the definitions of several commonly-used coverage terms.

- S0 or Method coverage: Is every method executed at least once by the test suite? Satisfying S0 requires calling `foo` and `bar` at least once each.
- S1 or Call coverage or Entry/Exit coverage: Has each method been called from every place it could be called? Satisfying S1 requires calling `bar` from both line 4 and line 6.
- C0 or Statement coverage: Is every statement of the source code executed at least once by the test suite, counting both branches of a conditional as a single statement? In addition to calling `bar`, satisfying C0 would require calling `foo` at least once with `x` true (otherwise the statement in line 4 will never be executed), and at least once with `y` false.
- C1 or Branch coverage: Has each branch been taken in each direction at least once? Satisfying C1 would require calling `foo` with both false and true values of `x` and with values of `y` and `z` such that `y && z` in line 4 evaluates once to true and once to false. A more stringent condition, *decision coverage*, requires that each *subexpression* that independently affects a conditional expression be evaluated to true and false. In this example, a test would additionally have to separately set `y` and `z` so that the condition `y && z` fails once for `y` being false and once for `z` being false.

Sometimes written with a subscript, S_0 .

- C2 or Path coverage: Has every possible route through the code been executed? In this simple example, where x, y, z are treated as booleans, there are 8 possible paths.
- Modified Condition/Decision Coverage (MCDC) combines a subset of the above levels: Every point of entry and exit in the program has been invoked at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome.

Achieving C0 coverage is relatively straightforward, and a goal of 100% C0 coverage is not unreasonable. Achieving C1 coverage is more difficult since test cases must be constructed more carefully to ensure each branch is taken at least once in each direction. C2 coverage is most difficult of all, and not all testing experts agree on the additional value of achieving 100% path coverage. Therefore, code coverage statistics are most valuable to the extent that they highlight undertested or untested parts of the code and show the overall comprehensiveness of your test suite. The SimpleCov gem⁸ is easy to configure and measures and displays the C0 and C1 coverage of your specs, allowing you to browse file-by-file to see which lines of your app were exercised by your test suites. If you have multiple suites, such as a set of Cucumber features as well as a set of specs, you must decide whether you need to know only whether a particular line of your app is exercised by *some* test, which may be either a Cucumber scenario or an RSpec example, or whether you need to know *which type* of test exercised it. SimpleCov does the former by default, but its instructions tell you how to do the latter.



This chapter, and the above discussion of coverage, have focused on unit tests. Chapter 7 explained how user stories could become automated acceptance tests; we can think of a Cucumber scenario as both a **system test**, because it exercises code in many different parts of the application in the same ways a user would, as well as an **acceptance test**, because a properly-written scenario reflects and verifies the behavior the user said they wanted. In SaaS, such tests may also be called *full-stack tests*, since a typical scenario exercises every part of the app from the browser-based UI to the database. Unlike unit tests, system tests rarely rely on test doubles to isolate behavior; on the contrary, the goal is to simulate real users as closely as possible.

Any test that covers more than one method but is not a full-stack test is generically an **integration test**. For example, an RSpec test of a controller action would probably stub out calls to the database and bypass the routing mechanism, neither of which is central to testing the controller action itself, but would probably include interactions with mechanisms such as parsing form input, which clearly are outside the controller action.

System and integration tests are important, but insufficient. Their resolution is poor: if an integration test fails, it is harder to pinpoint the cause since the test touches many parts of the code. Especially for system tests, coverage also tends to be poor because even though a single scenario touches many classes, it executes only a few code paths in each class. For the same reason, system and integration tests also tend to take longer to run. On the other hand, while unit tests run quickly and can isolate the subject code with great precision (improving both coverage resolution and error localization), because they rely on fake objects to isolate the subject code, they may mask problems that would only arise in integration tests. In other words, high assurance requires both good coverage and a mix of all three kinds of tests. Figure 8.13 summarizes the relative strengths and weaknesses of different types of tests.

We have focused on testing for correctness (“did you build the thing right”), but in practice, other flavors of tests are part of any comprehensive test suite:

	Unit	Functional	System/Integration
What is tested	One method/class	Several methods/classes	Large chunks of system
Running time	Very fast	Fairly fast	Slow
Error localization	Excellent	Moderate	Poor
Coverage	Excellent	Moderate	Poor
Use of doubles	Frequently	Occasionally	Rarely/never

Figure 8.13: Summary of the differences among unit tests, functional tests, and integration or whole-system tests.

- A ***smoke test*** consists of a minimal attempt to operate the software, to see whether anything is obviously wrong before running the rest of the test suite. For example, if a low-level coding error prevents a SaaS app from displaying its home page or accepting logins, there is no point in running further tests.
- ***Compatibility testing*** is less prominent in SaaS since the app developers control the server environment, but may still be important for testing the app's UI in different browsers. For example, Sauce Labs⁹ supports running SaaS integration tests on a variety of browsers and operating systems to check correct client behavior, and even captures a screencast of each run so you can visually check behaviors such as whether the same fonts look good in different browsers.
- ***Regression testing*** ensures that previously-fixed bugs do not reappear. We return to regression tests in Section 10.6.
- ***Performance, stress, and security*** testing are types of ***non-functional testing*** that ensure the software meets these operational criteria, which are particularly important for SaaS. We return to these in Chapter 12.
- ***Accessibility testing*** ensures that the software is usable by persons with disabilities. In SaaS, accessibility testing focuses primarily on the client-side user experience.



Summary

- Static and dynamic measures of coverage, including code-to-test ratio (reported by `rake stats`), C0 or C1 coverage (reported by SimpleCov), and C2 coverage, measure the extent to which your test suite exercises different paths in your code. SimpleCov provides one way to measure coverage for Ruby code, including Rails apps.
- Rather than setting “hard targets” for coverage levels, use coverage reports to identify under-tested parts of your app so you can enhance the test suite accordingly.
- Unit, integration, and system/acceptance tests differ in terms of their running time, resolution (ability to localize errors), ability to exercise a variety of code paths, and ability to perform a “reasonableness check” or so-called ***smoke test*** on the whole application. All three are vital to software assurance.
- In addition to ***functional tests*** that check correctness, we also need ***non-functional tests*** for accessibility, compatibility, security, and performance.

Self-Check 8.7.1. Why does high test coverage not necessarily imply a well-tested application?

- ◊ Coverage says nothing about the quality of the tests. However, low coverage certainly implies a poorly-tested application. ■

Self-Check 8.7.2. What is the difference between C0 code coverage and code-to-test ratio?

- ◊ C0 coverage is a *dynamic* measurement of what fraction of all statements are executed by a test suite. Code-to-test ratio is a *static* measurement comparing the total number of lines of code to the total number of lines of tests. ■

8.8 Other Testing Approaches and Terminology

The field of software testing is as broad and long-lived as software engineering and has its own literature. Its range of techniques includes formalisms for proving things about coverage, empirical techniques for selecting which tests to create, and directed-random testing. Depending on an organization’s “testing culture,” you may hear different terminology than we’ve used in this chapter. Ammann and Offutt’s *Introduction to Software Testing* (Ammann and Offutt 2008) is one of the best comprehensive references on the subject. Their approach is to divide a piece of code into **basic blocks**, each of which executes from the beginning to the end with no possibility of branching, and then join these basic blocks into a graph in which conditionals in the code result in graph nodes with multiple out-edges. We can then think of testing as “covering the graph”: each test case tracks which nodes in the graph it visits, and the fraction of all nodes visited at the end of the test suite is the test coverage. Ammann and Offutt go on to analyze various structural aspects of software from which such graphs can be extracted, and present systematic automated techniques for achieving and measuring coverage of those graphs.

One insight that emerges from this approach is that the levels of testing described in the previous section refer to *control flow coverage*, since they are only concerned with whether specific parts of the code are executed or not. Another important coverage criterion is *define-use coverage* or *DU-coverage*: given a variable x in some program, if we consider every place that x is assigned a value and every place that the value of x is used, DU-coverage asks what fraction of all *pairs* of define and use sites are exercised by a test suite. This condition is weaker than all-paths coverage but can find errors that control-flow coverage alone would miss.

Another testing term distinguishes **black-box tests**, whose design is based solely on the software’s external specifications, from **white-box tests** (also called *glass-box tests*), whose design reflects knowledge about the software’s implementation that is not implied by external specifications. For example, the external specification of a hash table might just state that when we store a key/value pair and later read that key, we should get back the stored value. A black-box test would specify a random set of key/value pairs to test this behavior, whereas a white-box test might exploit knowledge about the hash function to construct worst-case test data that results in many hash collisions. Similarly, white-box tests might focus on boundary values—parameter values likely to exercise different parts of the code.

Mutation testing, invented by Ammann and Offutt, is a test-automation technique in which small but syntactically legal changes are automatically made to the program’s source code, such as replacing $a+b$ with $a-b$ or replacing `if (c)` with `if (!c)`. Most such changes should cause at least one test to fail, so a mutation that causes *no* test to fail indicates either a lack of test coverage or a very strange program.

Fuzz testing consists of throwing random data at your application and seeing what breaks. In 2014, Google engineers reported¹⁰ that over a 2-year period, fuzz testing had helped find over 1,000 bugs in the open source video-processing utility `ffmpeg`. Fuzz testing has been particularly useful for finding security vulnerabilities that are missed by both manual code inspection and formal analysis, including stack and buffer overflows and unchecked null pointers. While such memory bugs do not arise in interpreted languages like Ruby and Python or in type-safe and memory-safe compiled languages such as Rust, fuzz testing can still find interesting bugs in SaaS apps. *Random* or *black box fuzzing* either generates completely random data or randomly mutates valid input data, such as changing certain bytes of metadata in a JPEG image to test the robustness of the image decoder. *Smart fuzzing* incorporates knowledge about the app’s structure and possibly a way to specify how to construct “realistic but fake” fuzz data. For example, smart-fuzzing SaaS might include randomizing the variables and values occurring in form postings or URIs, or attempting various cross-site scripting or SQL injection attacks, which we’ll discuss in Chapter 12. Finally, *white-box fuzzing* uses **symbolic execution**, which simulates execution of a program observing the conditions under which each branch is taken or not, then generates fuzzed inputs to exercise the branch paths not taken during the simulated execution. White-box fuzzing requires no explicit knowledge of the app’s structure and can theoretically provide C2 (all paths) coverage, but in practice the size of the search space is huge, and white-box fuzzing relies on a diverse set of “seed inputs” to be effective. This combination of formal analysis and random directed testing is representative of the current state of the art in thorough software testing. For a short contemporary survey of fuzz testing, see Godefroid’s article in Communications of the ACM (Godefroid 2020).

Summary of other testing approaches: We can think of testing as “covering a graph” of possible software behaviors. The graph can represent control flow (basic block coverage), variable assignment and usage (DU-coverage), a space of random inputs (fuzz testing), or a space of possible tests with respect to specific errors in the code (mutation testing). The different approaches are complementary and tend to catch different types of bugs.

Self-Check 8.8.1. *The Microsoft Zune music player had an infamous bug that caused all Zunes to “lock up” on December 31, 2008. Later analysis showed that the bug would be triggered on the last day of any leap year. What kinds of tests—black-box, glass-box, mutation, or fuzz—would have been likely to catch this bug?*

- ◊ A glass-box test for the special code paths used for leap years would have been effective. Fuzz testing might have been effective: since the bug occurs roughly once in every 1460 days, a few thousand fuzz tests would likely have found it. ■

8.9 CHIPS: The Acceptance Test/Unit Test Cycle



CHIPS 8.9: The Acceptance Test/Unit Test Cycle

<https://github.com/saasbook/hw-acceptance-unit-test-cycle>

In this assignment you will use a combination of Acceptance and Unit tests with the Cucumber and RSpec tools to add a “find movies with same director” feature to RottenPotatoes. You will alternate between BDD, in which you write one step of a Cucumber scenario, and TDD, in which you write tests and code to get that one step to pass, repeating until all steps in the scenario are complete. Along the way you’ll measure code coverage to make sure you are thoroughly testing your code.

8.10 The Plan-And-Document Perspective on Testing

The project manager takes the Software Requirements Specification from the requirements planning phase and divides it into the individual program units. Developers then write the code for each unit, and then perform unit tests to make sure they work. In many organizations, quality assurance staff performs the rest of the higher-level tests, such as module, integration, system, and acceptance tests.

There are three options on how to integrate the units and perform integration tests:

1. *Top-down integration* starts with the top of the tree structure showing the dependency among all the units. The advantage of top-down is that you quickly get some of the high level functions working, such as the user interface, which allows stakeholders to offer feedback for the app in time to make changes. The downside is that you have to create many stubs to get the app to limp along in this nascent form.
2. *Bottom-up integration* starts at the bottom of the dependency tree and works up. There is no need for stubs, as you can integrate all the pieces you need for a module. Alas, you don’t get an idea how the app will look until you get all the code written and integrated.
3. *Sandwich integration*, not surprisingly, tries to get the best of both worlds by integrating from both ends simultaneously. Thus, you try to reduce the number of stubs by selectively integrating some units bottom-up and try to get the user interface operational sooner by selectively integrating some units top-down.

The next step for the QA testers after integration tests is the system test, as the full app should work. This is the last step before showing it to customers for them to try out. Note that system tests cover both non-functional requirements, such as performance, and functional requirements of features found in the SRS.

One question for plan-and-document is how to decide when testing is complete. Typically, an organization will enforce a standard level of testing coverage before a product is ready for the customer. Examples might be statement coverage (all statements executed at least once), or all user input opportunities are tested with both good input and problematic input.

In the plan and document process, the final test is for the customers to try the product in their environment to decide whether they will accept the product or not. That is, the aim is validation, not just verification. In Agile development, the customer is involved in trying prototypes of the app early in the process, so there is no separate system test before running the acceptance tests.

As you should expect from the plan-and-document process, documentation plays an important role in testing. Figure 8.14 gives an outline for a test plan based on IEEE Standard 829-2008.

While testing is fundamental to software engineering, quoting another Turing Award winner:

Program testing can be used to show the presence of bugs, but never to show their absence!

—Edsger W. Dijkstra

Edsger W. Dijkstra
(1930–2002) received the 1972 Turing Award for fundamental contributions to developing programming languages.



Thus, there has been a great deal of research investigating approaches to verification beyond testing. Collectively, these techniques are known as **formal methods**. The general strategy is to start with a formal specification and prove that the behavior of the code follows the behavior of that spec. These are mathematical proofs, either done by a person or done by a computer. The two options are **automatic theorem proving** or **model checking**. Theorem proving uses a set of inference rules and a set of logical axioms to produce proofs from scratch. Model checking verifies selected properties by exhaustive search of all possible states that a system could enter during execution.

Because formal methods are so computationally intensive, they tend to be used only when the cost to repair errors is very high, the features are very hard to test, and the item being verified is not too large. Examples include vital parts of hardware like network protocols or safety critical software systems like medical equipment. For formal methods to actually work, the size of the design must be limited: the largest formally verified software to date is an operating system kernel that is less than 10,000 lines of code, and its verification cost about \$500 per line of code (Klein et al. 2010).

Hence, formal methods are *not* good matches to high-function software that changes frequently, as is generally the case for Software as a Service.

To put the cost of formal methods in perspective, NASA spent \$35M per year to maintain 420,000 lines of code¹¹ for the space shuttle, or about \$80 per line of code per year.

Top-Level Test Plan Outline

1. Introduction
 - 1.1. Document identifier
 - 1.2. Scope
 - 1.3. References
 - 1.4. System overview and key features
 - 1.5. Test overview
 - 1.5.1 Organization
 - 1.5.2 Overall test schedule
 - 1.5.3 Integrity level schema
 - 1.5.4 Resources summary
 - 1.5.5 Responsibilities
 - 1.5.6 Tools, techniques, methods, and metrics
2. Details of the Top-Level Test Plan
 - 2.1. Test processes including definition of test levels
 - 2.1.1 Process: Management
 - 2.1.1.1 Activity: Management of test effort
 - 2.1.2 Process: Acquisition
 - 2.1.2.1 Activity: Acquisition support test
 - 2.1.3 Process: Supply
 - 2.1.3.1 Activity: Planning test
 - 2.1.4 Process: Development
 - 2.1.4.1 Activity: Concept
 - 2.1.4.2 Activity: Requirements
 - 2.1.4.3 Activity: Design
 - 2.1.4.4 Activity: Implementation
 - 2.1.4.5 Activity: Test
 - 2.1.4.6 Activity: Installation/checkout
 - 2.1.5 Process: Operation
 - 2.1.5.1 Activity: Operational test
 - 2.1.6 Process: Maintenance
 - 2.1.6.1 Activity: Maintenance test
 - 2.2. Test documentation requirements
 - 2.3. Test administration requirements
 - 2.4. Test reporting requirements
 3. General
 - 3.1. Glossary
 - 3.2. Document change procedures and history

Figure 8.14: Outline of Top-Level Test Plan Documentation that follows the IEEE Standard 829-2008.

Tasks	<i>In Plan-and-Document</i>	<i>In Agile</i>
Test Plan and Documentation	Software Test Documentation such as IEEE Standard 829-2008	User stories
Order of Coding and Testing	1. Code units 2. Unit test 3. Module test 4. Integration test 5. System test 6. Acceptance test	1. Acceptance test 2. Integration test 3. Module test 4. Unit test 5. Code units
Testers	Developers for unit tests; QA testers for module, integration, system, and acceptance tests	Developers
When Testing Stops	Company policy (e.g., statement coverage, happy and sad user inputs)	All tests pass (green)

Figure 8.15: The relationship between the testing tasks of Plan-and-Document versus Agile methodologies.

Summary: Testing and formal methods reduce the risks of errors in designs.

- Unlike BDD/TDD, the plan-and-document process starts with writing code before you write the tests.
- Developers then perform unit tests.
- Especially in large projects, different people perform the higher-level tests. The integration tests options of putting the units together are top-down, bottom-up, or sandwich.
- Testers do a separate system test to ensure the product passes both functional and non-functional requirements before exposing it to customers for the final acceptance test.
- **Formal methods** rely on formal specifications and automated proofs or exhaustive state search to verify more than what testing can do, but they are so expensive to perform that today they are only applicable to small, stable, critical portions of hardware or software.
- Figure 8.15 shows the different test tasks for plan-and-document versus Agile processes.

Self-Check 8.10.1. Compare and contrast integration strategies including top-down, bottom-up, and sandwich integration.

◊ Top-down needs stubs to perform the tests, but it lets stakeholders get a feeling for how the app works. Bottom-up does not need stubs, but needs potentially everything written before stakeholders see it work. Sandwich integration works from both ends to try to get both benefits. ■

8.11 Fallacies and Pitfalls



Fallacy: 100% test coverage with all tests passing means no bugs.

There are many reasons this statement can be false. Complete test coverage says nothing about the quality of the individual tests. As well, some bugs may require passing a certain value as a method argument (for example, to trigger a divide-by-zero error), and control flow testing often cannot reveal such a bug. There may be bugs in the interaction between your app and an external service such as TMDb; stubbing out the service so you can perform local testing might mask such bugs.



Pitfall: Dogmatically insisting on 100% test coverage all passing (green) before you ship.

As we saw above, 100% test coverage is not only difficult to achieve at levels higher than C1, but gives no guarantees of bug-freedom even if you do achieve it. Test coverage is a useful tool for estimating the overall comprehensiveness of your test suite, but high confidence requires a variety of testing methods—integration as well as unit, fuzzing as well as hand-constructing test cases, define-use coverage as well as control-flow coverage, mutation testing to expose additional holes in the test strategy, and so on. Indeed, in Chapter 12 we will discuss operational issues such as security and performance, which call for additional testing strategies beyond the correctness-oriented ones described in this chapter.



Fallacy: You don't need much test code to be confident in the application.

While insisting on 100% coverage may be counterproductive, so is going to the other extreme. The *code-to-test ratio* in production systems (lines of noncomment code divided by lines of tests of all types) is usually less than 1, that is, there are more lines of test than lines of app code. As an extreme example, the SQLite database included with Rails contains over 1200 times as much test code as application code¹² because of the wide variety of ways in which it can be used and the wide variety of different kinds of systems on which it must work properly! While there is controversy over how useful a measure the code-to-test ratio is, given the high productivity of Ruby and its superior facilities for DRYing out your test code, a `rake stats` ratio between 0.2 and 0.5 is a reasonable target.



Pitfall: Relying too heavily on just one kind of test (unit, functional, integration).

Unit and functional tests are useful for covering rare corner cases and code paths. They also tell you how well-factored or modular your code is: a module or method that is easy to test has well-circumscribed external dependencies, which in turn reinforces that it can be well tested in isolation. On the other hand, because of that very isolation, even 100% unit test coverage tells you nothing about interactions *among* classes or modules. That's where integration-level tests such as the Cucumber scenarios of Chapter 7 are useful. Such tests touch only a tiny fraction of all possible application paths and exercise only a few behaviors in each method, but they do test the interfaces and interactions among modules. One rule of thumb used at Google and elsewhere (Whittaker et al. 2012) is “70–20–10”: 70% short and focused unit tests, 20% functional tests that touch multiple classes, 10% full-stack or integration tests. See Chapter 7 for the complementary pitfall of over-reliance on integration

tests.



Pitfall: Undertest integration points due to over-stubbing.

Mocking and stubbing confer many benefits, but they can also hide potential problems at integration points—places where one class or module interacts with another. Suppose `Movie` has some interactions with another class `Moviegoer`, but for the purposes of unit testing `Movie`, all calls to `Moviegoer` methods are stubbed out, and vice versa. Because stubs are written to “fake” the behavior of the collaborating class(es), we no longer know if `Movie` “knows how to talk to” `Moviegoer` correctly. Good coverage with functional and integration tests, which don’t stub out all calls across class boundaries, avoids this pitfall.



Pitfall: Writing tests after the code rather than before.

Thinking about “the code we wish we had” from the perspective of a test for that code tends to result in code that is testable. This seems like an obvious tautology until you try writing the code first without testability in mind, only to discover that surprisingly often you end up with mock trainwrecks (see next pitfall) when you do try to write the test.

In addition, in the traditional Waterfall lifecycle described in Chapter 1, testing comes after code development, but with SaaS that can be in “public beta” for months, no one would suggest that testing should only begin after the beta period. Writing the tests first, whether for fixing bugs or creating new features, eliminates this pitfall.



Pitfall: Mock Trainwrecks.

Mocks exist to help isolate your tests from their collaborators, but what about the collaborators’ collaborators? Suppose our `Movie` object has a `pics` attribute that returns a list of images associated with the movie, each of which is a `Picture` object that has a `format` attribute. You’re trying to mock a `Movie` object for use in a test, but you realize that the method to which you’re passing the `Movie` object is going to expect to call methods on its `pics`, so you find yourself doing something like this:

<https://gist.github.com/87f04dad610b1da187de8da024bf02af>

```
1 | movie = double('Movie', :pics => [double('Picture', :format => 'gif')])  
2 | expect(Movie.count_pics(movie)).to eq 1
```

This is called a *mock trainwreck*, and it’s a sign that the method under test (`count_pics`) has excessive knowledge of the innards of a `Picture`. In Chapters 9 and 11 we’ll encounter a set of additional guidelines to help you detect and resolve such **code smells**.



Pitfall: Inadvertently creating dependencies regarding the order in which specs are run, for example by using `before(:all)`.

If you specify actions to be performed only once for a whole group of test cases, you may introduce dependencies among those test cases without noticing. For example, if a `before :all` block sets a variable and test example A changes the variable’s value, test example B could come to rely on that change if A is usually run before B. Then B’s behavior in the future might suddenly be different if B is run first, which might happen because guard prioritizes running tests related to recently-changed code. Therefore it’s best to use `before :each` and `after :each` whenever possible.



Pitfall: Forgetting to re-prep the test database when the schema changes.

Remember that tests run against a separate copy of the database, not the database used in development (Section 4.2). Therefore, whenever you modify the schema by applying a migration, you must also run `rake db:test:prepare` to apply those changes to the test database; otherwise your tests may fail because the test code doesn't match the schema.

8.12 Concluding Remarks: TDD vs. Conventional Debugging

In this chapter we've used RSpec to develop a method using TDD with unit tests. Although TDD may feel strange at first, most people who try it quickly realize that they already use the unit-testing techniques it calls for, but in a different workflow. Often, a typical developer will write some code, assume it probably works, test it by running the whole application, and hit a bug. As an MIT programmer lamented at the first software engineering conference in 1968: "We build systems like the Wright brothers built airplanes—build the whole thing, push it off a cliff, let it crash, and start over again."

Once a bug has been hit, if inspecting the code doesn't reveal the problem, the typical developer would next try inserting print statements around the suspect area to print out the values of relevant variables or indicate which path of a conditional was followed. The TDD developer would instead write assertions using `expect`.

If the bug still can't be found, the typical developer might isolate part of the code by carefully setting up conditions to skip over method calls they don't care about or change variable values to force the code to go down the suspected buggy path. For example, they might do this by setting a breakpoint using a debugger and manually inspecting or manipulating variable values before continuing past the breakpoint. In contrast, the TDD developer would isolate the suspect code path using stubs and mocks to control what happens when certain methods are called and which direction conditionals will go.

By now, the typical developer is absolutely convinced that he'll certainly find the bug and won't have to repeat this tedious manual process, though this usually turns out to be wrong. The TDD developer has isolated each behavior in its own spec, so repeating the process just means re-running the spec.

In other words: If we write the code first and have to fix bugs, we end up using the same techniques required in TDD, but less efficiently and more manually, hence less productively.

But if we use TDD, bugs can be spotted immediately as the code is written. If our code works the first time, using TDD still gives us a regression test to catch bugs that might creep into this part of the code in the future.

- *How Google Tests Software* (Whittaker et al. 2012) is a rare glimpse into how Google has scaled up and adapted the techniques described in this chapter to instill a culture of testing that is widely admired by its competitors.
- The online RSpec documentation¹³ gives complete details and additional features used in advanced testing scenarios.
- *The RSpec Book* (Chelimsky et al. 2010) is the definitive published reference to RSpec and includes examples of features, mechanisms and best practices that go far beyond this introduction.

D. Chelimsky, D. Astels, B. Helmkamp, D. North, Z. Dennis, and A. Hellesøy. *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends (The Facets of Ruby Series)*. Pragmatic Bookshelf, 2010. ISBN 1934356379.

M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004. ISBN 9780131177055.

P. Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, 63(2):70–76, Feb 2020. doi: 10.1145/3363824.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM (CACM)*, 53(6):107–115, June 2010.

G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007. ISBN 0131495054. URL <https://www.amazon.com/xUnit-Test-Patterns-Refactoring-Code/dp/0131495054?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0131495054>.

J. A. Whittaker, J. Arbon, and J. Carollo. *How Google Tests Software*. Addison-Wesley Professional, 2012. ISBN 0321803027.

Notes

¹<https://xunitpatterns.com>

²<https://glossary.istqb.org/en/search>

³<https://themoviedb.org>

⁴<http://jmock.org/getting-started.html>

⁵<https://github.com/bblimke/webmock>

⁶<https://github.com/vcr>

⁷https://github.com/thoughtbot/factory_bot_rails

⁸<https://github.com/colszowka/simplecov>

⁹<https://saucelabs.com>

¹⁰<https://security.googleblog.com/2014/01/ffmpeg-and-thousand-fixes.html>

¹¹<http://www.fastcompany.com/magazine/06/writestuff.html>

¹²<http://www.sqlite.org/testing.html>

¹³<http://rspec.info>

9

Software Maintenance: Enhancing Legacy Software Using Refactoring and Agile Methods

Butler Lampson (1943–) was the intellectual leader of the legendary Xerox Palo Alto Research Center (Xerox PARC), which during its heyday in the 1970s invented graphical user interfaces, object-oriented programming, laser printing, and Ethernet. Three PARC researchers eventually won Turing Awards for their work there. Lampson received the 1994 Turing Award for contributions to the development and implementation of distributed personal computing environments: workstations, networks, operating systems, programming systems, displays, security, and document publishing.



There probably isn't a “best” way to build the system, or even any major part of it; much more important is to avoid choosing a terrible way, and to have clear division of responsibilities among the parts.

—Butler Lampson, *Hints for Computer System Design*, 1983

9.1	What Makes Code “Legacy” and How Can Agile Help?	270
9.2	Exploring a Legacy Codebase	273
9.3	Establishing Ground Truth With Characterization Tests	277
9.4	Comments and Commits: Documenting Code	279
9.5	Metrics, Code Smells, and SOFA	281
9.6	Method-Level Refactoring: Replacing Dependencies With Seams . .	286
9.7	The Plan-And-Document Perspective on Working With Legacy Code	292
9.8	Fallacies and Pitfalls	297
9.9	Concluding Remarks: Continuous Refactoring	298

Prerequisites and Concepts

Like a shark that must keep moving to live, software must change to remain viable. The big concepts in this chapter are that Agile development is a good approach to both maintain software and to enhance legacy code, and that **refactoring** is necessary on all development processes to keep code maintainable.

Concepts:

When writing code, **software metrics** and **code smells** can identify code that is hard to read. Transforming the code by **refactoring** should improve software metrics and eliminate code smells. Methods should be Short, do One thing, have Few arguments, and maintain a single level of Abstraction (SOFA).

To enhance legacy code using the Agile lifecycle:

- Understand the code at the *change points*, where you can plausibly make changes. Reading and enhancing comments is one way to understand the code.
- Explore how it works from all stakeholders' perspectives, which involves reading tests, design documents, and inspecting code.
- Write **characterization tests** to beef up test coverage *before* making changes to the code.

For the Plan-and-Document lifecycle:

- A *maintenance manager* runs the project during maintenance and estimates cost of **change requests**.
- Using cost-benefit analysis, a *Change Control Committee* triages change requests.
- Like Agile, maintenance relies on **regression testing** to ensure new releases work well and **refactoring** to make the code easier to maintain.

Surprisingly, the Agile process matches many needs of the maintenance phase of Plan-and-Development lifecycle.

Agile and Plan-and-Document processes have the same maintenance goals and many of the same techniques, but Agile suggests getting there by constant incremental refactoring rather than recoding all up front.

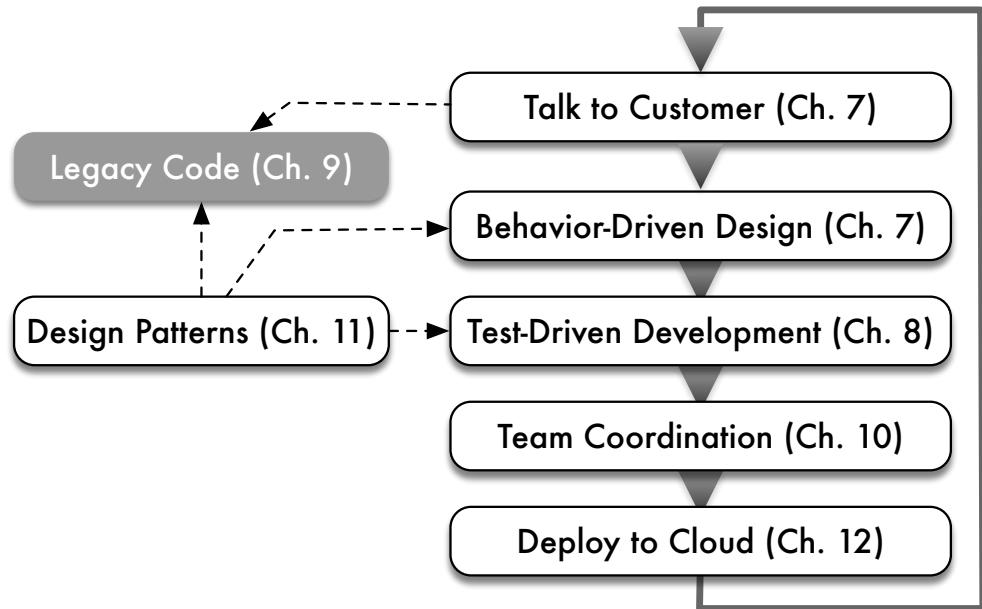


Figure 9.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter covers how Agile techniques can be helpful when enhancing legacy apps.

9.1 What Makes Code “Legacy” and How Can Agile Help?

1. *Continuing Change: [software] systems must be continually adapted or they become progressively less satisfactory*

—Lehman’s first law of software evolution.



As Chapter 1 explained, **legacy code** stays in use because it *still meets a customer need*, even though its design or implementation may be outdated or poorly understood. In this chapter we will show not only how to explore and come to understand a legacy codebase, but also how to apply Agile techniques to enhance and modify legacy code. Figure 9.1 highlights this topic in the context of the overall Agile lifecycle.

Maintainability is the ease with which a product can be improved. In software engineering, maintenance consists of four categories (Lientz et al. 1978):

- Corrective maintenance: repairing defects and bugs
- Perfective maintenance: expanding the software’s functionality to meet new customer requirements
- Adaptive maintenance: coping with a changing operational environment even if no new functionality is added; for example, adapting to changes in the production hosting environment
- Preventive maintenance: improving the software’s structure to increase future maintainability

Highly-readable unit, functional and integration tests (Chapter 8)	Git commit log messages (Chapter 10)
Lo-fi UI mockups and Cucumber-style user stories (Chapter 7)	Comments and RDoc-style documentation embedded in the code (Section 9.4)
Photos of whiteboard sketches about the application architecture, class relationships, etc. (Section 9.2)	Archived email, wiki/blog, notes, or video recordings of code and design reviews, for example in Campfire ¹ or Basecamp ² (Chapter 10)

Figure 9.2: While up-to-date formal design documents are valuable, Agile suggests we should place relatively more value on documentation that is “closer to” the working code.

Practicing these kinds of maintenance on legacy code is a skill learned by doing: we will provide a variety of techniques you can use, but there is no substitute for mileage. That said, a key component of all these maintenance activities is **refactoring**, a process that changes the structure of code (hopefully improving it) without changing the code’s functionality. The message of this chapter is that *continuous refactoring improves maintainability*. Therefore, a large part of this chapter will focus on refactoring.

Any piece of software, however well-designed, can eventually evolve beyond what its original design can accommodate. This process leads to maintainability challenges, one of which is the challenge of working with legacy code. Some developers use the term “legacy” when the resulting code is poorly understood because the original designers are long gone and the software has accumulated many **patches** not explained by any current design documents. A more jaded view, shared by some experienced practitioners (Glass 2002), is that such documents wouldn’t be very useful anyway. Once development starts, necessary design changes cause the system to drift away from the original design documents, which don’t get updated. In such cases developers must rely on *informal* design documents such as those that Figure 9.2 lists.

How can we enhance legacy software without good documentation? As Michael Feathers writes in *Working Effectively With Legacy Code* (Feathers 2004), there are two ways to make changes to existing software: *Edit and Pray* or *Cover and Modify*. The first method is sadly all too common: familiarize yourself with some small part of the software where you have to make your changes, edit the code, poke around manually to see if you broke anything (though it’s hard to be certain), then deploy and pray for the best.

In contrast, *Cover and Modify* calls for creating tests (if they don’t already exist) that cover the code you’re going to modify and using them as a “safety net” to detect unintended behavioral changes caused by your modifications, just as regression tests detect failures in code that used to work. The cover and modify point of view leads to Feathers’s more precise definition of “legacy code”, which we will use: *code that lacks sufficient tests to modify with confidence, regardless of who wrote it and when*. In other words, code that you wrote three months ago on a different project and must now revisit and modify might as well be legacy code.

Happily, the Agile techniques we’ve already learned for developing new software can also help with legacy code. Indeed, the task of understanding and evolving legacy software can be seen as an example of “embracing change” over longer timescales. If we inherit well-structured software with thorough tests, we can use BDD and TDD to drive addition of functionality in small but confident steps. If we inherit poorly-structured or untested code, we need to “bootstrap” ourselves into the desired situation in four steps:

1. Identify the *change points*, or places where you will need to make changes in the legacy system. Section 9.2 describes some exploration techniques that can help, and introduces one type of Unified Modeling Language (UML) diagram for representing the relationships among the main classes in an application.
2. If necessary, add **characterization tests** that capture how the code works now, to establish a baseline “ground truth” before making any changes. Section 9.3 explains what these tests are and how to create them using tools you’re already familiar with.
3. Determine whether the change points require **refactoring** to make the existing code more testable or to accommodate the required changes, for example, by breaking dependencies that make the code hard to test. Section 9.6 introduces a few of the most widely-used techniques from the many catalogs of refactorings that have evolved as part of the Agile movement.
4. Once the code around the change points is well factored and well covered by tests, make the required changes, using your newly-created tests as regressions and adding tests for your new code as in Chapters 7 and 8.

Summary of how Agile can help legacy code:

- Maintainability is the ease with which software can be enhanced, adapted to a changing operating environment, repaired, or improved to facilitate future maintenance. A key part of software maintenance is refactoring, a central part of the Agile process that improves the structure of software to make it more maintainable. Continuous refactoring therefore improves software maintainability.
- Working with legacy code begins with exploration to understand the code base, and in particular to understand the code at the *change points* where we expect to make changes.
- Without good test coverage, we lack confidence that refactoring or enhancing the code will preserve its existing behavior. Therefore, we adopt Feathers’s definition—“Legacy code is code without tests”—and create characterization tests where necessary to beef up test coverage before refactoring or enhancing legacy code.

■ Elaboration: Embedded documentation

 RDoc is a documentation system that looks for specially formatted comments in Ruby code and generates programmer documentation from them. It is similar to and inspired by JavaDoc. RDoc syntax is easily learned by example and from the Ruby Programming wikibook³. The default HTML output from RDoc can be seen, for example, in the Rails documentation⁴. Consider adding RDoc documentation as you explore and understand legacy code; running `rdoc .` (that’s a dot) in the root directory of a Rails app generates RDoc documentation from every `.rb` file in the current directory, `rdoc -help` shows other options, and `rake -T doc` in a Rails app directory lists other documentation-related Rake tasks.

Self-Check 9.1.1. Why do many software engineers believe that when modifying legacy code, good test coverage is more important than detailed design documents or well-structured

code?

- ◊ Without tests, you cannot be confident that your changes to the legacy code preserve its existing behaviors. ■

9.2 Exploring a Legacy Codebase

If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.

—Rob Pike

The goal of exploration is to understand the app from both the customers' and the developers' point of view. The specific techniques you use may depend on your immediate aims:

- You're brand new to the project and need to understand the app's overall architecture, documenting as you go so others don't have to repeat your discovery process.
- You need to understand just the moving parts that would be affected by a specific change you've been asked to make.
- You're looking for areas that need beautification because you're in the process of porting or otherwise updating a legacy codebase.

We can follow some “outside-in” steps to understand the structure of a legacy app at various levels:

1. Check out a scratch branch to run the app in a development environment
2. Learn and replicate the user stories, working with other stakeholders if necessary
3. Examine the database schema and the relationships among the most important classes
4. Skim all the code to quantify code quality and test coverage

Since operating on the live app could endanger customer data or the user experience, the first step is to get the application running in a development or staging environment in which perturbing its operation causes no inconvenience to users. Create a *scratch branch* of the repo that will never be merged with the mainline code and can therefore be used for experimentation. Create a development database if there isn't an existing one used for development. An easy way to do this is to clone the production database if it isn't too large, thereby sidestepping numerous pitfalls:

- The app may have relationships such as has-many or belongs-to that are reflected in the table rows. Without knowing the details of these relationships, you might create an invalid subset of data. Using RottenPotatoes as an example, you might inadvertently end up with a `review` whose `movie_id` and `moviegoer_id` refer to nonexistent movies or moviegoers.

<https://gist.github.com/617ca988e1425a36dc84e6e973554d1c>

```

1 # on production computer:
2 RAILS_ENV=production rake db:schema:dump
3 RAILS_ENV=production rake db:fixtures:extract
4 # copy db/schema.rb and test/fixtures/*.yml to development computer
5 # then, on development computer:
6 rake db:create      # uses RAILS_ENV=development by default
7 rake db:schema:load
8 rake db:fixtures:load

```

Figure 9.3: You can create an empty development database that has the same schema as the production database and then populate it with fixtures. Although Chapter 8 cautions against the abuse of fixtures, in this case we are using them to replicate known behavior from the production environment in your development environment.

- Cloning the database eliminates possible differences in behavior between production and development resulting from differences in database implementations, differences in how certain data types such as dates are represented in different databases, and so on.
- Cloning gives you realistic valid data to work with in development.

If you can't clone the production database, or you have successfully cloned it but it's too unwieldy to use in development all the time, you can create a development database by extracting fixture data from the real database⁵ using the steps in Figure 9.3.

Once the app is running in development, have one or two experienced customers demonstrate how they use the app, indicating during the demo what changes they have in mind (Nierstrasz et al. 2009). Ask them to talk through the demo as they go; although their comments will often be in terms of the user experience ("Now I'm adding Mona as an admin user"), if the app was created using BDD, the comments may reflect examples of the original user stories and therefore the app's architecture. Ask frequent questions during the demo, and if the maintainers of the app are available, have them observe the demo as well. In Section 9.3 we will see how these demos can form the basis of "ground truth" tests to underpin your changes.

Once you have an idea of how the app works, take a look at the database schema; Fred Brooks, Rob Pike, and others have all acknowledged the importance of understanding the data structures as a key to understanding the app logic. You can use an interactive database GUI to explore the schema, but you might find it more efficient to run `rake db:schema:dump`, which creates a file `db/schema.rb` containing the database schema in the migrations DSL introduced in Section 4.2. The goal is to match up the schema with the app's overall architecture.

Figure 9.4 shows a simplified Unified Modeling Language (UML) class diagram generated by the `railroadly` gem that captures the relationships among the most important classes and the most important attributes of those classes. While the diagram may look overwhelming initially, since not all classes play an equally important structural role, you can identify "highly connected" classes that are probably central to the application's functions. For example, in Figure 9.4, the `Customer` and `Voucher` classes are connected to each other and to many other classes. You can then identify the tables corresponding to these classes in the database schema.

Having familiarized yourself with the app's architecture, most important data structures, and major classes, you are ready to look at the code. The goal of inspecting the code is to get a sense of its overall quality, test coverage, and other statistics that serve as a proxy for



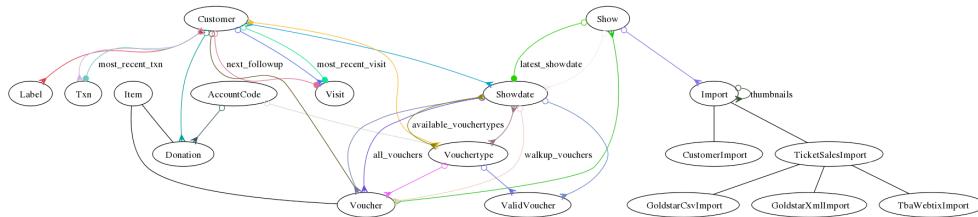


Figure 9.4: This simplified Unified Modeling Language (UML) class diagram, produced automatically by the `railroad` gem, shows the models in a Rails app that manages ticket sales, donations, and performance attendance for a small theater. Edges with arrowheads or circles show relationships between classes: a `Customer` has many `Visits` and `Vouchers` (open circle to arrowhead), has one `most_recent_visit` (solid circle to arrowhead), and has and belongs to many `Labels` (arrowhead to arrowhead). Plain edges show inheritance: `Donation` and `Voucher` are subclasses of `Item`. (All of the important classes here inherit from `ActiveRecord::Base`, but `railroad` draws only the app's classes.) We will see other types of UML diagrams in Chapter 11.

how painful it may be to understand and modify. Therefore, before diving into any specific file, run `rake stats` to get the total number of lines of code and lines of tests for each file; this information can tell you which classes are most complex and therefore probably most important (highest LOC), best tested (best code-to-test ratio), simple “helper” classes (low LOC), and so on, deepening the understanding you bootstrapped from the class diagram and database schema. (Later in this chapter we’ll show how to evaluate code with some additional quality metrics to give you a heads up of where the hairiest efforts might be.) If test suites exist, run them; assuming most tests pass, read the tests to help understand the original developers’ intentions. Then spend one hour (Nierstrasz et al. 2009) inspecting the code in the most important classes as well as those you believe you’ll need to modify (the *change points*), which by now you should be getting a good sense of.

Summary of legacy code exploration:

- The goal of exploration is to understand how the app works from multiple stakeholders’ points of view, including the customer requesting the changes and the designers and developers who created the original code.
- Exploration can be aided by reading tests, reading design documents if available, inspecting the code, and drawing or generating UML class diagrams to identify relationships among important entities (classes) in the app.
- Once you have successfully seen the app demonstrated in production, the next steps are to get it running in development by either cloning or fixturing the database and to get the test suite running in development.

Voucher < Item	
Knows:	reserved? fulfillment needed? checked-in? category
Does:	reserve cancel redeemable-dates changeable?
Belongs to:	Showdate Customer Vouchertype

Figure 9.5: A 3-by-5 inch (or A7 size) Class-Responsibility-Collaborator (CRC) card representing the `Voucher` class from Figure 9.4. The left column represents `Voucher`'s responsibilities—things it knows (instance variables) or does (instance methods). Since Ruby instance variables are always accessed through instance methods, we can determine responsibilities by searching the class file `voucher.rb` for instance methods and calls to `attr_accessor`. The right column represents `Voucher`'s collaborator classes; for Rails apps we can determine many of these by looking for `has_many` and `belongs_to` in `voucher.rb`.

■ ***Elaboration: Class–Responsibility–Collaborator (CRC) cards***

CRC cards (Figure 9.5) were proposed in 1989⁶ as a way to help with object-oriented design. Each card identifies one class, its responsibilities, and collaborator classes with which it interacts to complete tasks. As this external screencast⁷ shows, a team designing new code selects a user story (Section 7.1). For each story step, the team identifies or creates the CRC card(s) for the classes that participate in that step and confirms that the classes have the necessary Responsibilities and Collaborators to complete the step. If not, the collection of classes or responsibilities may be incomplete, or the division of responsibilities among classes may need to be changed. When exploring legacy code, you can create CRC cards to document the classes you find while following the flow from the controller action that handles a user story step through the models and views involved in the other story steps.

Self-Check 9.2.1. *What are some reasons it is important to get the app running in development even if you don't plan to make any code changes right away?*

- ◊ A few reasons include:
 1. For SaaS, the existing tests may need access to a test database, which may not be accessible in production.
 2. Part of your exploration might involve the use of an interactive debugger or other tools that could slow down execution, which would be disruptive on the live site.
 3. For part of your exploration you might want to modify data in the database, which you can't do with live customer data.
-

9.3 Establishing Ground Truth With Characterization Tests

If there are no tests (or too few tests) covering the parts of the code affected by your planned changes, you'll need to create some tests. How do you do this given limited understanding of how the code works now? One way to start is to establish a baseline for “ground truth” by creating ***characterization tests***: tests written after the fact that capture and describe the *actual, current* behavior of a piece of software, even if that behavior has bugs. By creating a **Repeatable** automatic test (see Section 8.1) that mimics what the code does right now, you can ensure that those behaviors stay the same as you modify and enhance the code, like a high-level regression test.

It's often easiest to start with an integration-level characterization test such as a Cucumber scenario, since these make the fewest assumptions about how the app works and focus only on the user experience. Indeed, while good scenarios ultimately make use of a “domain language” rather than describing detailed user interactions in imperative steps (Section 7.8), at this point it's fine to start with imperative scenarios, since the goal is to increase coverage and provide ground truth from which to create more detailed tests. Once you have some green integration tests, you can turn your attention to unit- or functional-level tests, just as TDD follows BDD in the outside-in Agile cycle.

Whereas integration-level characterization tests just capture behaviors that we observe without requiring us to understand *how* those behaviors happen, a unit-level characterization

<https://gist.github.com/b44d2059ebcccd4a1d45111884ba350b>

```

1 # WARNING! This code has a bug! See text!
2 class TimeSetter
3   def self.convert(d)
4     y = 1980
5     while (d > 365) do
6       if (y % 400 == 0 ||
7           (y % 4 == 0 && y % 100 != 0))
8         if (d > 366)
9           d -= 366
10          y += 1
11        end
12      else
13        d -= 365
14        y += 1
15      end
16    end
17    return y
18  end
19 end

```

Figure 9.6: This method is hard to understand, hard to test, and therefore, by Feathers’s definition of legacy code, hard to modify. In fact, it contains a bug—this example is a simplified version of a bug in the Microsoft Zune music player that caused any Zune booted on December 31, 2008, to freeze permanently, and for which the only resolution was to wait until the first minute of January 1, 2009, before rebooting.

<https://gist.github.com/ca621c822bd39c51e7d6aa27f783bbbe>

```

1 require 'simplecov'
2 SimpleCov.start
3 require './time_setter'
4 describe TimeSetter do
5   { 365 => 1980, 366 => 1981, 900 => 1982 }.each_pair do |arg,result|
6   it "#{arg} days puts us in #{result}" do
7     expect(TimeSetter.convert(arg)).to eq(result)
8   end
9 end
10 end

```

Figure 9.7: This simple spec, resulting from the reverse-engineering technique of creating characterization tests achieves 100% C0 coverage and helps us find a bug in Figure 9.6.

test seems to require us to understand the implementation. For example, consider the code in Figure 9.6. As we’ll discuss in detail in the next section, it has many problems, not least of which is that it contains a bug. The method `convert` calculates the current year given a starting year (in this case 1980) and the number of days elapsed since January 1 of that year. If 0 days have elapsed, then it is January 1, 1980; if 365 days have elapsed, it is December 31, 1980, since 1980 was a leap year; if 366 days have elapsed, it is January 1, 1981; and so on. How would we create unit tests for `convert` without understanding the method’s logic in detail?

Feathers describes a useful technique for “reverse engineering” specs from a piece of code we don’t yet understand: create a spec with an assertion that we know will probably fail, run the spec, and use the information in the error message to change the spec to match actual behavior. Essentially, we create specs that assert incorrect results, then fix the specs based on the actual test behavior. Our goal is to capture the current behavior as completely as possible so that we’ll immediately know if code changes break the current behavior, so we aim for 100% C0 coverage (even though that’s no guarantee of bug-freedom!), which is challenging because the code as presented has no seams. Doing this for `convert` results in

the specs in Figure 9.7 and even finds a bug in the process!

Screencast 9.3.1: Creating characterization specs for TimeSetter.

<http://youtu.be/8QwvqtMp5QM>

We create specs that assert incorrect results, then fix them based on the actual test behavior. Our goal is to capture the current behavior as completely as possible so that we'll immediately know if code changes break the current behavior, so we aim for 100% C0 coverage (even though that's no guarantee of bug-freedom!), which is challenging because the code as presented has no seams. Our effort results in finding a bug that crippled thousands of Microsoft Zune players on December 31, 2008.

Summary of characterization tests:

- To Cover and Modify when we lack tests, we first create characterization tests that capture how the code works now.
- Integration-level characterization tests, such as Cucumber scenarios, are often easier to start with since they only capture externally visible app behavior.
- To create unit- and functional-level characterization tests for code we don't fully understand, we can write a spec that asserts an incorrect result, fix the assertion based on the error message, and repeat until we have sufficient coverage.

■ Elaboration: What about specs that should pass, but don't?

If the test suite is out-of-date, some tests may be failing red. Rather than trying to fix the tests before you understand the code, mark them as “pending” (for example, using RSpec’s `pending` method) with a comment that reminds you to come back to them later to find out why they fail. Stick to the current task of preserving existing functionality while improving coverage, and don’t get distracted trying to fix bugs along the way.

Self-Check 9.3.1. State whether each of the following is a goal of unit and functional testing, a goal of characterization testing, or both:

- i Improve coverage
- ii Test boundary conditions and corner cases
- iii Document intent and behavior of app code
- iv Prevent regressions (reintroduction of earlier bugs)

◊ (i) and (iii) are goals of unit, functional, and characterization testing. (ii) and (iv) are goals of unit and functional testing, but non-goals of characterization testing. ■

9.4 Comments and Commits: Documenting Code

Not only does legacy code often lack tests and good documentation, but its comments are often missing or inconsistent with the code. We now offer a brief sermon on comments, so

```
https://gist.github.com/c350c1632f0e9fa7ca47c0c208cc9e8f
1 # Add one to i.
2 i += 1
3
4 # Lock to protect against concurrent access.
5 mutex = SpinLock.new
6
7 # This method swaps the panels.
8 def swap_panels(panel_1, panel_2)
9   # ...
10 end
```

Figure 9.8: Examples of bad comments, which state the obvious. You'd be surprised how often comments just mimic code even in otherwise well-written apps. (Thanks to John Ousterhout for these examples and some of this advice on comments.)

```
https://gist.github.com/f537017d1909733bbde757ea3abe951a
```

```
1 # Good Comment:
2 # Scan the array to see if the symbol exists
3
4 # Much better than:
5 # Loop through every array index, get the
6 # third value of the list in the content to
7 # determine if it has the symbol we are looking
8 # for. Set the result to the symbol if we
9 # find it.
```

Figure 9.9: Example of comments that raises the level of abstraction compared to comments that describe how you implement it.

that once you write successful characterization tests you can capture what you've learned by adding comments to the legacy code. Good comments have two properties:

1. They document things that aren't obvious from the code.
2. They are expressed at a higher level of abstraction than the code.

Figure 9.8 shows examples of comments that violate both properties, and Figure 9.9 shows a better example.

First, if you write comments as you code, much of what your code does is surely obvious to you, since you just wrote it. (Alas, *not* commenting as you go is a common defect of legacy code.) But if you or someone else reads your code later, long after you've forgotten those design ideas, comments should help you remember the non-obvious reasons you wrote the code the way you did. Examples of non-obvious things include the units for variables, code invariants, subtle problems that required a particular implementation, or unusual code that is there solely to work around some bug or account for a non-obvious boundary condition or corner case. In the case of legacy code, you are trying to add comments to document what went through another programmer's mind; once you figure it out, be sure to write it down before you forget!

Second, comments should raise the level of abstraction from the code. The programmer's goal is to write classes and other code that hides complexity; that is, to make it easier for others to use this existing code rather than re-create it themselves. Comments should therefore address concerns such as: What do I need to know to invoke this method? Are there preconditions, assumptions, or caveats? Among other jobs, a comment should provide enough of this information that someone who wants to call an existing class or method doesn't have to read its source code to figure these things out.

These guidelines are also generally true for commit messages, which you supply whenever you commit a set of code changes. However, one important principle is that you shouldn't put information in a commit message that a future developer will need to know while working on the code. Historical information—why a certain function was deleted or refactored, for example—is appropriate for including in a commit message. But information that a developer would need to know to use the code *as it exists now* should be in a comment, where the developer cannot fail to see it when they go to edit the code.

As with many other elements of Agile, when a process isn't working smoothly, it's trying to tell you something about your code. For example, we saw in Chapter 8 that when a test is hard to write due to the need for extensive mocking and stubbing, the test is trying to tell you that your code is not testable because it's poorly factored. Similarly here: if following the above guideline about comments vs. commits means you find yourself writing lots of cautionary caveats in the comments, your code is telling you that it might benefit from a refactoring cleanup so that you wouldn't need to post so many warning signs for the next developer who comes along with the intention of modifying it.

While virtually every other software engineering sermon in this book is paired with a tool that makes it easy for you to stay on the true path and for others to check if you have strayed, this is not the case for comments and commit messages. The only enforcement mechanism beyond self-discipline is inspection, which we discuss in Sections 10.4 and 10.7.

Summary of comments:

- Comments are best written at the same time as the code, not as an afterthought.
- Comments should not repeat what is obvious from the code. They should explain *why* the code is written the way it is, rather than simply repeating what it does.
- Comments should raise the level of abstraction from the code, describing what a logical block of code does rather than providing line-by-line details.
- Commits should include historical information about why the code is the way it is, but information that developers need *while* using the current code belongs in comments. If this leads to too many comments, your code may need cleanup.

Self-Check 9.4.1. *True or False: One reason legacy code is long lasting is because it typically has good comments.*

◊ False. We wish it were true. Comments are often missing or inconsistent with the code, which is one reason it is called legacy code rather than beautiful code. ■

9.5 Metrics, Code Smells, and SOFA

7. Declining Quality - The quality of [software] systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

—Lehman's seventh law of software evolution

A key theme of this book is that engineering software is about creating not just working code, but *beautiful* working code. This chapter should make clear why we believe this: beautiful code is easier and less expensive to maintain. Given that software can live much longer



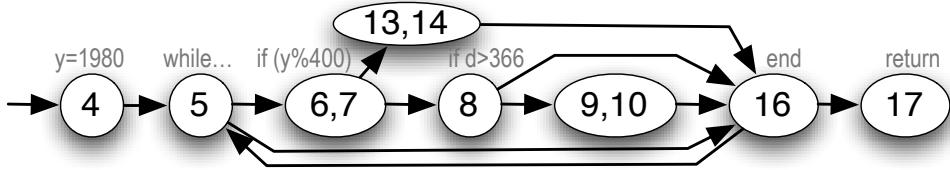


Figure 9.10: The node numbers in this control flow graph correspond to line numbers in Figure 9.6. Cyclomatic complexity is $E - N + 2P$ where E is the number of edges, N the number of nodes, and P the number of connected components. `convert` scores a cyclomatic complexity of 4 as measured by saikuro and an ABC score (Assignments, Branches, Conditionals) of 23 as measured by flog. Figure 9.11 puts these scores in context.

than hardware, even engineers whose aesthetic sensibilities aren't moved by the idea of beautiful code can appreciate the practical economic advantage of reducing lifetime maintenance costs.

How can you tell when code is less than beautiful, and how do you improve it? We've all seen examples of code that's less than beautiful, even if we can't always pin down the specific problems. We can identify problems in two ways: quantitatively using **software metrics** and qualitatively using **code smells**. Both are useful and tell us different things about the code, and we apply both to the ugly code in Figure 9.6.

Software metrics are quantitative measurements of code complexity, which is often an estimate of the difficulty of thoroughly testing a piece of code. Dozens of metrics exist, and opinion varies widely on their usefulness, effectiveness, and “normal range” of values. Most metrics are based on the **control flow graph** of the program, in which each graph node represents a **basic block** (a set of statements that are always executed together), and an edge from node A to node B means that there is some code path in which B's basic block is executed immediately after A's.

Figure 9.10 shows the control flow graph corresponding to Figure 9.6, which we can use to compute two widely-used indicators of method-level complexity:

1. **Cyclomatic complexity** measures the number of linearly-independent paths through a piece of code.
2. **ABC score** is a weighted sum of the number of Assignments, Branches and Conditionals in a piece of code.

These analyses are usually performed on source code and were originally developed for statically-typed languages. In dynamic languages, the analyses are complicated by metaprogramming and other mechanisms that may cause changes to the control flow graph at runtime. Nonetheless, they are useful first-order metrics, and as you might expect, the Ruby community has developed tools to measure them. `saikuro` computes a simplified version of cyclomatic complexity and `flog` computes a variant of the ABC score that is weighted in a way appropriate for Ruby idioms. Both of these and more are included in the `metric_fu` gem (part of the courseware). Running `rake metrics` on a Rails app computes various metrics including these, and highlights parts of the code in which multiple metrics are outside their recommended ranges. In addition, CodeClimate⁸ provides many of these metrics as a service: by creating an account there and linking your GitHub repository to it, you can view a “report card” of your code metrics anytime, and the report is automatically updated when

Plan-and-Document
software projects sometimes include specific contractual requirements based on software metrics.

Software engineer Frank McCabe Sr. invented the cyclomatic complexity metric in 1976.

Metric	Tool	Target score	Book Reference
Code-to-test ratio	rake stats	$\leq 1 : 2$	Section 8.7
C0 coverage	SimpleCov	$\geq 90\%$	Section 8.7
ABC score	flog (rake metrics)	$< 20/\text{method}$	Section 9.5
Cyclomatic	saikuro (rake metrics)	$< 10/\text{method}$	Section 9.5

Figure 9.11: A summary of useful metrics we've seen so far that highlight the connection between beauty and testability, including Ruby tools that compute them and suggested "normal" ranges. (The recommended value for cyclomatic complexity comes from NIST, the U.S. National Institute of Standards and Technologies.) The `metric_fu` gem includes `flog`, `saikuro`, and additional tools for computing metrics we'll meet in Chapter 11.

Name	Symptom	Possible refactorings
Shotgun Surgery	Making a small change to a class or method results in lots of little changes rippling to other classes or methods.	Use Move Method or Move Field to bring all the data or behaviors into a single place.
Data Clump	The same three or four data items seem to often be passed as arguments together or manipulated together.	Use Extract Class or Preserve Whole Object to create a class that groups the data together, and pass around instances of that class.
Inappropriate Intimacy	One class exploits too much knowledge about the implementation (methods or attributes) of another.	Use Move Method or Move Field if the methods really need to be somewhere else, use Extract Class if there is true overlap between two classes, or introduce a Delegate to hide the implementation.
Repetitive Boilerplate	You have bits of code that are the same or nearly the same in various different places (non-DRY).	Use Extract Method to pull redundant code into its own method that the repetitive places can call. In Ruby, you can even use <code>yield</code> to extract the "enclosing" code and having it yield back to the non-repetitive code.

Figure 9.12: Four whimsically-named code smells from Fowler's list of 22, along with the refactorings (some of which we'll meet in the next section) that might remedy the smell if applied. Refer to Fowler's book for the refactorings mentioned in the table but not introduced in this book.

you push new code to GitHub. Figure 9.11 summarizes useful metrics we've seen so far that speak to testability and therefore to code beauty.

The second way to spot code problems is by looking for **code smells**, which are structural characteristics of source code not readily captured by metrics. Like real smells, code smells call our attention to places that *may* be problematic. Martin Fowler's classic book on refactoring (Fowler et al. 1999) lists 22 code smells, four of which we show in Figure 9.12, and Robert C. Martin's *Clean Code* (Martin 2008) has one of the more comprehensive catalogs with an amazing 63 code smells, of which three are specific to Java, nine are about testing, and the remainder are more general.

Four particular smells that appear in Martin's *Clean Code* are worth emphasizing, because they are symptoms of other problems that you can often fix by simple refactorings. These four are identified by the acronym **SOFA**, which states that a well-written method should:

- be **Short**, so that its main purpose is quickly grasped;
- do only **One** thing, so testing can focus on thoroughly exercising that one thing;
- take **Few** arguments, so that all-important combinations of argument values can be tested;



Design smells (see Chapter 11) tell us when something's wrong in the way classes interact, rather than within the methods of a specific class.

What	Guideline	Example
Variable or class name	Noun phrase	PopularMovie, top_movies
Method with side effects	Verb phrase	pay_for_order, charge_credit_card!
Method that returns a value	Noun phrase	movie.producers, actor_list
Boolean variable or method	Adjective phrase	already_rated?, @is_oscar_winner

Figure 9.13: variable-naming guidelines based on simple English, excerpted from Green and Ledgard 2011. Given that disk space is free and modern editors have auto-completion that saves you retyping the full name, your colleagues will thank you for writing `@is_oscar_winner` instead of `OsWin`.

<https://gist.github.com/2183dec77d51a632f93923c6c3946fe6>

```

1 start with Year = 1980
2 while (days remaining > 365)
3   if Year is a leap year
4     then if possible, peel off 366 days and advance Year by 1
5   else
6     peel off 365 days and advance Year by 1
7 return Year

```

Figure 9.14: The computation of the current year given the number of days since the beginning of a start year (1980) is much more clear when written in pseudocode. Notice that *what the method does* is quick to grasp, even though each step would have to be broken down into more detail when turned into code. We will refactor the Ruby code to match the clarity and conciseness of this pseudocode.

- maintain a consistent level of Abstraction, so that it doesn't jump back and forth between saying *what to do* and saying *how to do it*.



Figure 9.6 violates at least the first and last of these, and exhibits other smells as well, as we can see by running `reek` on it:

<https://gist.github.com/223f7e7b28b390b650dd196f80048a2f>

```

1 time_setter.rb -- 5 warnings:
2 TimeSetter#self.convert calls (y + 1) twice (Duplication)
3 TimeSetter#self.convert has approx 6 statements (LongMethod)
4 TimeSetter#self.convert has the parameter name 'd' (UncommunicativeName)
5 TimeSetter#self.convert has the variable name 'd' (UncommunicativeName)
6 TimeSetter#self.convert has the variable name 'y' (UncommunicativeName)

```

Not DRY (line 2). Admittedly this is only a minor duplication, but as with any smell, it's worth asking ourselves why the code turned out that way.

Uncommunicative names (lines 4–6). Variable `y` appears to be an integer (lines 6, 7, 10, 14) and is related to another variable `d`—what could those be? For that matter, what does the class `TimeSetter` set the time to, and what is being converted to what in `convert?` Four decades ago, memory was precious and so variable names were kept short to allow more space for code. Today, there's no excuse for poor variable names; Figure 9.13 provides suggestions.

Too long (line 3). More lines of code per method means more places for bugs to hide, more paths to test, and more mocking and stubbing during testing. However, excessive length is really a symptom that emerges from more specific problems—in this case, failure to stick to a single level of Abstraction. As Figure 9.14 shows, `convert` really consists of a small number of high-level steps, each of which could be divided into sub-steps. But in the code, there is no way to tell where the boundaries of steps or sub-steps would be, making the method harder to understand. Indeed, the nested conditional in lines 6–8 makes it hard for a programmer to mentally “walk through” the code, and complicates testing since you have to select sets of test cases that exercise each possible code path.

The ancient wisdom that a method shouldn't exceed one screenful of code was based on text-only terminals with 24 lines of 80 characters. A modern 22-inch monitor shows 10 times that much, so guidelines like SOFA are more reliable today.

As a result of these deficiencies, you probably had to work hard to figure out what this relatively simple method does. (You might blame this on a lack of comments in the code, but once the above smells are fixed, there will be hardly any need for them.) Astute readers usually note the constants 1980, 365, and 366, and infer that the method has something to do with leap years and that 1980 is special. In fact, `convert` calculates the current year given a starting year of 1980 and the number of days elapsed since January 1 of that year, as Figure 9.14 shows using simple pseudocode. In Section 9.5, we will make the Ruby code as transparent as the pseudocode by **refactoring** it—applying transformations that improve its structure without changing its behavior.

A few specific examples of doing one thing are worth calling out because they occur frequently:

- Handling an exception is one thing. If method M computes something and also tries to handle various exceptions that could arise while doing so, consider splitting out a method M' that just does the work, and having M do exception handling and delegate the “real” work to M' .
- Queries (computing something) and commands (doing something that causes a side effect) are distinct, so a method should either compute something that is side-effect-free or it should cause a specific side effect, but not both. Such violations of *command-query separation* also complicate testing.

Summary

- Software metrics provide a quantitative measure of code quality. While opinion varies on which metrics are most useful and what their “normal” values should be (especially in dynamic languages such as Ruby), metrics such as cyclomatic complexity and ABC score can be used to guide your search toward code that is in particular need of attention, just as low C0 coverage identifies undertested code.
- Code smells provide qualitative but specific descriptions of problems that make code hard to read. Depending on which catalog you use, over 60 specific code smells have been identified.
- The acronym SOFA names four desirable properties of a method: it should be **S**hort, do **O**ne thing, have **F**ew arguments, and maintain a single level of **A**bstraction.

Self-Check 9.5.1. Give an example of a dynamic language feature in Ruby that could distort metrics such as cyclomatic complexity or ABC score.

- ◊ Any metaprogramming mechanism could do this. A trivial example is `s="if (d>=366){...}"; eval s`, since the evaluation of the string would cause a conditional to be executed even though there’s no conditional in the code itself, which contains only an assignment to a variable and a call to the `eval` method. A subtler example is a method such as `before_filter` (Section 5.1), which essentially adds a new method to a list of methods to be called before a controller action. ■

Self-Check 9.5.2. Which SOFA guideline—be *Short*, do *One thing*, have *Few arguments*, stick to a single level of *Abstraction*—do you think is most important from a unit-testability point of view?

- ◊ Few arguments implies fewer ways that code paths in the method can depend on the arguments, making testing more tractable. Short methods are certainly easier to test, but this property usually follows when the other three are observed. ■

9.6 Method-Level Refactoring: Replacing Dependencies With Seams

2. *Increasing Complexity - As [a software] system evolves, its complexity increases unless work is done to maintain or reduce it.*

—Lehman’s second law of software evolution

With the characterization specs developed in Section 9.3, we have a solid foundation on which to base our refactoring to repair the problems identified in Section 9.5. The term *refactoring* refers not only to a general process, but also to an instance of a specific code transformation. Thus, just as with code smells, we speak of a catalog of refactorings, and there are many such catalogs to choose from. We prefer Fowler’s catalog, so the examples in this chapter follow Fowler’s terminology and are cross-referenced to Chapters 6, 8, 9, and 10 of his book *Refactoring: Ruby Edition* (Fields et al. 2009). While the correspondence between code smells and refactorings is not perfect, in general each of those chapters describes a group of method-level refactorings that address specific code smells or problems, and further chapters describe refactorings that affect multiple classes, which we’ll learn about in Chapter 11.

Each refactoring consists of a descriptive name and a step-by-step process for transforming the code via small incremental steps, testing after each step. Most refactorings will cause at least temporary test failures, since unit tests usually depend on implementation, which is exactly what refactoring changes. A key goal of the refactoring process is to minimize the amount of time that tests are failing (red); the idea is that each refactoring step is small enough that adjusting the tests to pass before moving on to the next step is not difficult. If you find that getting from red back to green is harder than expected, you must determine if your understanding of the code was incomplete, or if you have really broken something while refactoring.

Getting started with refactoring can seem overwhelming: without knowing what refactorings exist, it may be hard to decide how to improve a piece of code. Until you have some experience improving pieces of code, it may be hard to understand the explanations of the refactorings or the motivations for when to use them. Don’t be discouraged by this apparent chicken-and-egg problem; like TDD and BDD, what seems overwhelming at first can quickly become familiar.

As a start, Figure 9.15 shows four of Fowler’s refactorings that we will apply to our code. In his book, each refactoring is accompanied by an example and an extremely detailed list of mechanical steps for performing the refactoring, in some cases referring to other refactorings that may be necessary in order to apply this one. For example, Figure 9.16 shows the first few steps for applying the Extract Method refactoring. With these examples in mind, we can refactor Figure 9.6.

Name (Chapter)	Problem	Solution
Extract method (6)	You have a code fragment that can be grouped together.	Turn the fragment into a method whose name explains the purpose of the method.
Decompose Conditional (9)	You have a complicated conditional (if-then-else) statement.	Extract methods from the condition, “then” part, and “else” part(s).
Replace Method with Method Object (6)	You have a long method that uses local variables in such a way that you cannot apply Extract Method.	Turn the method into its own object so that all the local variables become instance variables on that object. You can then decompose the method into other methods on the same object.
Replace Magic Number with Symbolic Constant (8)	You have a literal number with a particular meaning.	Create a constant, name it after the meaning, and replace the number with it.

Figure 9.15: Four example refactorings, with parentheses around the chapter in which each appears in Fowler’s book. Each refactoring has a name, a problem that it solves, and an overview of the code transformation(s) that solve the problem.

Fowler’s book also includes detailed mechanics for each refactoring, as Figure 9.16 shows.

1. Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it). If the code you want to extract is very simple, such as a single message or function call, you should extract it if the name of the new method reveals the intention of the code in a better way. If you can’t come up with a more meaningful name, don’t extract the code.
2. Copy the extracted code from the source method into the new target method.
3. Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method.
4. See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables.
5. Look to see whether any of these local-scope variables are modified by the extracted code. If one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned. If this is awkward, or if there is more than one such variable, you can’t extract the method as it stands. You may need to use *Split Temporary Variable* and try again. You can eliminate temporary variables with *Replace Temp with Query* (see the discussion in the examples).
6. Pass into the target method as parameters local-scope variables that are read from the extracted method.
7. ...

Figure 9.16: Fowler’s detailed steps for the Extract Method refactoring. In his book, each refactoring is described as a step-by-step code transformation process that may refer to other refactorings.

```
https://gist.github.com/44ef9cea09c14d79a37ef43f66168cf7
1 # NOTE: line 7 fixes bug in original version
2 class TimeSetter
3   def self.convert(d)
4     y = 1980
5     while (d > 365) do
6       if leap_year?(y)
7         if (d >= 366)
8           d -= 366
9           y += 1
10        end
11      else
12        d -= 365
13        y += 1
14      end
15    end
16    return y
17  end
18  private
19  def self.leap_year?(year)
20    year % 400 == 0 ||
21    (year % 4 == 0 && year % 100 != 0)
22  end
23 end
```

Figure 9.17: Applying the Extract Method refactoring to lines 6–7 of Figure 9.6 makes the conditional’s purpose immediately clear (line 6) by replacing the condition with a well-named method (lines 19–22), which we declare `private` to keep the class’s implementation details well encapsulated. For even more transparency, we could apply Extract Method again to `leap_year?` by extracting methods `every_400_years?` and `every_4-years-except_centuries?`.

Long method is the most obvious code smell in Figure 9.6, but that’s just an overall symptom to which various specific problems contribute. The high ABC score (23) of `convert` suggests one place to start focusing our attention: the condition of the `if` in lines 6–7 is difficult to understand, and the conditional is nested two-deep. As Figure 9.15 suggests, a hard-to-read conditional expression can be improved by applying the very common refactoring *Decompose Conditional*, which in turn relies on *Extract Method*. We move some code into a new method with a descriptive name, as Figure 9.17 shows. Note that in addition to making the conditional more readable, the separate definition of `leap_year?` makes the leap year calculation separately testable and provides a seam at line 6 where we could stub the method to simplify testing of `convert`, similar to the example in the Elaboration at the end of Section 8.4. In general, when a method mixes code that says *what to do* with code that says *how to do it*, this may be a warning to check whether you need to use Extract Method in order to maintain a consistent level of Abstraction.

The conditional is also nested two-deep, making it hard to understand and increasing `convert`’s ABC score. The *Decompose Conditional* refactoring also breaks up the complex condition by replacing each arm of the conditional with an extracted method. Notice, though, that the two arms of the conditional correspond to lines 4 and 6 of the pseudocode in Figure 9.14, both of which have the *side effects* of changing the values of `d` and `y` (hence our use of `!` in the names of the extracted methods). In order for those side effects to be visible to `convert`, we must turn the local variables into class variables throughout `TimeSetter`, giving them more descriptive names `@@year` and `@@days_remaining` while we’re at it. Finally, since `@@year` is now a class variable, we no longer need to pass it as an explicit argument to `leap_year?`. Figure 9.18 shows the result.

As long as we’re cleaning up, the code in Figure 9.18 also fixes two minor code smells. The first is uncommunicative variable names: `convert` doesn’t describe very well what

<https://gist.github.com/35461c627212c8098f6517855331ab02>

```
1 # NOTE: line 7 fixes bug in original version
2 class TimeSetter
3   ORIGIN_YEAR = 1980
4   def self.calculate_current_year(days_since_origin)
5     @@year = ORIGIN_YEAR
6     @@days_remaining = days_since_origin
7     while (@@days_remaining > 365) do
8       if leap_year?
9         peel_off_leap_year!
10      else
11        peel_off_regular_year!
12      end
13    end
14    return @@year
15  end
16  private
17  def self.peel_off_leap_year!
18    if (@@days_remaining >= 366)
19      @@days_remaining -= 366 ; @@year += 1
20    end
21  end
22  def self.peel_off_regular_year!
23    @@days_remaining -= 365 ; @@year += 1
24  end
25  def self.leap_year?
26    @@year % 400 == 0 ||
27    (@@year % 4 == 0 && @@year % 100 != 0)
28  end
29 end
```

Figure 9.18: We decompose the conditional in line 7 by replacing each branch with an extracted method. Note that while the total number of lines of code has increased, convert itself has become Shorter, and its steps now correspond closely to the pseudocode in Figure 9.14, sticking to a single level of Abstraction while delegating details to the extracted helper methods.

this method does, and the parameter name `d` is not useful. The other is the use of “magic number” literal constants such as 1980 in line 4; we apply *Replace Magic Number with Symbolic Constant* (Fowler chapter 8) to replace it with the more descriptive constant name `ORIGIN_YEAR`. What about the other constants such as 365 and 366? In this example, they’re probably familiar enough to most programmers, but if you saw 351 rather than 365, and if line 26 (in `leap_year?`) used the constant 19 rather than 400, you might not recognize the constants as being related to the **Hebrew calendar**. Remember that refactoring only improves the code for human readers; the computer doesn’t care. So in such cases use your judgment as to how much refactoring is enough.

In our case, re-running `flog` on the refactored code in Figure 9.18 brings the ABC score for the newly-renamed `calculate_current_year` from 23.0 down to 6.6, which is well below the suggested NIST threshold of 10.0. Also, `reek` now reports only two smells. The first is “low cohesion” for the helper methods `peel_off_leap_year` and `peel_off_regular_year`; this is a design smell, and we will discuss what it means in Chapter 11. The second smell is declaration of class variables inside a method. When we applied *Decompose Conditional* and *Extract Method*, we turned local variables into class variables `@@year` and `@@days_remaining` so that the newly-extracted methods could successfully modify those variables’ values. Our solution is effective, but clumsier than *Replace Method with Method Object* (Fowler chapter 6). In that refactoring, the original method `convert` is turned into an object *instance* (rather than a class) whose instance variables capture the object’s state; the helper methods then operate on the instance variables.

Figure 9.19 shows the result of applying such a refactoring, but there is an important caveat. So far, none of our refactorings have caused our characterization specs to fail, since the specs were just calling `TimeSetter.convert`. But applying *Replace Method With Method Object* changes the calling interface to `convert` in a way that makes tests fail. If we were working with real legacy code, we would have to find every site that calls `convert`, change it to use the new calling interface, and change any failing tests accordingly. In a real project, we’d want to avoid changes that needlessly break the calling interface, so we’d need to consider carefully whether the readability gained by applying this refactoring would outweigh the risk of introducing this breaking change.

Summary of refactoring:

- A refactoring is a particular transformation of a piece of code, including a name, a description of when to use the refactoring and what it does, and a detailed sequence of mechanical steps to perform it. Effective refactorings improve software metrics, eliminate code smells, or both.
- Although most refactorings will inevitably cause some existing tests to fail (if not, the code in question is probably undertested), a key goal of the refactoring process is to minimize the amount of time until those tests are modified and once again passing green.
- Sometimes applying a refactoring may result in recursively having to apply simpler refactorings first, as *Decompose Conditional* may require applying *Extract Method*.

<https://gist.github.com/e4028ce92d0109b232142a45c3d03c66>

```
1 # An example call would now be:
2 # year = TimeSetter.new(367).calculate_current_year
3 # rather than:
4 # year = TimeSetter.calculate_current_year(367)
5 class TimeSetter
6   ORIGIN_YEAR = 1980
7   def initialize(days_since_origin)
8     @year = ORIGIN_YEAR
9     @days_remaining = days_since_origin
10    end
11    def calculate_current_year
12      while (@days_remaining > 365) do
13        if leap_year?
14          peel_off_leap_year!
15        else
16          peel_off_regular_year!
17        end
18      end
19      return @year
20    end
21    private
22    def peel_off_leap_year!
23      if (@days_remaining >= 366)
24        @days_remaining -= 366 ; @year += 1
25      end
26    end
27    def peel_off_regular_year!
28      @days_remaining -= 365 ; @year += 1
29    end
30    def leap_year?
31      @year % 400 == 0 ||
32      (@year % 4 == 0 && @year % 100 != 0)
33    end
34  end
```

Figure 9.19: If we use Fowler's recommended refactoring, the code is cleaner because we now use instance variables rather than class variables to track side effects, but it changes the way `calculate_current_year` is called because it's now an instance method. This would break existing code and tests, and so might be deferred until later in the refactoring process.

■ *Elaboration: Refactoring and language choice*

Some refactorings compensate for programming language features that may encourage bad code. For example, one suggested refactoring for adding seams is *Encapsulate Field*, in which direct access to an object's instance variables is replaced by calls to getter and setter methods. This makes sense in Java, but as we've seen, getter and setter methods provide the *only* access to a Ruby object's instance variables from outside the object. (The refactoring still makes sense inside the object's own methods, as the Elaboration at the end of Section 2.3 suggests.) Similarly, the *Generalize Type* refactoring suggests creating more general types to improve code sharing, but Ruby's mixins and duck typing make such sharing easy. As we'll see in Chapter 11, it's also the case that some design patterns are simply unnecessary in Ruby because the problem they solve doesn't arise in dynamic languages.

Self-Check 9.6.1. Which is not a goal of method-level refactoring: (a) reducing code complexity, (b) eliminating code smells, (c) eliminating bugs, (d) improving testability?

- ◊ (c). While debugging is important, the goal of refactoring is to *preserve* the code's current behavior while changing its structure. ■

9.7 The Plan-And-Document Perspective on Working With Legacy Code

One reason for the term **lifecycle** from Chapter 1 is that a software product enters a maintenance phase after development completes. Roughly two-thirds of the costs are in maintenance versus one-third in development. One reason that companies charge roughly 10% of the price of software for annual maintenance is to pay the team that does the maintenance.

Organizations following Plan-And-Document processes typically have different teams for development and maintenance, with developers being redistributed onto new projects once the project is released. Thus, we now have a *maintenance manager* who takes over the role of the project manager during development, and we have *maintenance software engineers* who make changes to the code. Sadly, maintenance engineering has an unglamorous reputation, so it is typically performed by either the newest or least accomplished managers and engineers in an organization. Many organizations use different people for Quality Assessment (to do the testing) and for user documentation.

For software products developed using Plan-And-Document processes, the environment for maintenance is very different from the environment for development:

- *Working software*—A working software product is in the field during this whole phase, and new releases must not interfere with existing features.
- *Customer collaboration*—Rather than trying to meet a specification that is part of a negotiated contract, the goal for this phase is to work with customers to improve the product for the next release.
- *Responding to change*—Based on use of the product, customers send a stream of **change requests**, which can be new features as well as bug fixes. One challenge of the maintenance phase is prioritizing whether to implement a change request and in which release it should appear.

Change requests are called *maintenance requests* in IEEE standards.

Regression testing plays a much bigger role in maintenance to avoid breaking old features when developing new ones. Refactoring also plays a much bigger role, as you may need to

refactor to implement a change request or simply to make the code more maintainable. There is less incentive for the extra time and cost of refactoring in the initial phase of Plan-And-Document processes if the company developing the software is not the one that maintains it, which is one reason refactoring plays a smaller role during development.

As mentioned above, **change management** is based on change requests made by customers and other stakeholders to fix bugs or to improve functionality (see Section 10.7). They typically fill out **change request forms**, which are tracked using a ticket tracking system so that each request is responded to and resolved. A key tool for change management is a version control system, which tracks all modifications to all objects, as we describe in Sections 10.3 and 10.2.

The prior paragraphs should sound familiar, for we are describing Agile development; in fact, the three bullets are copied from the Agile Manifesto (see Section 1.3). Thus, *maintenance is essentially an Agile process*. Change requests are like user stories; the triaging of change requests is similar to the assignment of points and using Pivotal Tracker to decide how to prioritize stories; and new releases of the software product act as Agile iterations of the working prototype. Plan-and-document maintenance even follows the same strategy of breaking a large change request into many smaller ones to make them easier to assess and implement, just as we do with user stories assigned more than eight points (see Section 7.4). Hence, if the same team is developing and maintaining the software, nothing changes after the first release of the product when using the Agile lifecycle.

Although one paper reports successfully using an Agile process to maintain software developed using Plan-And-Document processes (Poole and Huisman 2001), normally an organization that follows Plan-And-Document for development also follows it for maintenance. As we saw in earlier chapters, this process expects a strong project manager who makes the cost estimate, develops the schedule, reduces risks to the project, and formulates a careful plan for all the pieces of the project. This plan is reflected in many documents, which we saw in Figures 7.6 and 8.14 and will see in the next chapter in Figures 10.11, 10.12, and 10.13. Thus, the impact of change in Plan-And-Document processes is not just the cost to change the code, but also to change the documentation and testing plan. Given the many more objects of Plan-And-Document, it takes more effort to synchronize to keep them all consistent when a change is made.

A *change control board* examines all significant requests to decide if the changes should be included in the next version of the system. This group needs estimates of the cost of a change to decide whether or not to approve the change request. The maintenance manager must estimate the effort and time to implement each change, much as the project manager did for the project initially (see Section 7.9). The group also asks the QA team for the cost of testing, including running all the regression tests and developing new ones (if needed) for a change. The documentation group also estimates the cost to change the documentation. Finally, the customer support group checks whether there is a workaround to decide if the change is urgent or not. Besides cost, the group considers the increased value of the product after the change when deciding what to do.

To help keep track what must be done in Plan-And-Document processes, you will not be surprised to learn that IEEE offers standards to help. Figure 9.21 shows the outline of a maintenance plan from the IEEE Maintenance Standard 1219-1998.

Ideally, changes can all be scheduled to keep the code, documents, and plans all in synchronization with an upcoming release. Alas, some changes are so urgent that everything else is dropped to try to get the new version to the customer as fast as possible. For example:

<i>Tasks</i>	<i>In Plan-and-Document</i>	<i>In Agile</i>
Customer change request	Change request forms	User story on 3x5 cards in Connextra format
Change request cost/time estimate	By Maintenance Manager	Points by Development Team
Triage of change requests	Change Control Board	Development team with customer participation
Roles	Maintenance Manager	N.A.
	Maintenance SW Engineers	Development team
	QA team	
	Documentation teams	
	Customer support group	

Figure 9.20: The relationship between the maintenance related tasks of Plan-and-Document versus Agile methodologies.

Table of Contents
1. Introduction 2. References 3. Definitions 4. Software Maintenance Overview 4.1 Organization 4.2 Scheduling Priorities 4.3 Resource Summary 4.4 Responsibilities 4.5 Tools, Techniques, and Methods 5. Software Maintenance Process 5.1 Problem/modification identification/classification, and prioritization 5.2 Analysis 5.3 Design 5.4 Implementation 5.5 System Testing 5.6 Acceptance Testing 5.7 Delivery 6. Software Maintenance Reporting Requirements 7. Software Maintenance Administrative Requirements 7.1 Anomaly Resolution and Reporting 7.2 Deviation Policy 7.3 Control Procedures 7.4 Standards, Practices, and Conventions 7.5 Performance Tracking 7.6 Quality Control of Plan 8. Software Maintenance Documentation Requirements

Figure 9.21: Maintenance plan outline from the IEEE 1219-1998 Standard for Maintenance in Systems and Software Engineering.

- The software product crashes.
- A security hole has been identified that makes the data collected by the product particularly vulnerable.
- New releases of the underlying operating system or libraries force changes to the product for it to continue to function.
- A competitor brings out a product or feature that if not matched will dramatically affect the business of the customer.
- New laws are passed that affect the product.

While the assumption is that the team will update the documentation and plans as soon as the emergency is over, in practice emergencies can be so frequent that the maintenance team can't keep everything in synch. Such a buildup is called a **technical debt**. The procrastination can lead to code that is increasingly difficult to maintain, which in turn leads to an increasing need to refactor the code as the code's "viscosity" makes it more and more difficult to add functionality cleanly. While refactoring is a natural part of Agile, it is less likely for the Change Control Committee to approve changes that require refactoring, as such changes are much more expensive. That is—as the name is intended to indicate—if you don't repay your technical debt, it grows: the "uglier" the code gets, the more error-prone and time-consuming it is to refactor!

Backfilling is the term maintenance engineers use to describe getting code back in synch after emergencies.

In addition to estimating the cost of each potential change for the Change Control Board, an organization's management may ask what will be the annual cost of maintenance of a project. The maintenance manager may base this estimate on software metrics, just as the project manager may use metrics to estimate the cost to develop a project (see Section 7.9). The metrics used for maintenance are different, as they are measuring the maintenance process. Examples of metrics that may indicate increased difficulty of maintenance include the average time to analyze or implement a change request and increases in the number of change requests made or approved.

At some point in the lifecycle of a software product, the question arises whether it is time for it to be replaced. An alternative that is related to refactoring is called *reengineering*. Like refactoring, the idea is to keep functionality the same but to make the code much easier to maintain. Examples include:

- Changing the database schema.
- Using a reverse engineering tool to improve documentation.
- Using a structural analysis tool to identify and simplify complex control structures.
- Using a language translation tool to change code from a procedure-oriented language like C or COBOL to an object-oriented language like C++ or Java.

The hope is that reengineering will be much less expensive and much more likely to succeed than reimplementing the software product from scratch.

Summary: The insight from this section is that you can think of Agile as a maintenance process, in that change is the norm, you are in continuous contact with the customer, and that new iterations of the product are routinely deployed to the customer as new releases. Hence, regression testing and refactoring are standard in the Agile process just as they are in the maintenance phase of Plan-and-Document. In Plan-and-Document processes:

- *Maintenance managers* play the role of project managers: they interface with the customer and upper management, make the cost and schedule estimates, document the maintenance plan, and manage the *maintenance software engineers*.
- Customers and other stakeholders issue ***change requests***, which a *Change Control Committee* triages based on the benefit of the change and cost estimates from the maintenance manager, the documentation team, and the QA team.
- ***Regression testing*** plays a bigger role in maintenance to ensure that new features do not interfere with old ones.
- ***Refactoring*** plays a bigger role as well, in part because there is often less refactoring in Plan-and-Document processes during product development than in Agile development.
- An alternative to starting over when the code becomes increasingly difficult to maintain is to *reengineer* the code to lower the cost of having a much more maintainable system.

One argument for Agile development is therefore as follows: if two-thirds of the cost of product are in the maintenance phase, why not use the same maintenance-compatible software development process for the whole lifecycle?

Self-Check 9.7.1. *True or False: The cost of maintenance usually exceeds the cost of development.*

- ◊ True. ■

Self-Check 9.7.2. *True or False: Refactoring and reengineering are synonyms.*

- ◊ False: While related terms, reengineering often relies on automatic tools and occurs as software ages and maintainability becomes more difficult, yet refactoring is a continuous process of code improvement that happens during both development and maintenance. ■

Self-Check 9.7.3. *Match the Plan-and-Document maintenance terms on the left to the Agile terms on the right:*

<i>Change request</i>	<i>Iteration</i>
<i>Change request cost estimate</i>	<i>Icebox, Active columns in Pivotal Tracker</i>
<i>Change request triage</i>	<i>Points</i>
<i>Release</i>	<i>User story</i>

- ◊ Change request \iff User story; Change request cost estimate \iff Points; Release \iff Iteration; and Change request triage \iff Icebox, Active columns in Pivotal Tracker.

■

9.8 Fallacies and Pitfalls



Pitfall: Using TDD and CRC to think only tactically and not strategically about design.

The extreme version of CRC cards seems to fit well with Agile: design and build the simplest thing that could possibly work, and embrace the fact that you'll need to change it later. But it's possible to take this approach too far. One suggestion from accomplished software craftsman and engineer **John Ousterhout**¹ is to "design it twice": use CRC cards to come up with a design, then put it aside and try a different design from scratch, perhaps thinking a bit adversarially about how you want to beat the team that did the original design. If you're unable to improve on the original design, you can be more confident that it represents a reasonable starting point. But surprisingly often, you'll find a simpler or more elegant design after you've had a chance to think through the problem the first time.



Pitfall: Conjoined Methods

Ousterhout (Ousterhout 2018) warns against creating *conjoined methods*: two methods that collaborate tightly in accomplishing one goal, so that there is a lot of interaction between them and neither can be effectively understood without also understanding the other. This advice is consistent with the SOFA advice that a method should do **One** thing (Ousterhout would say that each of the conjoined methods only does part of a thing) but is an easy pitfall to experience if you're overzealous in making methods **Short**. One sign of this is that it's nearly impossible to isolate one method from the other in tests; this is different from a helper method, which breaks out a well-defined subtask that can be individually tested.



Pitfall: Conflating refactoring with enhancement.

When you're refactoring or creating additional tests (such as characterization tests) in preparation to improve legacy code, there is a great temptation to fix "little things" along the way: methods that look just a little messy, instance variables that look obsolete, dead code that looks like it's never reached from anywhere, "really simple" features that look like something you could quickly add while doing other tasks. *Resist these temptations!* First, the reason to establish ground-truth tests ahead of time is to bootstrap yourself into a position from which you can make changes with confidence that you're not breaking anything. Trying to make such "improvements" in the absence of good test coverage invites disaster. Second, as we've said before and will repeat again, programmers are optimists: tasks that look trivial to fix may sidetrack you for a long time from your primary task of refactoring, or worse, may get the code base into an unstable state from which you must backtrack in order to continue refactoring. The solution is simple: when you're refactoring or laying groundwork, focus obsessively on completing those steps *before* trying to enhance the code.



Fallacy: It'll be faster to start from a clean slate than to fix this design.

Putting aside the practical consideration that management will probably wisely forbid you from doing this anyway, there are many reasons why this belief is almost always wrong.

¹Inventor of the **Magic** VLSI design software, the scripting language **Tcl**, the GUI toolkit **Tk**, and too many other software projects to mention!

First, if you haven’t taken the time to understand a system, you are in no position to estimate how hard it will be to redesign, and you will probably vastly underestimate the effort required, given programmers’ incurable optimism. Second, however ugly it may be, the current system *works*; a main tenet of doing short Agile iterations is “always have working code,” and by starting over you are immediately throwing that away. Third, if you use Agile methods in your redesign, you’ll have to develop user stories and scenarios to drive the work, which means you’ll need to prioritize them and write up quite a few of them to make sure you’ve captured at least the functionality of the current system. It would probably be faster to use the techniques in this chapter to write scenarios for just those parts of the system to be improved and drive new code from there, rather than doing a complete rewrite.

Does this mean you should *never* wipe the slate clean? No. As Rob Mee of Pivotal Labs points out, a time may come when the current codebase is such a poor reflection of the original design intent that it becomes a liability, and starting over may well be the best thing to do. (Sometimes this results from not refactoring in a timely way!) But in all but the most trivial systems, this should be regarded as the “nuclear option” when all other paths have been carefully considered and determined to be inferior ways to meet the customer’s needs.



Pitfall: Rigid adherence to metrics or “allergic” avoidance of code smells.

In Chapter 8 we warned that correctness cannot be assured by relying on a single type of test (unit, functional, integration/acceptance) or by relying exclusively on quantitative code coverage as a measure of test thoroughness. Similarly, code quality cannot be assured by any single code metric or by avoiding any specific code smells. Hence the `metric_fu` gem inspects your code for multiple metrics and smells so you can identify “hot spots” where multiple problems with the same piece of code call for refactoring.

9.9 Concluding Remarks: Continuous Refactoring

A ship in port is safe, but that’s not what ships are built for.

—Admiral Grace Murray Hopper

As we said in the opening of the chapter, modifying legacy code is not a task to be undertaken lightly, and the techniques required must be honed by experience. The first time is always the hardest. But fundamental skills such as refactoring help with both legacy code and new code, and as we saw, there is a deep connection among legacy code, refactoring, and testability and test coverage. We took code that was neither good nor testable—it scored poorly on complexity metrics and code smells, and isolating behaviors for unit testing was awkward—and refactored it into code that has much better metric scores, is easier to read and understand, and is easier to test. In short, we showed that *good methods are testable and testable methods are good*. We used refactoring to beautify existing code, but the same techniques can be used when performing the enhancements themselves. For example, if we need to add functionality to an existing method, rather than simply adding a bunch of lines of code and risk violating one or more SOFA guidelines, we can apply Extract Method to place the functionality in a new method that we call from the existing method. As you can see, this technique has the nice benefit that we already know how to develop new methods using TDD!

This observation explains why TDD leads naturally to good and testable code—it’s hard for a method not to be testable if the test is written first—and illustrates the rationale behind

the “refactor” step of Red–Green–Refactor. If you are refactoring constantly as you code, each individual change is likely to be small and minimally intrusive on your time and concentration, and your code will tend to be beautiful. When you extract smaller methods from larger ones, you are identifying collaborators, describing the purpose of code by choosing good names, and inserting seams that help testability. When you rename a variable more descriptively, you are documenting design intent.

But if you continue to encrust your code with new functionality *without* refactoring as you go, when refactoring finally does become necessary (and it will), it will be more painful and require the kind of significant scaffolding described in Sections 9.2 and 9.3. In short, refactoring will suddenly change from a background activity that takes incremental extra time to a foreground activity that commands your focus and concentration at the expense of adding customer value.



Since programmers are optimists, we often think “That won’t happen to me; I wrote this code, so I know it well enough that refactoring won’t be so painful.” But in fact, your code becomes legacy code the moment it’s deployed and you move on to focusing on another part of the code. Unless you have a time-travel device and can talk to your former self, you might not be able to divine what you were thinking when you wrote the original code, so the code’s clarity must speak for itself.

This Agile view of continuous refactoring should not surprise you: just as with development, testing, or requirements gathering, refactoring is not a one-time “phase” but an ongoing process. In Chapter 12 we will see that the view of continuous vs. phased also holds for deployment and operations.

It may be a surprise that the fundamental characteristics of Agile make it an excellent match to the needs of software maintenance. In fact, we can think of Agile as not having a development phase at all, but being in maintenance mode from the very start of its lifecycle!

Working with legacy code isn’t exclusively about refactoring, but as we’ve seen, refactoring is a major part of the effort. The best way to get better at refactoring is to do it a lot. Initially, we recommend you browse through Fowler’s refactoring book just to get an overview of the many refactorings that have been catalogued. We recommend the Ruby-specific version (Fields et al. 2009), since not all smells or refactorings that arise in statically-typed languages occur in Ruby; versions are available for other popular languages, including Java. We introduced only a few in this chapter; Figure 9.22 lists more. As you become more experienced, you’ll recognize refactoring opportunities without consulting the catalog each time.

Code smells came out of the Agile movement. Again, we introduced only a few from a more extensive catalog; Figure 9.23 lists more. Good programmers don’t deliberately create code with code smells; more often, the smells creep in as the code grows and evolves over time, sometimes beyond its original design. Pytel and Saleh’s *Rails Antipatterns* (Pytel and Saleh 2010) and Tucker’s treatment of code smells and refactoring in the context of contributing to open source software (Tucker et al. 2011) address these realistic situations.

We also introduced some simple software metrics; over four decades of software engineering, many others have been produced to capture code quality, and many analytical and empirical studies have been done on the costs and benefits of software maintenance. Robert Glass (Glass 2002) has produced a pithy collection of *Facts & Fallacies of Software Engineering*, informed by both experience and the scholarly literature and focusing in particular on the perceived vs. actual costs and benefits of maintenance activities.

Sandi Metz’s *Practical Object-Oriented Design in Ruby* (Metz 2012) covers object-

Category	Refactorings		
Composing Methods	<i>Extract method</i>	Replace temp with method	Introduce explaining variable
	<i>Replace method with method object</i>	Inline temp	Split temp variable
	Remove parameter assignments	Substitute algorithm	
Organizing Data	self-encapsulate field replace array/hash with Object	replace data value with object <i>Replace magic number with symbolic constant</i>	change value to reference
Simplifying Conditionals	<i>Decompose Conditional</i> Replace Conditional with Polymorphism Consolidate Duplicate Conditional Fragments	Consolidate Conditional Replace Type Code with Polymorphism Remove Control Flag	Introduce Assertion Replace Nested Conditional with Guard Clauses Introduce Null Object
Simplifying Method Calls	Rename Method Replace Parameter with Explicit Methods	Add Parameter Preserve Whole Object	Separate Query from Modifier Replace Error Code with Exception

Figure 9.22: Several more of Fowler's refactorings, with the ones introduced in this chapter in italics.

Duplicated Code	Temporary Field	Large Class	Long Parameter List
Divergent Change	Feature Envy	Primitive Obsession	Metaprogramming Madness
Data Class	Lazy Class	Speculative Generality	Parallel Inheritance Hierarchies
Refused Bequest	Message Chains	Middle Man	Incomplete Library Class
Too Many Comments	Case Statements	Alternative Classes with Different Interfaces	

Figure 9.23: More of Fowler's and Martin's code smells.

oriented design from the perspective of minimizing the cost of change, and expands on many of the themes in this chapter with practical examples.

The other primary sources for this chapter are Feathers's excellent practical treatment of working with legacy code (Feathers 2004), Nierstrasz and Demeyer's book on reengineering object-oriented software (Nierstrasz et al. 2009), and of course, the Ruby edition of Fowler's classic catalog of refactorings (Fields et al. 2009).

Finally, John Ousterhout's *A Philosophy of Software Design* (Ousterhout 2018) collects practical advice for structuring software at the class and method level, with a view towards robustness and manageability. It's aimed at more advanced developers and is an excellent source of wisdom when you're ready to go beyond the introductory material in this chapter.

M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004. ISBN 9780131177055.

J. Fields, S. Harvie, M. Fowler, and K. Beck. *Refactoring: Ruby Edition*. Addison-Wesley Professional, 2009. ISBN 0321603508.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN 0201485672.

R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002. ISBN 0321117425.

R. Green and H. Ledgard. Coding guidelines: Finding the art in the science. *Communications of the ACM*, 54(12):57–63, Dec 2011.

B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN 9780132350884.

S. Metz. *Practical Object-Oriented Design in Ruby: An Agile Primer (Addison-Wesley Professional Ruby)*. Addison-Wesley Professional, 2012. ISBN 0321721330. URL <http://poodr.com>.

O. Nierstrasz, S. Ducasse, and S. Demeyer. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2009. ISBN 395233412X.

J. K. Ousterhout. *A Philosophy of Software Design*. Yaknyam Press, 2018.

C. Poole and J. W. Huisman. Using extreme programming in a maintenance environment. *Software, IEEE*, 18(6):42–50, 2001.

C. Pytel and T. Saleh. *Rails AntiPatterns: Best Practice Ruby on Rails Refactoring (Addison-Wesley Professional Ruby Series)*. Addison-Wesley Professional, 2010. ISBN 9780321604811.

A. Tucker, R. Morelli, and C. de Silva. *Software Development: An Open Source Approach (Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series)*. CRC Press, 2011. ISBN 143981290X.

Notes

¹<http://campfirenow.com>

²<http://basecamphq.com>

³http://en.wikibooks.org/wiki/Ruby_Programming/RubyDoc

⁴<http://api.rubyonrails.org>

⁵<http://paulschreiber.com/blog/2010/06/15/rake-task-extracting-database-contents/>

⁶<http://c2.com/doc/oopsla89/paper.html>

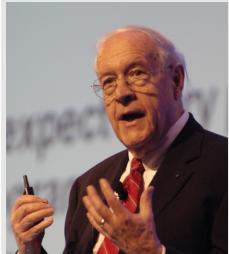
⁷<https://vimeo.com/24668095>

⁸<http://codeclimate.com>

10

Agile Teams

Frederick P. “Fred” Brooks, Jr. (1931–) wrote the classic software engineering book *The Mythical Man-Month* based on his years leading OS/360, the operating system for the highly successful IBM System/360 family of computers. This family was the first to feature instruction set compatibility across models, making it the first system to which the term “computer architecture” could be meaningfully applied. Brooks received the 1999 Turing Award for landmark contributions to computer architecture, operating systems, and software engineering.



There are no winners on a losing team, and no losers on a winning team.

—Fred Brooks, quoting North Carolina basketball coach Dean Smith, 1990

10.1	It Takes a Team: Two-Pizza and Scrum	304
10.2	Using Branches Effectively	306
10.3	Pull Requests and Code Reviews	311
10.4	Delivering the Backlog Using Continuous Integration	315
10.5	CHIPS: Agile Iterations	320
10.6	Reporting and Fixing Bugs: The Five R’s	320
10.7	The Plan-And-Document Perspective on Managing Teams	322
10.8	Fallacies and Pitfalls	330
10.9	Concluding Remarks: From Solo Developer to Teams	331

Prerequisites and Concepts

Prerequisites:

You should have a free GitHub account and be comfortable with the basic Git operations and commands for solo work: creating a new repo in your development environment and pushing it to GitHub, cloning an existing repo from GitHub to your development environment, adding and removing files in a repo, committing changes accompanied by descriptive log messages, and pushing your changes to GitHub. Even if your Integrated Development Environments (IDEs) provides buttons and menus for interacting with Git and GitHub, you need to be comfortable doing these operations from the command line.

To learn or refresh these skills, we recommend this basic Git guide¹ (developed for UC Berkeley CS61B Data Structures).

Concepts:

Whether Agile or Plan-and-Document, programming is now primarily a team sport. This chapter covers techniques that can help teams succeed in coordinating and delivering their work. All software teams rely on *version control* to manage code and configuration data, *code reviews* to ensure quality, *release management* to ship new versions, and *change management* while maintaining a shipped product, but the processes around these activities differ between Agile and P&D teams. The version of these concepts for Agile is:

- “Two-pizza” teams are four to nine people in size.
- Self-organizing teams follow the *Scrum* model, which relies on one teammate to act as the *Product Owner*, who represents the customer, and one to act as the *Scrum Lead*, who acts as a buffer between the team and external distractions. These roles rotate between the team members over time.
- Code reviews occur continuously, often as the result of *pull requests*, which start the process of integrating new team contributions into the mainline code.

For the Plan-and-Document lifecycle, you will become familiar with the same concepts from a different perspective:

- The *project manager* writes the contract, interfaces with the customer and upper management, recruits and manages the development team, resolves conflicts, and documents the plans for managing configurations and the project itself.
- While group sizes are similar to Agile, large teams can be created by combining groups into a hierarchy under the project manager, with each group having its own leader.

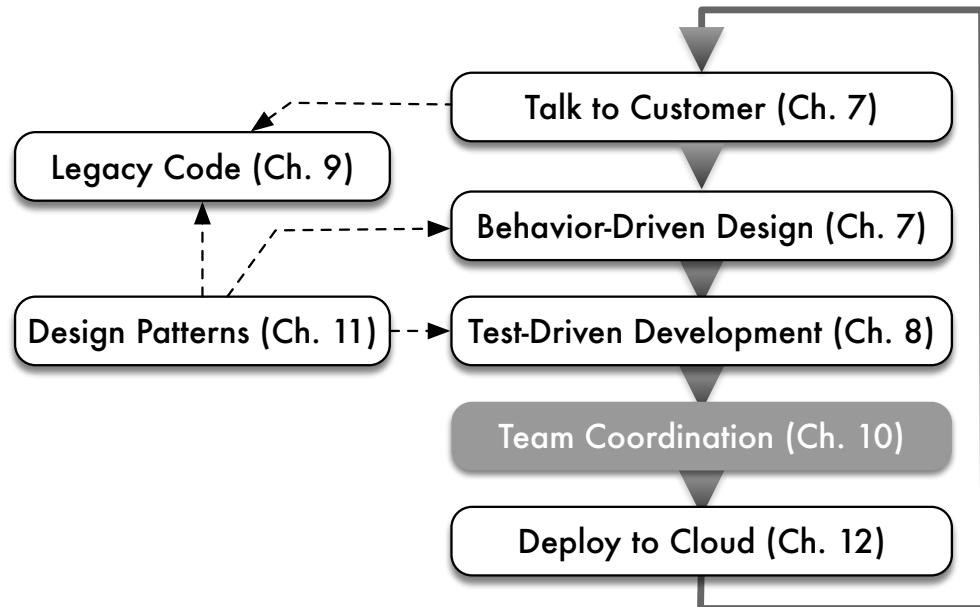


Figure 10.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter emphasizes evaluating the productivity of the team so as to come up with schedules that are more accurate by tracking velocity.

10.1 It Takes a Team: Two-Pizza and Scrum

The Six Phases of a Project: (1) Enthusiasm, (2) Disillusionment, (3) Panic, (4) Search for the guilty, (5) Punishment of the innocent, (6) Praise for non-participants.

—Dutch Holland (Holland 2004)

The days of the hero programmer are now past. Whereas once a brilliant individual could create breakthrough software, the rising bar on functionality and quality means software development is now primarily a team sport. Hence, success today means that not only do you have great design and coding skills, but that you work well with others and can help your team succeed. As the opening quote from Fred Brooks states, you cannot win if your team loses, and you cannot lose if your team wins.

Hence, the first step of a software development project is to form and organize a team. As to its size, a “two-pizza” team—a group that can be fed by two pizzas in a meeting—is typical for SaaS projects. Our discussions with senior software engineers suggest the typical team size varies by company, with four to nine developers being a typical range.

While there are many ways to organize a two-pizza team, a popular one today is **Scrum** (Schwaber and Beedle 2001). Its frequent short meetings—15 minutes every day at the same place and time—inspire the name. During the scrum, each team member answers three questions:

1. What have you done since yesterday?
2. What are you planning to do today?
3. Are there any impediments or stumbling blocks?

Jeff Bezos, the CEO of Amazon who received his college degree in computer science, coined the two-pizza characterization of team size.

The benefit of these daily scrums is that by understanding what each team member is doing and has already completed, the team can identify work that would help others make faster progress. Indeed, daily scrums are sometimes referred to as *standups*, implying that the meeting should be kept short enough that everyone can remain standing the entire time.

When combined with the weekly or biweekly iteration model of Agile to collect the feedback from all the stakeholders, the Scrum organization makes it more likely that the rapid progress will be towards what the customers want. Rather than use the Agile term iteration, Scrum uses the term **sprint**.

A Scrum has three main roles:

1. **Team**—A two-pizza size team that delivers the software.
2. **Scrum Lead**—A team member who acts as buffer between the Team and external distractions, keeps the team focused on the task at hand, enforces team rules, and removes impediments that prevent the team from making progress. One example is enforcing **coding standards**, which are style guidelines that improve the consistency and readability of the code.
3. **Product Owner**—A team member (not the Scrum Lead) who represents the voice of the customer and prioritizes user stories.

A scrum is held on every minor infraction in the sport of rugby. The game stops to bring the players together for a quick "meeting" in order to restart the game.



Scrum relies on self-organization, and team members often rotate through different roles. For example, we recommend that each team member rotate through the Product Owner role, changing on every iteration or sprint.

In any group working together, conflicts can occur around which technical direction the group should go. Depending in part on the personalities of the members of the team, they may not be able to quickly reach agreement. One approach to resolving conflicts is to start with a list of all the items on which the sides agree, as opposed to starting with the list of disagreements. This technique can make the sides see that perhaps they are closer together than they thought. Another approach is for each side to articulate the other's arguments. This technique makes sure both sides understand what the arguments are, even if they don't agree with some of them. This step can reduce confusion about terms or assumptions, which may be the real cause of the conflict.

Of course, such an approach requires great team dynamics. Everyone on the team should ideally feel **psychological safety**—the belief that they will not be humiliated for voicing their ideas to the team, even if those ideas might turn out to be wrong. Indeed, a two-year study by Google² found that the most effective Google teams weren't the ones with the most senior engineers or the smartest people, but the teams with high psychological safety. One way Agile teams promote psychological safety is to do a short Retrospective meeting, often shortened to "retro," at the end of each iteration. A typical format for the retro focuses on Plus/Minus/Interesting (PMI): each team member writes down, perhaps anonymously at first, what they thought went well, went poorly, and was unusual or noteworthy (neither good nor bad) during the iteration. The PMI items are often not technical, for example, "When I brought up my concern about some new code to Armando, I felt he was dismissive about my idea" or "Dave really helped me with a bug I'd been chasing this week without making me feel stupid." All team members then review the items (and where appropriate reveal their identities, or say "I agree," or "I noticed that too"), noting especially if some items were raised by more than one team member. The PMI items can also be compared to the previous iteration's retrospective items, since one goal of Agile is continuous improvement.

Postfacto is a free Web-based tool from Pivotal Labs that facilitates retros.



The rest of this chapter focuses on coordinating the work of the team, and in particular the use of version control tools to support coordination. How is the code repository managed? What happens if team members accidentally make conflicting changes to a set of files? Which lines in a given file were changed, when, and by whom? How does one developer work on a new feature without introducing problems into stable code? How does the team ensure the quality of the code and tests on an ongoing basis as more contributions accrete in the repository? As we will see, when software is developed by a team rather than an individual, version control can be used to address these questions using **merging** and **branching**. Both tasks involve combining changes made by many developers into a single code base, a task that sometimes requires manual resolution of conflicting changes.

Summary: SaaS is a good match for two-pizza teams and Scrum, a self-organized small team that meets daily. Two team members take on the additional roles of Scrum Lead, who removes impediments and keeps the team focused, and Product Owner, who speaks for the customer. It can be helpful to follow structured strategies to resolve conflicts when they occur and to reflect on past work in a retrospective meeting.

■ *Elaboration: Coding standards*

Coding standards or *stylesheets* are style guidelines that everyone on the team is expected to follow, usually related to indentation, variable naming, and so on. The goal is to improve the consistency and readability of the code. For example, here is one for Rails³. Some projects keep their own coding style guidelines in a document in the project’s main repository; others adhere to guidelines for whichever frameworks or languages the project uses.

Self-Check 10.1.1. *True or False: Scrum is an appropriate methodology when it is difficult to plan ahead.*

- ◊ True: Scrum relies more on real-time feedback than on the traditional management approach of central planning with command and control. ■

10.2 Using Branches Effectively

Besides taking snapshots of your work and backing it up, version control also lets you manage multiple versions of a project’s code base simultaneously, for example, to allow part of the team to work on an experimental new feature without disrupting working code, or to fix a bug in a previously-released version of the code that some customers are still using.

Branches are designed for such situations. Rather than thinking of commits as just a sequence of snapshots, we should instead think of a directed, acyclic *graph* of commits. When a new repo is created, by default it contains only a single branch, usually called the main branch, on which a linear sequence of commits is made. At any point in time, a new branch can be created that “splits off” from any commit of an existing branch, creating a copy of the codebase as it exists at that commit. As soon as a branch is created, that branch and the one from which it was split are separate: commits to one branch don’t affect the other, though depending on project needs, commits in either may be merged back into the other. Indeed, branches can even be split off from other branches, but overly complex branching structures offer few benefits and are difficult to maintain. Finally, unlike a real tree branch, a repo branch can be merged back into another branch later—either the branch from which

Prior to June 2020, the main branch was usually called *master*.

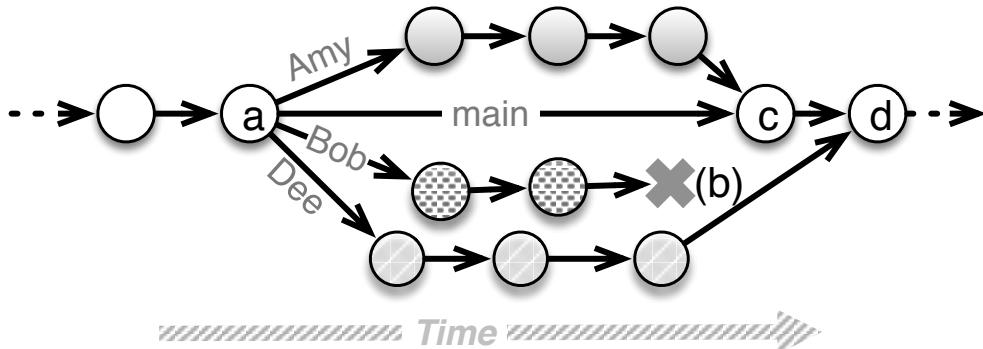


Figure 10.2: Each circle represents a commit. Amy, Bob and Dee each start branches based on the same commit (a) to work on different RottenPotatoes features. After several commits, Bob decides his feature won't work out, so he deletes his branch (b); meanwhile, Amy completes her work and merges her feature branch back into the main branch, creating the merge-commit (c). Finally, Dee completes her feature, but since the main branch has changed due to Amy's merge-commit (c), Dee has to do some manual conflict resolution to complete her merge-commit (d).

it split off, or some other branch. A branch can also be deleted, for example, if the project decides to abandon the work-in-progress that branch represents.

We highlight two common branch management strategies that can be used together or separately, both of which strive to ensure that the main branch always contains a stable working version of the code. Figure 10.2 shows a *feature branch*, which allows a developer or sub-team to make the changes necessary to implement a particular feature without affecting the main branch until the changes are complete and tested. If the feature is merged into the main branch and a decision is made later to remove it (perhaps it failed to deliver the expected customer value), the specific commits related to the merge of the feature branch can sometimes be undone, as long as there haven't been many changes to the main branch that depend on the new feature.

Topic branch is a generic term that may refer to feature, release, or bug fix branches.

Figure 10.3 shows how *release branches* are used to fix problems found in a specific release. They are widely used for delivering non-SaaS products such as libraries or gems whose releases are far enough apart that the main branch may diverge substantially from the most recent release branch. For example, the Linux kernel, for which developers check in thousands of lines of code per day, uses release branches to designate stable and long-lived releases of the kernel. Release branches often receive multiple merges from the development or main branch and contribute multiple merges to it. Release branches are less common in delivering SaaS because of the trend toward continuous integration/continuous deployment (Section 1.4): if you deploy several times per week, the deployed version won't have time to get out of sync with the main branch, so you might as well deploy directly from the main branch. We discuss continuous deployment further in Chapter 12.

Figure 10.4 shows some commands for manipulating Git branches. At any given time, the *current branch* is whichever one you're working on in your copy of the repo. Since in general each copy of the repo contains all the branches, you can quickly switch back and forth between branches in the same repo (but see Fallacies and Pitfalls for an important caveat about doing so).

Small teams working on a common set of features commonly use a *shared-repository* model for managing the repo: one particular copy of the repo, referred to as the *origin* repo, is designated as authoritative, and all developers agree to push their changes to the origin

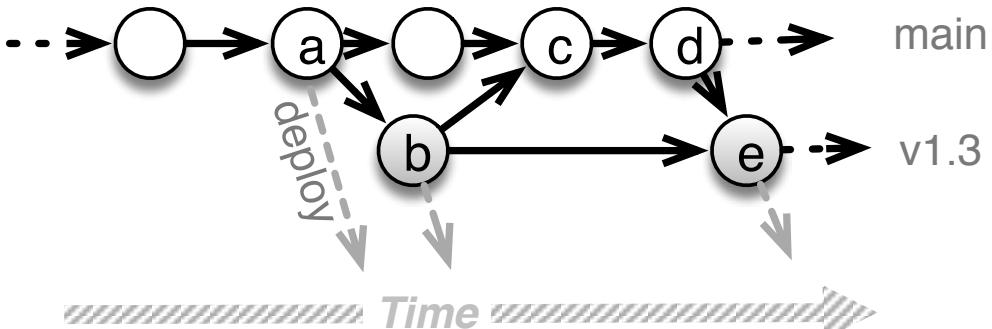


Figure 10.3: (a) A new release branch is created to “snapshot” version 1.3 of RottenPotatoes. A bug is found in the release and the fix is committed in the release branch (b); the app is redeployed from the release branch. The commit(s) containing the fix are merged into the main branch (c), but the code in the main branch has evolved sufficiently from the code in the release that manual adjustments to the bug fix need to be made. Meanwhile, the dev team working on the main branch finds a critical security flaw and fixes it with one commit (d). The specific commit containing the security fix can be merged into the release branch (e) using `git cherry-pick`, since we don’t want to apply any other main branch changes to the release branch except for this fix.

Many earlier VCSs such as Subversion supported only the shared-repository model of development, and the “one true repo” was often called the *master*, a term that meant something quite different in Git and was abandoned in 2020 in favor of *main*.

PaaS
(Platform-as-a-Service)
deployment servers such as
Heroku often appear as a
Git remote; pushing code to
that remote deploys the
pushed version of the app.

`git pull` actually
combines two separate
commands: `git fetch`,
which copies new commits
from the origin, and `git
merge`, which tries to
reconcile the commits with
those in the branch on the
local clone.

and periodically pull from the origin to get others’ changes. Famously, Git itself doesn’t care which copy of the repo is authoritative—any developer can *pull* changes from or *push* changes to any other developer’s copy of that repo if the repo’s permissions are configured to allow it—but for small teams, it’s convenient (and conventional) for the origin repo to reside in the cloud on a service such as GitHub. Each team member can *clone* the origin repo onto their development computer, do their work, make their commits on their local clone, and periodically push their commits to the origin. From the point of view of each developer’s local clone, the origin is one of possibly several *remote* copies of the repo, or simply “remotes.” In the shared-repository model, the origin repo is usually the only remote. Section 10.4 describes scenarios in which there may be multiple remotes.

As Figure 10.4 shows, the `git push` and `git pull` commands usually specify which copy of the repository and which branch should be involved in a push or pull operation. If Amy commits changes to her clone of the repo, those changes aren’t visible to her teammate Bob until she does a push and Bob does a pull.

This raises the possibility of a *merge conflict* scenario. Returning to Figure 10.2, suppose that Dee’s feature branch results in changes to some of the same files as Amy’s feature branch. At the commit marked (c), Amy has successfully merged her changes into the main branch. When Dee tries to push her changes (d), she will initially be prevented from doing so because additional commits have occurred in the origin repo since she last pulled (probably at point (a) when creating her branch). Dee must bring her copy of the repo up-to-date with respect to the origin before she can push her changes. One way to do this is for her to switch back to her main branch, then run `git pull origin main` to get the latest commits on main from the origin repo; her copy of the repo now looks the same as it did to Amy right after point (c).

Now Dee can try to merge her feature branch back into the main branch. If Dee’s branch changes different files than Amy’s branch, or if they change different parts of the same file that are far apart, the merge will succeed and Dee can then push her merged main branch back to the shared repo (`git push origin main`).

But if Dee and Amy had edited parts of the same file that were within a few lines of each other, as in Figure 10.5, Git will conclude that there is no safe way to automatically

- **git branch**
List existing branches in repo, indicating current branch with *. If you’re using sh or a bash-derived shell on a Unix-like system, placing the following in `~/.profile` will make the shell prompt display the current Git branch when you’re in a Git repo directory:
`https://gist.github.com/17f75e5697e5bca7a9ce2be75d012a65`

```
1 | export PS1="[\`git branch --no-color 2>/dev/null | \
2 |     sed -e '/\*/d' -e 's/* \(.*/\)\1/'`\% "
```
- **git checkout name**
Switch to existing branch *name*.
- **git branch name**
If branch *name* exists, switch to it; otherwise create a new branch called *name* without switching to it. The shortcut `git checkout -b name [commit-id]` creates and switches to a new branch based on *commit-id*, which defaults to most recent commit in the current branch.
- **git push [repo] [branch]**
Push the changes (commits) on *branch* to remote repository *repo*. (The first time you do this for a given branch, it creates that branch on the remote *repo*.) With no arguments, pushes the current local branch to the current branch’s remote, or the remote called `origin` by default.
- **git pull [repo] [branch]**
Fetches and merges commits from branch *branch* on the remote *repo* into your local repo’s **current** branch (*even if the current branch’s name doesn’t match the branch name you’re pulling from—beware!*). To fetch a remote branch *foo* for which you have no corresponding local branch, first use `git checkout -b foo` to create a local branch by that name and switch to it, then `git pull origin foo`. With no arguments, *repo* and *branch* default to the values of `git config branch.currentbranch.remote` and `git config branch.currentbranch.merge` respectively, which are automatically set up by certain Git commands and can be changed with `git branch --track`. If you setup a new repo in the usual way, *repo* defaults to `origin` and *branch* defaults to `main`.
- **git remote show [repo]**
If *repo* omitted, show a list of existing remote repos. If *repo* is the name of an existing remote repo, shows branches located at *repo* and which of your local branches are set up to track them. Also shows which local branches are not up-to-date with respect to *repo*.
- **git merge branch**
Merge all changes from *branch* into the current branch.
- **git rebase source-branch**
Try to rewrite history as if the current branch had originated from the latest commit on *source-branch*. You can also specify a specific commit-ID instead of the name of a source branch. Useful in pull requests since it shifts the work of conflict resolution from the target branch’s maintainer to the feature branch’s maintainer.
- **git cherry-pick commits**
Rather than merging all changes (commits) from a given branch, apply *only* the changes introduced by each of the named *commits* to the current branch.
- **git checkout branch file1 file2...**
For each file, merge the differences in *branch*’s version of that file into the current branch’s version of that file.

Figure 10.4: Common Git commands for handling branches and merging. Branch management involves merging;
Figure 10.6 tells how to undo merges gone awry.

<https://gist.github.com/708b2be6cbc71f1f3d499e683a4474fb>

```

1 | Roses are red,
2 | Violets are blue.
3 | <<<<< HEAD:poem.txt
4 | I love GitHub,
5 | =====
6 | ProjectLocker rocks,
7 | >>>>> 77976da35a1db4580b80ae27e8d65caf5208086:poem.txt
8 | and so do you.

```

Figure 10.5: When Bob tries to merge Amy’s changes, Git inserts conflict markers in `poem.txt` to show a merge conflict. Line 3 marks the beginning of the conflicted region with `<<<`; everything until `==` (line 5) shows the contents of the file in `HEAD` (the latest commit in Bob’s local repo) and everything thereafter until the end-of-conflict marker `>>>`(line 7) shows Amy’s changes (the file as it appears in Amy’s conflicting commit, whose commit-ID is on line 7). Lines 1,2 and 8 were either unaffected or merged automatically by Git.

create a version of the file that reflects both sets of changes, and it will leave a conflicted *and uncommitted* version of the file with conflict markers (`<<<` and `>>>`) in Dee’s main branch. Dee must now manually edit that file and add and commit the manually edited version to complete the merge, after which she can push the merged main back to the shared repo. In the next section, we will discuss an alternative process for preventing merge conflicts before they occur. If a merge goes badly awry, Figure 10.6 provides some mechanisms for partially or fully undoing the results of the merge. Figure 10.7 lists some useful Git commands to help keep track of who did what and when. Figure 10.8 shows some convenient notational alternatives to the cumbersome 40-digit Git commit-IDs.

Finally, don’t overlook the importance of a *scratch branch*, which is a branch that is never intended to be merged back into the mainline code. You can create a scratch branch to explore code changes such as exploring a spike (Section 7.4) or dry-running a radical change such as upgrading to a major new version of your app framework.

Whichever branches you create, if those branches are pushed to the main repo, over time the number of branches will grow. GitHub has a user interface for viewing and pruning stale (inactive) branches, which helps keep your codebase manageable.

Summary of branching:

- Small teams typically use a “shared-repo” model, in which pushes and pulls use a single authoritative copy of the repo. In Git, the authoritative copy is often referred to as the `origin` repo and is stored in the cloud on GitHub or on an internal company server.
- Branches allow variation in a code base. For example, feature branches support the development of new features without destabilizing working code, and release branches allow fixing bugs in previous releases whose code has diverged from the main line of development.
- Merging changes from one branch into another (for example, from a feature branch back into the main branch) may result in conflicts that must be manually resolved.
- With Agile and SaaS, feature branches are usually short-lived and release branches are uncommon.

Self-Check 10.2.1. *Describe a scenario in which merges could go in both directions—changes in a feature branch merged into the main branch, and changes in the main branch merged into a feature branch. (In Git, this is called a crisscross merge.)*

◊ Diana starts a new branch to work on a feature. Before she completes the feature, an important bug is fixed and the fix is merged into the main branch. Because the bug is in a part of the code that interacts with Diana’s feature, she merges the fix from main into her own feature branch. Finally, when she finishes the feature, her feature branch is merged back into main. ■

10.3 Pull Requests and Code Reviews

There is a really interesting group of people in the United States and around the world who do social coding now. The most interesting stuff is not what they do on Twitter, it's what they do on GitHub.

—Al Gore, former US Vice President, 2013

Section 10.7 describes the use of design reviews or code reviews to improve quality of the software product. You may be surprised to learn that most companies using Agile methods do not perform formal design or code reviews. But perhaps more surprising is that experienced Agile companies’ code is *better and more frequently* reviewed compared to companies that do formal code reviews.

For the explanation of this paradox, recall the basic idea of extreme programming: every good programming practice is taken to an extreme. Section 2.2 already described one form of “extreme code review” in the form of pair programming, in which the navigator continuously reviews the code being entered by the driver. In this section we describe another form of code review, in which the rest of the team has frequent opportunities to review the work of their colleagues. Our description follows the process used at GitHub, where formal code reviews are rare.

When a developer (or a pair) has finished work on a branch, rather than directly merging the topic branch into the main branch, the developer makes a **pull request**, or PR for short, asking that the branch’s changes be merged into (usually) the main branch. All developers sharing the repo see each PR, and each has the responsibility to determine how merging those changes might affect their own code. If anyone has a concern, an online discussion coalesces around the PR. For example, GitHub’s user interface allows any developer to make comments either on the PR overall or on specific lines of particular files modified by the PR. This discussion might result in changes to the code in question before the PR is accepted and merged; any further changes made on the PR’s topic branch and pushed to GitHub will automatically be reflected in the PR, so the PR process is really a form of code review. And since many PRs typically occur each day, these “mini-reviews” are occurring continuously, so there is no need for special meetings.

Merge request is an alternative name for pull request.

The PR therefore serves as a way to focus a code review discussion on a particular set of changes. That said, a PR shouldn’t be opened until the developer is confident their code is ready. Such preparation includes the following:

- The PR’s “description” field should provide a well-written explanation of what the proposed code does overall. For example, since PRs are often in support of a particular user story, the PR description could indicate which parts of the necessary functionality are covered by the proposed code.

- The code to be merged should be well covered by tests, all of which should be passing. (CHIPS 10.5 describes one way to configure GitHub so that every push to a topic branch automatically triggers a run of the test suite.)
- The commit messages should clearly indicate what was changed in each commit. (Since a PR is a request to merge one branch into another, it will often be the case that the branch being merged has had several commits.)
- Documentation (design documents, the README file, the project wiki, and so on) has been updated if necessary to explain new design decisions or changes to important configuration files (such as the `Gemfile` for Ruby projects).
- Any temporary or non-essential files that were versioned during development of the code have been removed from version control.
- Steps have been taken to eliminate or minimize merge conflicts that will occur when the PR is accepted and merged.

The last item above can be tricky, because as the previous section explained, it's not uncommon for a merge to encounter conflicts that must be manually resolved. Indeed, when you open a PR, GitHub checks and informs you whether the merge would require manual conflict resolution. Such a scenario motivates the possible use of *rebasing*, an operation in which you tell Git to make the world look as if you had branched from a later commit. For example, if there have been 3 new commits on main since the time you branched off of it, and you now rebase your branch on top of main, Git will *first* apply those 3 new commits to the original state of your branch, and *then* try to apply your own commits. This latter step may cause conflicts if the 3 commits on the main branch touched some of the same files your changes have touched. If so, Git generally requires you to resolve each conflict as it is detected, before proceeding with the rebase, and allows you to abort the rebase entirely if things get too ugly. But once you have resolved those conflicts, merging your branch back to main is guaranteed not to cause any new conflicts (unless, of course, additional commits to main happened while you were rebasing). In other words, from the point of view of trying to merge changes into a shared main branch, rebasing before merging forces you to resolve the conflicts at rebasing time, thus saving work for whoever will merge your branch back into the main branch.

There is an important caveat to rebasing. By its very nature, rebasing rewrites history, by making the world look as if your branch had been created from a different commit than it actually was, and by rewriting the commit history of the branch itself. This rewriting of history can occur in one of three scenarios:

Commit squashing is an optional rebasing step in which the branch's commits are “squashed” into a single commit that can be easily backed out to undo the effects of merging the PR.

1. You have not yet pushed to the shared repo. The history of your branch exists only in your copy of the repo, so rewriting that history does not affect the team.
2. You have pushed to the shared repo, but no one else has made additional changes based on your branch. This is the common case when using branch-per-feature, since there is normally no reason for one developer to base work on another developer's commits in a feature branch. You may need to use the `--force` flag to `git push` when pushing the branch, to indicate your acknowledgment that you're changing the shared repo's view of history.

3. You have pushed to the shared repo, and others have based work off of your changes. Anyone who has based their work off of your branch commits will now be out of sync since their history doesn't match the shared repo's history.

Case 3 is rare, but if it occurs, you should coordinate carefully with your team *before* forcing a push to avoid others' repos getting out of sync with the shared copy. One good practice is to construct feature branch names in some way that signals that others should not build off of those commits, for example by prepending the developer's initials to the branch name or using a standard naming convention such as `feature-xxx` for feature branch names.

An alternative to rebasing is merging: running `git pull origin main` in your feature branch at any time will update your clone from the origin repo, then merge any new changes from the main branch into your feature branch. Compared with rebasing, this approach is nondestructive because it doesn't rewrite history, but it also adds a lot of extra commits (the merge commits) to your feature branch, which can make it tricky to reconstruct the history of the feature branch using `git log`. Atlassian has an excellent set of tutorials⁴ covering this and many other Git-related topics.

All this having been said, if you're breaking down your user stories into tasks of manageable size (Section 7.4) and doing frequent deployments (Section 12.4), messy merges and rebases should rarely be necessary in Agile development.

Once the PR has been opened, one or more team members should review the PR. GitHub's user interface allows commenting on the PR as a whole as well as on specific changed lines of code, and the "split view" feature makes it easy to see which specific lines were changed in each file.

Each reviewer—there should be more than one, or reviewers can work as a pair—should *first* read the description to understand what the code changes are intended to do. If the description is unclear, the developer who opened the PR should amend it.

Once the description is clearly understood, it is helpful to next review the tests, since good tests also serve as documentation of how the code is supposed to behave. Tests are assumed to be passing, or the original developer would not have opened the PR for review.

Finally, the reviewers examine the new or changed code, and (politely) ask for additional explanation wherever needed to understand why the code is written as it is. Frequently, the developer who opened the PR will make additional commits to address reviewers' comments; the PR will remain open and the new commits will be added to it, and this feedback cycle can continue until the PR is either approved (merged into the main branch) or closed without merging.

-
- `git reset --hard ORIG_HEAD`
Revert your repo to last committed state just before the merge.
 - `git reset --hard HEAD`
Revert your repo to last committed state.
 - `git checkout commit -- [file]`
Restore a file, or if omitted the whole repo, to its state at *commit* (see Figure 10.8 for ways to refer to a commit besides its 40-digit SHA-1 hash). Can be used to recover files that were previously deleted using `git rm`.
 - `git revert commit`
Reverts the changes introduced by *commit*. If that commit was the result of a merge, effectively undoes the merge, and leaves the current branch in the state it was in before the merge. Git tries to back out just the changes introduced by that commit without disturbing other changes since that commit, but if the commit happened a long time ago, manual conflict resolution may be required.
-

Figure 10.6: When a merge goes awry, these commands can help you recover by undoing all or part of the merge.

<code>git blame [file]</code>	Annotate each line of a file to show who changed it last and when.
<code>git diff [file]</code>	Show differences between current working version of <i>file</i> and last committed version.
<code>git diff branch [file]</code>	Show differences between current version of <i>file</i> and the way it appears in the most recent commit on <i>branch</i> (see Section 10.2).
<code>git log [ref..ref] [files]</code>	Show log entries affecting all <i>files</i> between the two commits specified by the <i>refs</i> (which must be separated by exactly two dots), or if omitted, entire log history affecting those files.
<code>git log --since="date" files</code>	Show the log entries affecting all <i>files</i> since the given date (examples: "25-Dec-2019", "2 weeks ago").

Figure 10.7: Git commands to help track who changed what file and when. Many commands accept the option `--oneline` to produce a compact representation of their reports. If an optional *[file]* argument is omitted, default is “all tracked files.” Note that all these commands have *many more options*, which you can see with `git help command`.

<code>HEAD</code>	The most recently committed version on the current branch.
<code>HEAD~</code>	The prior commit on the current branch (<code>HEAD~n</code> refers to the <i>n</i> 'th previous commit).
<code>ORIG_HEAD</code>	When a merge is performed, <code>HEAD</code> is updated to the newly-merged version, and <code>ORIG_HEAD</code> refers to the commit state before the merge. Useful if you want to use <code>git diff</code> to see how each file changed as a result of the merge.
<code>1dfb2c~2</code>	2 commits prior to the commit whose ID has <code>1dfb2c</code> as a unique prefix.
<code>"branch@{date}"</code>	The last commit prior to <i>date</i> (see Figure 10.7 for date format) on <i>branch</i> , where <code>HEAD</code> refers to the current branch.

Figure 10.8: Convenient ways to refer to certain commits in Git commands, rather than using a full 40-digit commit ID or a unique prefix of one. `git rev-parse expr` resolves any of the above expressions into a full commit ID.

Summary of merge management for small teams:

1. By opening a pull request to merge a feature branch rather than performing the merge directly, the rest of the team is notified of the proposed changes and has a chance to do an on-the-spot code review around the pull request before it is accepted.
2. Rebasing rewrites history by “rewinding” a branch to originate from a different commit than it actually does, and then trying to apply the branch’s commits, thereby forcing the branch maintainer to resolve conflicts that *would* result if the un-rebased branch were merged. A common use for rebasing is to allow subsequent creation of a pull request that is guaranteed to be free of merge conflicts.
3. Because it rewrites history after the fact, rebasing should only be used when you can be sure that no one else has based their additional work on that branch’s commits.

■ Elaboration: When to open a pull request

Our advice above has been to open a PR when you’re confident that the code is ready for review, but some teams instead open a PR much earlier as a “draft PR,” as the code is in progress. The requester knows the PR won’t be merged, but this way the rest of the team can comment on the code as it evolves, rather than waiting until it’s fully ready. The PR can remain open as the code evolves in response to comments. Once the PR is judged ready for final review prior to merge, the “draft” designation is removed. The “early draft PR” approach is consistent with the XP philosophy: if code reviews are good, do them as early as possible and on fine-grained evolution of the code. The disadvantage is that it may create additional work for other team members compared to the simpler “Please look over my code now” approach, especially if the developer opening the PR is experienced and unlikely to benefit from very-early-stage code review.

Self-Check 10.3.1. *True or false: If you attempt `git push` and it fails with a message such as “Non-fast-forward (error): failed to push some refs,” this means some file contains a merge conflict between your repo’s version and the origin repo’s version.*

◊ Not necessarily. It just means that your copy of the repo is missing some commits that are present in the origin copy, and until you merge in those missing commits, you won’t be allowed to push your own commits. Merging in these missing commits *may* lead to a merge conflict, but frequently does not. ■

10.4 Delivering the Backlog Using Continuous Integration

As Section 7.4 explained, the **backlog** is the somewhat pessimistic term for the work remaining to be done during the current Agile iteration. That section described how to prioritize and estimate the difficulty of the iteration’s planned work, and briefly introduced Pivotal Tracker as a way to track the work. While there is no single “correct” workflow for Agile teams, in this section we describe a widely-used workflow and suggest best practices for using it effectively. We assume that the stories have already been prioritized and assigned points during an Iteration Planning Meeting, as Section 7.4 described.

The key idea behind delivering the backlog is **continuous integration** (CI), which minimizes the time between when changes are made on a feature branch and when those changes

are merged into the main branch and deployed for customer review. A good CI workflow for Agile two-pizza teams starts with a shared team repo and the use of pull requests to integrate changes from feature branches into the main branch. We introduce two new concepts here that are central to CI. The first is that of a service such as Travis⁵, whose job is to run your complete test suite, usually in the cloud, each time significant changes are made during development of a new feature. The rationale is that while you’re continuously testing the code for the feature you’re developing, you may not be taking the time to run the full test suite, which can take minutes or even tens of minutes for large projects. Somewhat confusingly, such services are usually called CI services, even though technically CI refers not just to running the test suite but to the entire workflow by which changes are integrated into mainline code. Most CI services can be connected directly to a GitHub repository and automatically trigger a CI run every time code is pushed to any branch in that repository.

The second concept is that of a staging server, which is configured as similarly as possible to the production server but usually much smaller in scale. The purpose of a staging server is to provide a safe place to deploy new features for customer review before they are deployed in production. The staging server may not even be a specific persistent server like the production server, but an ephemeral one “spun up” just to give the customer the opportunity to see a specific feature in action. A staging server has its own copy of the database containing test data (possibly extracted from real customer data), and is usually off-limits to outside users.

In this workflow, we distinguish two copies of a repo, each of whose main branches is represented by a thick horizontal line in Figure 10.9. Initially, we will describe the workflow from the point of view of the *origin* repo, which is owned by some team member and shared in the cloud by the team, and some developer’s local *clone* of the origin. (We will use “developer” to mean either an individual team member or a pair working on a story.) The local repo is created by running `git clone` with the URL of the origin repo. From the point of view of the local repo, the origin repo is one of possibly several *remotes*, and usually the default remote when there is more than one. The following numbered steps are keyed to the numbers in the figure, and annotated to indicate the associated state of the story in Pivotal Tracker.

1. On the main branch of local, the developer uses `git pull origin main` to ensure she has the most up-to-date version of main.
2. The developer creates a new feature branch for the story (Section 10.2). At this point the story state changes from Unstarted to Started.
3. The developer writes tests and code for the feature (Chapters 7 and 8), committing frequently. Periodically pushing the feature branch’s commits to the origin repo (`git push origin feature-branch`) keeps a copy of the feature branch on the origin repo up-to-date with the one on the local feature branch.
4. This team has their workflow configured so that any push to the origin repo will automatically trigger an external CI run on whatever branch was pushed.
5. When the code and tests are ready, the developer marks the story Finished, and opens a pull request. Note that the PR relates the topic branch *on the origin repo* to the main branch *also on the origin repo*, that is, the developer must have pushed the most recent commits on her local topic branch to its counterpart on origin.



A full CI run may include compilation (for compiled languages) and running operational tests (Chapter 12) beyond the scope of the specific feature being developed.

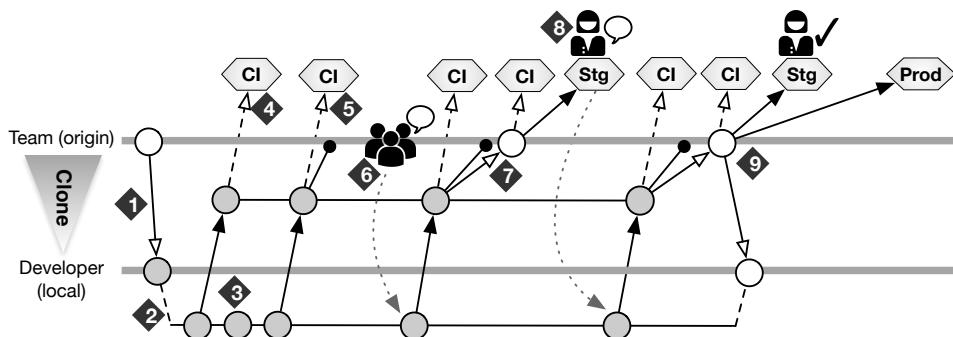


Figure 10.9: A basic workflow for a coordinated team delivering features when the team itself “owns” the app being developed. The boxes along the top indicate suggested interpretation of the Pivotal Tracker story states (started, finished, delivered, accepted) relative to events in the workflow.

6. Other team members review and comment on the PR (Section 10.3), in this case leading to required changes by the developer, who makes the changes and reopens the PR (or opens a new one).
7. The revised PR is accepted and the changes are merged into the origin repo’s main branch, triggering another CI run, since the main branch now includes not only this PR but possibly other developers’ PRs that have been merged since Step 1.
8. Assuming CI passes, the origin’s main branch is deployed to a staging server and the story is marked Delivered. The customer can now review and comment on the new feature. If the customer requests revisions, another round of changes, PR, and merging follows.

With the above workflow, in a team with several developers (or pairs) many stories and feature branches may be “in flight” at the same time.

What happens next depends on which repo is the *base repo* for the app, that is, the definitive repo from which the production app is released and which is stewarded by the app’s owners. If the development team in question owns the app, then the team’s origin repo may be the app’s base repo as well. But if the app is owned by other developers, then *their* repo is the base repo, and this team’s origin repo is just used for development. In that case, a pull request can be opened from the origin’s main branch to the base repo’s main branch to merge the changes, and goes through the usual process of review.

Figure 10.10 shows this “fork-and-pull” collaboration model with the upstream or base repo shown as the top line. The subteam’s origin repo, sometimes also called the *head repo*, is a fork of the base repo and opens PRs to merge head repo changes into the base repo. Just as feature branch developers may periodically rebase (Section 10.3) against their origin repo’s main branch to get the latest changes and avoid later merge conflicts, the head repo may pull changes from the base repo for the same reason, but because of the way Git works, these changes cannot propagate directly from the base repo on GitHub to its fork. Instead, some developer must pull the changes into their own local copy of the repo, then push the changes to the appropriate branch of the origin repo. The official GitHub tutorial on forking⁶ gives detailed steps for keeping a fork up-to-date with an upstream repo, though in your authors’ opinion this alternative tutorial⁷ is both clearer and more concise.

Fork historically meant a schism in which one team makes a copy of another team’s source code and starts developing it independently, often against the wishes of the original authors. GitHub overloaded the term to mean “creating your own copy of a repo to which you want to contribute but lack push access.”

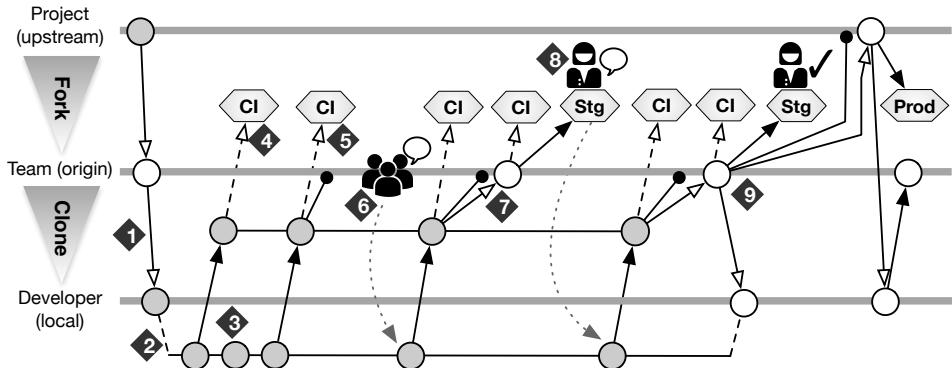


Figure 10.10: Variation of Figure 10.9 when the team does not own the app being developed. The final steps now involve coordinating with the upstream repo's stewards to get the changes merged there, and then update the team's origin repo (via a two-step process, as the text describes) to get the latest upstream changes.

These workflows should make even more clear why Section 7.4 recommends keeping stories simple: A 1-point story is one whose implementation strategy is mostly known to the team, so it can be delivered quickly and predictably, and if mistakes are made, they can be easily undone with little time being wasted. As with all aspects of Agile, a short cycle with quick feedback is better than spending a lot of time making large changes that carry more uncertainty and risk. A small story also means a simple and short-lived branch, speeding up pull requests and often eliminating the need for rebasing.

Still, no matter how careful your team is, sometimes a pull request may need to be revised before it's accepted, or the customer may partially reject a feature leading to the branch being modified and the PR being reopened, as both figures show. Such scenarios are part and parcel of Agile development, but to keep your repo clean and your team sane, we recommend mitigating such "loops" by following these best practices for delivering the backlog:

1. *Follow your own advice.* The goal of the Iteration Planning Meeting is to determine points and priorities for this iteration. Those decisions should be respected during the iteration, but if they need to be revisited, that should be a team effort rather than a unilateral decision by one developer.
2. *Work on one story at a time.* One developer or pair should finish a story before starting stories that require other changes, unless they run into impediments that prevent further progress on delivering that story.
3. *A sustainable pace is more important than total points.* Rather than delivering 5 points all at once at the iteration's end, deliver 1 point per day for 5 days, giving the rest of the team (and the customer) the opportunity to review your contributions as they come in.

Summary of pull-request-based workflow using branch-per-feature:

1. Pick a story to implement and mark it "Started"
2. Create and switch to a new feature branch for the story in your local copy of the repo
3. Develop code and tests using BDD and TDD on the branch, committing frequently
4. When the branch is ready for review and all tests are passing, open a pull request

5. Based on team feedback on the pull request, continue making changes on the branch until all feedback has been addressed
6. Merge the branch into the main or main branch
7. Mark the story “Finished”
8. The story will be marked “Delivered” when it is ready for the customer to try out, and “Accepted” when the customer signs off

Summary of Delivering the Backlog via CI:

- Continuous integration is about frequently integrating new code and tests into the mainline. Each integration provides an opportunity to get feedback from the team via “mini code reviews” during pull requests, and from the customer via frequent deployment to a staging server.
- Central to CI is the continuous running of tests as code is developed, usually using a separate service such as Travis.
- When a subteam is working on features, they may fork the upstream repo and do their teamwork on their development repo, issuing a final pull request to the upstream repo when their own workflow is complete.
- CI relies heavily on automation. For example, workflows can be constructed that automatically trigger a testing run optionally followed by automatic deployment to a staging server when commits are pushed to a specific repo or branch.

■ Elaboration: Further Automating CI

You can configure your workflow so that CI not only runs automatically on every push, but if the push passes CI, it is automatically deployed to an ephemeral staging server for customer review. This way, both code reviews and customer reviews can occur before the formal merge of the PR integrates these changes into the main branch. Some companies such as Salesforce go even further: if any test fails in CI, the CI tool performs binary searches across recent commits to pinpoint which specific commit introduced a new bug, and automatically opens and assigns a bug report for the developer responsible for that commit (Hansma 2011).



Self-Check 10.4.1. *Can you think of a scenario in which it makes sense for the team to review a PR for a particular story, but it does not make sense to deploy the story to staging for the customer’s feedback?*

- ◊ Often, a customer-requested feature is broken down into many separate stories, some of which do not result in new functionality visible to the customer, such as adding a new model without yet creating any views for it. The customer’s feedback will be needed as soon as there is a way to interact with the feature, even if incomplete, but not before. ■

10.5 CHIPS: Agile Iterations



CHIPS 10.5: Agile Iterations

<https://github.com/saasbook/hw-agile-iterations>

Perform one or two complete iterations of the Agile lifecycle by planning and tracking features for a SaaS app, adding code and tests for the features, deploying the features to staging and production, and using tools to track test coverage and code quality throughout.

10.6 Reporting and Fixing Bugs: The Five R's

Inevitably, bugs happen. If you're lucky, they are found before the software is in production, but production bugs happen too. Everyone on the team must agree on processes for managing the phases of the bug's lifecycle:

1. **R**eporting a bug
2. **R**eproducing the problem, or else **R**eclassifying it as “not a bug” or “won’t be fixed”
3. Creating a **R**egression test that demonstrates the bug
4. **R**epairing the bug
5. **R**eleasing the repaired code

Any stakeholder may find and report a bug in server-side or client-side SaaS code. A member of the development or QA team must then reproduce the bug, documenting the environment and steps necessary to trigger it. This process may result in reclassifying the bug as “not a bug” for various reasons:

- This is not a bug but a request to make an enhancement or change a behavior that is working as designed
- This bug is in a part of the code that is being undeployed or is otherwise no longer supported
- This bug occurs only with an unsupported user environment, such as a very old browser lacking necessary features for this SaaS app
- This bug is already fixed in the latest version (uncommon in SaaS, whose users are always using the latest version)

Once the bug is confirmed as genuine and reproducible, it's entered into a bug management system. A plethora of such systems exists, but the needs of many small to medium teams can be met by a tool you're already using: Pivotal Tracker allows marking a story as a Bug rather than a Feature, which assigns the story zero points but otherwise allows it to be tracked to completion just like a regular user story. An advantage of this tool is that Tracker manages the bug's lifecycle for you, so existing processes for delivering user stories can be readily adapted to fixing bugs. For example, fixing the bug must be prioritized relative to other work; in a waterfall process, this may mean prioritization relative to other outstanding

Large projects for widely-used software may use considerably more complex bug tracking systems, such as the open-source Bugzilla⁸.

bugs while in the maintenance phase, but in an Agile process it usually means prioritization relative to developing new features from user stories. Using Tracker, the Product Manager can move the bug story above or below other stories based on the bug's severity and impact on the customer. For example, bugs that may cause data loss in production will get prioritized very high.

The next step is repair, which always begins with *first* creating the *simplest possible* automated test that fails in the presence of the bug, and *then* changing the code to make the test(s) pass green. This should sound familiar to you by now as a TDD practitioner, but this practice is true even in non-TDD environments: *no bug can be closed out without a test*. Depending on the bug, unit tests, functional tests, integration tests, or a combination of these may be required. *Simplest* means that the tests depend on as few preconditions as possible, tightly circumscribing the bug. For example, simplifying an RSpec unit test would mean minimizing the setup preceding the test action or in the `before` block, and simplifying a Cucumber scenario would mean minimizing the number of `Given` or `Background` steps. These tests usually get added to the regular regression suite to ensure the bug doesn't recur undetected. A complex bug may require multiple commits to fix; a common policy in BDD+TDD projects is that commits with failing or missing tests shouldn't be merged to the main development branch until the tests pass green.

Many bug tracking systems can automatically cross-reference bug reports with the commit-IDs that contain the associated fixes and regression tests. For example, using GitHub's service hooks⁹, a commit can be annotated with the story ID of the corresponding bug or feature in Tracker, and when that commit is pushed to GitHub, the story is automatically marked as Delivered. Depending on team protocol and the bug management system in use, the bug may be "closed out" either immediately by noting which release will contain the fix or after the release actually occurs.

As we will see in Chapter 12, in most Agile teams releases are very frequent, shortening the bug lifecycle.

"Severity 1" bugs at Amazon.com require the responsible engineers to initiate a conference call within 15 minutes of learning of the bug—a stricter responsiveness requirement than for on-call physicians! (Bodik et al. 2006)



Summary: the 5 R's of bug fixing

- A bug must be reported, reproduced, demonstrated in a regression test, and repaired, all before the bug fix can be released.
- No bug can be closed out without an automated test demonstrating that we really understand the bug's cause.
- Bugs that are really enhancement requests or occur only in obsolete versions of the code or in unsupported environments may be reclassified to indicate they're not going to be fixed.

Self-Check 10.6.1. Why do you think "bug fix" stories are worth zero points in Tracker even though they follow the same lifecycle as regular user stories?

- ◊ A team's velocity would be artificially inflated by fixing bugs, since they'd get points for implementing the feature in the first place and then more points for actually getting the implementation right. ■

Self-Check 10.6.2. True or false: a bug that is triggered by interacting with another service (for example, authentication via Twitter) cannot be captured in a regression test because the

necessary conditions would require us to control Twitter's behavior.

- ◊ False: integration-level mocking and stubbing, for example using the Webmock gem¹⁰ or the techniques described in Section 8.4, can almost always be used to mimic the external conditions necessary to reproduce the bug in an automated test. ■

Self-Check 10.6.3. *True or false: a bug in which the browser renders the wrong content or layout due to JavaScript problems might be reproducible manually by a human being, but it cannot be captured in an automated regression test.*

- ◊ False: tools such as Jasmine and Webdriver (Section 6.8) can be used to develop such tests.

■

10.7 The Plan-And-Document Perspective on Managing Teams

In Plan-And-Document processes, project management starts with the project manager. Project managers are the bosses of the projects:

- They write the contract proposal to win the project from the customer.
- They recruit the development team from existing employees and new hires.
- They typically write team members' performance reviews, which shape salary increases.
- From a Scrum perspective (Section 10.1), they act as Product Owner—the primary customer contact—and they act as Scrum Lead, as they are the interface to upper management and they procure resources for the team.
- As we saw in Section 7.9, project managers also estimate costs, make and maintain the schedule, and decide which risks to address and how to overcome or avoid them.
- As you would expect for Plan-And-Document processes, project managers must document their project management plan. Figure 10.11 gives an outline of Project Management Plans from the corresponding IEEE standard.

As a result of all these responsibilities, project managers receive much of the blame if projects have problems. Quoting a textbook author from his introduction to project management:

...if a post mortem were to be conducted for every [problematic] project, it is very likely that a consistent theme would be encountered: project management was weak.

—(Pressman 2010)

We cover four major tasks for project managers to increase their chances of being successful:

1. Team size, roles, space, communication
2. Managing people and conflicts
3. Inspections and metrics
4. Configuration management

<p>1. Project overview</p> <ul style="list-style-type: none"> 1.1 Project summary <ul style="list-style-type: none"> 1.1.1 Purpose, scope and objectives 1.1.2 Assumptions and constraints 1.1.3 Project deliverables 1.1.4 Schedule and budget summary 1.2 Evolution of the plan <p>2. References</p> <p>3. Definitions</p> <p>4. Project context</p> <ul style="list-style-type: none"> 4.1 Process model 4.2 Process improvement plan 4.3 Infrastructure plan 4.4 Methods, tools and techniques 4.5 Product acceptance plan 4.6 Project organization <ul style="list-style-type: none"> 4.6.1 External interfaces 4.6.2 Internal interfaces 4.6.3 Authorities and responsibilities <p>5. Project planning</p> <ul style="list-style-type: none"> 5.1 Project initiation <ul style="list-style-type: none"> 5.1.1 Estimation plan 5.1.2 Staffing plan 5.1.3 Resource acquisition plan 5.1.4 Project staff training plan 	<p>5.2 Project work plans</p> <ul style="list-style-type: none"> 5.2.1 Work activities 5.2.2 Schedule allocation 5.2.3 Resource allocation 5.2.4 Budget allocation 5.2.5 Procurement plan <p>6. Project assessment and control</p> <ul style="list-style-type: none"> 6.1 Requirements management plan 6.2 Scope change control plan 6.3 Schedule control plan 6.4 Budget control plan 6.5 Quality assurance plan 6.6 Subcontractor management plan 6.7 Project closeout plan <p>7. Product delivery</p> <p>8. Supporting process plans</p> <ul style="list-style-type: none"> 8.1 Project supervision and work environment 8.2 Decision management 8.3 Risk management 8.4 Configuration management 8.5 Information management <ul style="list-style-type: none"> 8.5.1 Documentation 8.5.2 Communication and publicity 8.6 Quality assurance 8.7 Measurement 8.8 Reviews and audits 8.9 Verification and validation
--	---

Figure 10.11: Format of a project management plan from the IEEE 16326-2009 ISO/IEC/IEEE Systems and Software Engineering—Life Cycle Processes—Project Management standard.

1. Team size, roles, space, and communication. The Plan-and-Document processes can scale to larger sizes, where group leaders report to the project manager. However, each subgroup typically stays the size of the two-pizza teams we saw in Section 10.1. Size recommendations are three to seven people (Braude and Bernstein 2011) to no more than ten (Sommerville 2010). Fred Brooks gave us the reason in Chapter 7: adding people to the team increases parallelism, but also increases the amount of time each person must spend communicating. These team sizes are reasonable considering the fraction of time spent communicating.

Given we know the size of the team, members of a subgroup in Plan-and-Document processes can be given different roles in which they are expected to lead. For example (Pressman 2010):

- Configuration management leader
- Quality assurance leader
- Requirements management leader
- Design leader
- Implementation leader

One surprising result is that the type of space for the team to work in affects project management. One study found that colocating the team in open space could double productivity (Teasley et al. 2000). The reasons include that team members had easy access to each other for both coordination of their work and for learning, and they could post their work artifacts on the walls so that all could see. Another study of teams in open space concludes:

One of the main drivers of success was the fact that the team members were at hand, ready to have a spontaneous meeting, advise on a problem, teach/learn something new, etc. We know from earlier work that the gains from being at hand drops off significantly when people are first out of sight, and then most severely when they are more than 30 meters apart.

—(Allen and Henn 2006)

While the team relies on email and texting for communicating and shares information in wikis and the like, there is also typically a weekly meeting to help coordinate the project. Recall that the goal is to minimize the time spent communicating unnecessarily, so it is important that the meetings be effective. Below is our digest of advice from the many guidelines found on the Web on how to have efficient meetings. We use the acronym SAMOSAS as a memory device; surely bringing a plate of them will make for an effective meeting!

Samosas are a popular stuffed deep-fried snack from India.



- Start and stop meeting on time.
- Agenda created in advance of meeting; if there is no agenda, then cancel the meeting.
- Minutes must be recorded so everyone can recall results afterwards; the first agenda item is finding a note taker.
- One speaker at a time; no interruptions when another is speaking.
- Send material in advance, since people read much faster than speakers talk.

- Action items at end of meeting, so people know what they should do as a result of the meeting.
- Set the date and time of the next meeting.

2. Managing people and conflicts. Thousands of books have been written on how to manage people, but the two most useful ones that we have found are *The One Minute Manager* and *How to Win Friends and Influence People* (Blanchard and Johnson 1982; Carnegie 1998). What we like about the first book is that it offers short quick advice. Be clear about the goals of what you want done and how well it should be done, but to encourage creativity, leave it up to the team member to decide how to do it. When meeting with individuals to review progress, start with positive feedback to help build their confidence. Then, be honest with them about what is not going well, and what they need to do to fix it. Finally, conclude with positive feedback and encouragement to continue improving their work. What we like about the second book is that it helps teach the art of persuasion, to get people to do what you think should be done without ordering them to do it. These skills also help persuade people you *cannot* command: your customers and your management.

Both books are helpful when it comes to resolving conflicts within a team. Conflicts are not necessarily bad, in that it can be better to have the conflict than to let the project crash and burn. Intel Corporation labels this attitude *constructive confrontation*. If you have a strong opinion that a person is proposing the wrong thing technically, you are obligated to bring it up, even to your bosses. The Intel culture is to speak up even if you disagree with the highest ranked people in the room.

If conflict continues, given that Plan-and-Document processes have a project manager, that person can make the final decision. One reason the US made it to the moon in the 1960s is that a leader of NASA, Wernher von Braun, had a knack for quickly resolving conflicts on close decisions. His view was that picking an option arbitrarily but quickly was frequently better, since the choice was roughly 50-50, so that the project could move ahead rather than take the time to carefully collect all the evidence to see which choice was slightly better.

However, once a decision is made, the teams needs to embrace it and move ahead. The Intel motto for this resolution is *disagree and commit*: “I disagree, but I am going to help even if I don’t agree.”

3. Inspections and metrics. Inspections like **design reviews** and **code reviews** allow feedback on the system even before everything is working. The idea is that once you have a design and initial implementation plan, you are ready for feedback from developers beyond your team. Design and code reviews follow the Waterfall lifecycle in that each phase is completed in sequence before going on to the next phase, or at least for the phases of a single iteration in Spiral or RUP development.

A design review is a meeting in which the authors of program present its design. The goal of the review is to improve software quality by benefiting from the experience of the people attending the meeting. A code review is held once the design has been implemented. This peer-oriented feedback also helps with knowledge exchange within the organization and offers coaching that can help the careers of the presenters.

Shalloway suggests that formal design and code reviews are often too late in the process to make a big impact on the result (Shalloway 2002). He recommends to instead have earlier, smaller meetings that he calls “approach reviews.” The idea is to have a few senior developers assist the team in coming up with an approach to solve the problem. The group brainstorms about different approaches to help find a good one.

If you plan to do a formal design review, Shalloway suggests that you first hold a “mini-design review” after the approach has been selected and the design is nearing completion. It involves the same people as before, but the purpose is to prepare for the formal review.

The formal review itself should start with a high-level description of what the customers want. Then give the architecture of the software, showing the APIs of the components. It will be important to highlight the design patterns used at different levels of abstraction (see Chapter 11). You should expect to explain *why* you made the decisions, and whether you considered plausible alternatives. Depending on the amount of time and the interests of those at the meeting, the final phase would be to go through the code of the implemented methods. At all these phases, you can get more value from the review if you have a concrete list of questions or issues that you would like to hear about.

One advantage of code reviews is that they encourage people outside your team to look at your comments as well as your code. As we don’t have a tool that can enforce the advice from Chapter 9 about making sure the comments raise the level of abstraction, the only enforcing mechanism is the code review.

In addition to reviewing the code and the comments, inspections can give feedback on every part of the project in Plan-and-Document processes: the project plan, schedule, requirements, testing plan, and so on. This feedback helps with **verification and validation** of the whole project, to ensure that it is on a good course. There is even an IEEE standard on how to document the verification and validation plan for the project, which Figure 10.12 shows.

Like the algorithmic models for cost estimation (see Section 7.9), some researchers have advocated that software metrics could replace inspections or reviews to assess project quality and progress. The idea is to collect metrics across many projects in an organization over time, establish a baseline for new projects, and then see how the project is doing compared to baseline. This quote captures the argument for metrics:

Without metrics, it is difficult to know how a project is executing and the quality level of the software.

—(Braude and Bernstein 2011)

Below are sample metrics that can be automatically collected:

- Code size, measured in thousands of lines of code (*KLOC*) or in function points (Section 7.9).
- Effort, measured in person-months spent on project.
- Project milestones planned versus fulfilled.
- Number of test cases completed.
- Defect discovery rate, measured in defects discovered (via testing) per month.
- Defect repair rate, measured in defects fixed per month.

Other metrics can be derived from these so as to normalize the numbers to help compare results from different projects: KLOC per person-month, defects per KLOC, and so on.

The problem with this approach is that there is little evidence of correlation between these metrics that we can automatically collect and project outcomes. Ideally, the metrics would

<ul style="list-style-type: none"> 1. Purpose 2. Referenced documents 3. Definitions 4. V&V overview <ul style="list-style-type: none"> 4.1 Organization 4.2 Top-level schedule 4.3 Integrity level scheme 4.4 Resources summary 4.5 Responsibilities 4.6 Tools, techniques, and methods 5. V&V processes <ul style="list-style-type: none"> 5.1 Common V&V Processes, Activities and Tasks 5.2 System V&V Processes, Activities and Tasks <ul style="list-style-type: none"> 5.2.1 Acquisition Support 5.2.2 Supply Planning 5.2.3 Project Planning 5.2.4 Configuration Management 5.2.5 Stakeholder Requirements Definition 5.2.6 Requirements Analysis 5.2.7 Architectural Design 5.2.8 Implementation 5.2.9 Integration 5.2.10 Transition 5.2.11 Operation 5.2.12 Maintenance 5.2.13 Disposal 5.3 Software V&V Processes, Activities and Tasks <ul style="list-style-type: none"> 5.3.1 Software Concept 5.3.2 Software Requirements 5.3.3 Software Design 5.3.4 Software Construction 5.3.5 Software Integration Test 5.3.6 Software Qualification Test 5.3.7 Software Acceptance Test 5.3.8 Software Installation and Checkout (Transition) 5.3.9 Software Operation 5.3.10 Software Maintenance 5.3.11 Software Disposal 	<ul style="list-style-type: none"> 5.4 Hardware V&V Processes, Activities and Tasks <ul style="list-style-type: none"> 5.4.1 Hardware Concept 5.4.2 Hardware Requirements 5.4.3 Hardware Design 5.4.4 Hardware Fabrication 5.4.5 Hardware Integration Test 5.4.6 Hardware Qualification Test 5.4.7 Hardware Acceptance Test 5.4.8 Hardware Transition 5.4.9 Hardware Operation 5.4.10 Hardware Maintenance 5.4.11 Hardware Disposal 6. V&V reporting requirements <ul style="list-style-type: none"> 6.1 Task reports 6.2 Anomaly reports 6.3 V&V final report 6.4 Special studies reports (optional) 6.5 Other reports (optional) 7. V&V administrative requirements <ul style="list-style-type: none"> 7.1 Anomaly resolution and reporting 7.2 Task iteration policy 7.3 Deviation policy 7.4 Control procedures 7.5 Standards, practices, and conventions 8. V&V test documentation requirements
---	--

Figure 10.12: Outline of a plan for System and Software Verification and Validation from the IEEE 1012-2012 Standard.

correlate and we could have much finer-grained understanding than comes from the occasional and time-consuming inspections. This quote captures the argument de-emphasizing metrics:

However, we are still quite a long way from this ideal situation, and there are no signs that automated quality assessment will become a reality in the foreseeable future.

—(Sommerville 2010)

4. Configuration management. Configuration management includes four varieties of changes, three of which we have seen before. The first is **version control**, sometimes also called *source and configuration management* (SCM), described in Sections 10.2–10.4. This variety keeps track of versions of components as they are changed. The second, *system building*, is closely related to the first. Tools like `make` assemble the compatible versions of components into an executable program for the target system. The third variety is **release management**, which we cover in Chapter 12. The last is **change management**, which comes from change requests made by customers and other stakeholders to fix bugs or to improve functionality (see Section 9.7).

As you surely expect by now, IEEE has a standard for Configuration Management. Figure 10.13 shows its table of contents.

Summary: In Plan-and-Document processes:

- Project managers are in charge: they write the contract, recruit the team, and interface with the customer and upper management.
- The project manager documents the project plan and configuration plan, along with the verification and validation plan that ensures that other plans are followed!
- To limit time spent communicating, groups are three to ten people. They can be composed into hierarchies to form larger teams reporting to the project manager, with each group having its own leader.
- Guidelines for managing people include giving them clear goals but empowering them, and starting with the positive feedback in reviews but being honest about shortcomings and how to overcome them.
- While conflicts need to be resolved, they can be helpful in finding the best path forward for a project.
- Inspections like design reviews and code reviews let outsiders give feedback on the current design and future plans. Such reviews allow the team to benefit from the experience of others. They are also a good way to check if good practices are being followed and if the plans and documents are sensible.
- Configuration management is a broad category that includes change management while maintaining a product, version control of software components, system building of a coherent working program from those components, and release management to ship the product to customers.

Table of Contents	
1.	Overview
1.1	Scope
1.2	Purpose
2.	Definitions, acronyms, and abbreviations
2.1	Definitions
2.2	Acronyms and abbreviations
3.	Tailoring
4.	Audience
5.	The configuration management process
6.	CM planning lower-level process
6.1	Purpose
6.2	Activities and tasks
7.	CM management lower-level process
7.1	Purpose
7.2	Activities and tasks
8.	Configuration identification lower-level process
8.1	Purpose
8.2	Activities and tasks
9.	Configuration change control lower-level process
9.1	Purpose
9.2	Activities and Tasks
10.	Configuration status accounting lower-level process
10.1	Purpose
10.2	Activities and tasks
11.	CM configuration auditing lower-level process
11.1	Purpose
11.2	Activities and Tasks
12.	Interface control lower-level process
12.1	Purpose
12.2	Activities and Tasks
13.	Supplier configuration item control lower-level process
13.1	Purpose
13.2	Activities and Tasks
14.	Release management lower-level process
14.1	Purpose
14.2	Activities and tasks

Figure 10.13: A table of contents for the IEEE 828-2012 Standard for Configuration Management in Systems and Software Engineering.

Self-Check 10.7.1. Compare the size of teams in Plan-and-Document processes versus Agile processes.

- ◊ Plan-and-Document processes can form hierarchies of subgroups to create a much larger project, but each subgroup is basically the same size as a “two-pizza” team for Agile. ■

Self-Check 10.7.2. True or False: Design reviews are meetings intended to improve the quality of the software product using the wisdom of the attendees, but they also result in technical information exchange and can be highly educational for junior members of the organization, whether presenters or just attendees.

- ◊ True. ■

10.8 Fallacies and Pitfalls



Fallacy: If a software project is falling behind schedule, you can catch up by adding more people to the project.

The main theme of Fred Brooks’s classic book, *The Mythical Man-Month*, is that not only does adding people not help, it makes it worse. The reason is twofold: it takes a while for new people to learn about the project, and as the size of the team grows, the amount of communication increases, which can reduce the time available for people to get their work done. His summary, which some call Brooks’s Law, is:

Adding manpower to a late software project makes it later.

—Fred Brooks, Jr.



Pitfall: Dividing work based on the software stack rather than on features.

It’s less common than it used to be to divide the team into a front-end specialist, back-end specialist, customer liaison, and so forth, but it still happens. Your authors and others believe that better results come from having each team member deliver *all* aspects of a chosen feature or story—Cucumber scenarios, RSpec tests, views, controller actions, model logic, and so on. Especially when combined with pair programming, having each developer maintain a “full stack” view of the product spreads architectural knowledge around the team.



Fallacy: It’s fine to make simple changes on the main branch.

Programmers are optimists. When we set out to change our code, we always think it will be a one-line change. Then it turns into a five-line change; then we realize the change affects another file, which has to be changed as well; then it turns out we need to add tests or change existing tests that relied on the old code; and so on. For this reason, *always* create a feature branch when starting new work. Branching with Git is nearly instantaneous, and if the change truly does turn out to be small, you can delete the branch after merging to avoid having it clutter your branch namespace.



Pitfall: Forgetting to add files to the repo.

If you create a new file but forget to add it to the repo, *your* copy of the code will still work but when others pull your changes your code won’t work for them. Use `git status`

regularly to see the list of Untracked Files, and use the `.gitignore` file to avoid being warned about files you never want to track, such as binary files or temporary files.



Pitfall: Versioning files that shouldn't be versioned.

If a file isn't required to run the code, it probably shouldn't be in the repo: temporary files, binary files, log files, and so on should not be versioned. If files of test data are versioned, they should be part of a proper test suite. Files containing sensitive information such as API keys should *never* be checked into GitHub in plaintext (i.e. without encryption). If the files must be checked in, they should be encrypted.



Pitfall: Accidentally stomping on changes after merging or switching branches.

If you do a pull or a merge, or if you switch to a different branch, some files may suddenly have different contents on disk. If any such files are already loaded into your editor, the versions being edited will be *out of date*, and even worse, if you now save those files, you will either overwrite merged changes or save a file that isn't in the branch you think it is. The solution is simple: *before* you pull, merge or switch branches, make sure you commit all current changes; *after* you pull, merge or switch branches, reload any files in your editor that may be affected—or to be really safe, just quit your editor before you commit. Be careful too about the potentially destructive behavior of certain Git commands such as `git reset`.



Pitfall: Letting your copy of the repo get too far out of sync with the origin (authoritative) copy.

It's best not to let your copy of the repo diverge too far from the origin, or merges (Section 10.2) will be painful. You should update frequently from the origin repo before starting work, and if necessary, rebase incrementally so you don't drift too far away from the main branch.



Fallacy: Since each subteam is working on its own branch, we don't need to communicate regularly or merge frequently.

Branches are a great way for different team members to work on different features simultaneously, but without frequent merges and clear communication of who's working on what, you risk an increased likelihood of merge conflicts and accidental loss of work when one developer "resolves" a merge conflict by deleting another developer's changes.



Pitfall: Making commits too large.

Git makes it quick and easy to do a commit, so you should do them frequently and make each one small, so that if some commit introduces a problem, you don't have to also undo all the other changes. For example, if you modified two files to work on feature A and three other files to work on feature B, do two separate commits in case one set of changes needs to be undone later. In fact, advanced Git users use `git add` with specific files, rather than `git add .` which adds every file in the current directory, to "cherry pick" a subset of changed files to include in a commit. And don't forget that no one else will see the commit until you use `git push` to propagate them to the team's origin repo.

10.9 Concluding Remarks: From Solo Developer to Teams of Teams

The first 90% of the code accounts for the first 10% of the development time. The remaining 10% of the code accounts for the other 90% of the development time.

—Tom Cargill, quoted in *Programming Pearls*, 1985

The history of version control systems mirrors the movement towards distributed collaboration among “teams of teams,” with two-pizza teams emerging as a popular unit of cohesiveness. From about 1970–1985, the original Unix **Source Code Control System** (SCCS) and its longer-lived descendant **Revision Control System** (RCS) required the repo and all development to stay on the same computer (which might be a multi-user system) and disallowed simultaneous editing of the same file by different developers. The **Concurrent Versions System** (CVS) and **Subversion** introduced simultaneous editing and branches, but only a single repo. Git completed the decentralization by allowing any copy of a repo to push or pull from any other, enabling completely decentralized “teams of teams,” and by making branching and merging much quicker and easier than its predecessors. Today, distributed collaboration is the norm: rather than a large distributed team, fork-and-pull allows a large number of Agile two-pizza teams to make independent progress, and the use of Git to support such efforts has become ubiquitous. The two-pizza team size makes it easier for a team to stay organized than the giant programming teams possible in Plan-and-Document. The decentralized approach also distributes responsibility for project planning and cost estimation more than P&D, which relies on the project manager to make the time and cost estimates, assess risks, and to run the project so that it delivers the product on time and on budget with the required functionality.

We have previously seen two examples of processes in which the P&D and Agile versions comprise the same skills and steps, but sequenced differently. Test-driven development uses the same elements as conventional code writing followed by debugging, but in a different order. Agile iterations include the same elements as a waterfall project, but in a different order. Team coordination is a third example: Agile teams use the same processes as P&D teams—releases, code reviews, customer reviews, cost and effort estimation, assignment of different parts of the coding task to different developers—but in a different order. Agile proponents believe the techniques in this chapter can help an agile team avoid many of the pitfalls that have made software projects infamous for being late and over budget. Checking in continuously with other developers (via PRs) and customers (via frequent deployments to staging) during each iteration guides your team into spending its resources most effectively and is more likely to result in software that makes customers happy within the time and cost budget. A disciplined workflow using version control allows developers to make progress on many fronts simultaneously without interfering with each others’ work, and also allows disciplined and systematic management of the bug lifecycle.

Finally, as with any experience, you should reflect on what went well, what didn’t go well, and what you would do differently. It is not a sin to make a mistake, as long as you learn from it; the sin is making the same mistake repeatedly. Many Agile teams’ end-of-iteration Retrospective meeting allows this learning to happen incrementally each week and makes the team more cohesive over time, rather than waiting until the end of a long project to determine what could have gone better.

For more comprehensive details on this chapter’s topics, we recommend these resources:

- You can find very detailed descriptions of Git’s powerful features in *Version Control*

With Git (Loeliger 2009), which takes a more tutorial approach, and in the free Git Community Book¹¹, which is also useful as a thorough reference on Git. For detailed help on a specific command, use `git help command`, for example, `git help branch`; but be aware that these explanations are for reference, not tutorial.

- Atlassian has an excellent set of tutorials¹² covering many Git-related topics, including rebasing.
- Many medium-sized projects that don't use Pivotal Tracker, or whose bug-management needs go somewhat beyond what Tracker provides, rely on the Issues feature built into every GitHub repo. The Issues system allows each team to create appropriate "labels" for different bug types and priorities and create their own "bug lifecycle" process.

T. J. Allen and G. Henn. *The Organization and Architecture of Innovation: Managing the Flow of Technology*. Butterworth-Heinemann, 2006.

K. H. Blanchard and S. Johnson. *The One Minute Manager*. William Morrow, Cambridge, MA, 1982.

P. Bodík, A. Fox, M. I. Jordan, D. Patterson, A. Banerjee, R. Jagannathan, T. Su, S. Teng-inakai, B. Turner, and J. Ingalls. Advanced tools for operators at Amazon.com. In *First Workshop on Hot Topics in Autonomic Computing (HotAC'06)*, Dublin, Ireland, June 2006.

E. Braude and M. Bernstein. *Software Engineering: Modern Approaches, Second Edition*. John Wiley and Sons, 2011. ISBN 9780471692089.

D. Carnegie. *How to Win Friends and Influence People*. Pocket, 1998.

S. Hansma. Go fast and don't break things: Ensuring quality in the cloud. In *Workshop on High Performance Transaction Systems (HPTS 2011)*, Asilomar, CA, Oct 2011. Summarized in Conference Reports column of USENIX ;login 37(1), February 2012.

D. Holland. *Red Zone Management*. WinHope Press, 2004. ISBN 0967140188.

J. Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, 2009. ISBN 0596520123.

R. Pressman. *Software Engineering: A Practitioner's Approach, Seventh Edition*. McGraw-Hill, 2010. ISBN 0073375977.

K. Schwaber and M. Beedle. *Agile Software Development with Scrum (Series in Agile Software Development)*. Prentice Hall, 2001. ISBN 0130676349.

A. Shalloway. *Agile Design and Code Reviews*. 2002. URL <http://www.netobjectives.com/download/designreviews.pdf>.

I. Sommerville. *Software Engineering, Ninth Edition*. Addison-Wesley, 2010. ISBN 0137035152.

S. Teasley, L. Covi, M. S.Krishnan, and J. S. Olson. How does radical collocation help a team succeed? In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 339–346, Philadelphia, Pennsylvania, December 2000.

Notes

¹<https://sp19.datastructur.es/materials/guides/using-git.html>
²<https://rework.withgoogle.com/guides/understanding-team-effectiveness/steps/introduction/>
³<https://github.com/bbatsov/rails-style-guide>
⁴<https://www.atlassian.com/git/tutorials>
⁵<http://travis-ci.org>
⁶<https://help.github.com/en/github/getting-started-with-github/fork-a-repo>
⁷<https://gist.github.com/Chaser324/ce0505fbed06b947d962>
⁸<http://mozilla.org>
⁹<http://github.com/>
¹⁰<https://github.com/bblimke/webmock>
¹¹<http://book.git-scm.com>
¹²<https://www.atlassian.com/git/tutorials>

11

Design Patterns for SaaS Apps

William Kahan (1933–) received the 1989 Turing Award for his fundamental contributions to numerical analysis. Kahan dedicated himself to “making the world safe for numerical computations.”



Things are genuinely simple when you can think correctly about what's going on without having a lot of extraneous or confusing thoughts to impede you. Think of Einstein's maxim, “Everything should be made as simple as possible, but no simpler.”

—“A Conversation with William Kahan,” *Dr. Dobbs’ Journal*, 1997

11.1 Patterns, Antipatterns, and SOLID Class Architecture	338
11.2 Just Enough UML	342
11.3 Single Responsibility Principle	345
11.4 Open/Closed Principle	347
11.5 Liskov Substitution Principle	351
11.6 Dependency Injection Principle	354
11.7 Demeter Principle	358
11.8 The Plan-And-Document Perspective on Design Patterns	362
11.9 6S: A Clean Code Checklist	363
11.10 Fallacies and Pitfalls	365
11.11 Concluding Remarks: Frameworks Capture Design Patterns	366

Prerequisites and Concepts

The big concept of this chapter is that *design patterns* can improve the quality of the classes. A design pattern captures proven solutions to problems by separating the things that change from those that don't.

Concepts:

Five object-oriented design principles identified by the acronym SOLID describe sound design of interactions among classes. An *antipattern* indicates poor class design, which a *design smell* can identify. Thus, using design smells to detect violations to the *SOLID principles* for good class design is analogous to using *code smells* to detect violations of the *SOFA principles* for good method design (Section 9.5).

The five letters of the SOLID acronym stand for:

1. *Single Responsibility Principle*: a class should have one and only one responsibility; that is, only one reason to change. The *Lack of Cohesion Of Methods* metric indicates the antipattern of too large a class.
2. *Open/Closed Principle*: a class should be open for extension, but closed against modification. The *Case Statement* design smell suggests a violation.
3. *Liskov Substitution Principle*: a method designed to work on an object of type T should also work on an object of any subtype of T. That is, all of T's subtypes should preserve T's "contract." The *refused bequest* design smell often indicates a violation.
4. *Dependency Injection Principle*: if two classes depend on each other but their implementations may change, it would be better for them to both depend on a separate abstract interface which is "injected" between them.
5. *Demeter Principle*: a method can call other methods in its own class, and methods on the classes of its own instance variables; everything else is taboo. A design smell that indicates a violation is *inappropriate intimacy*.

For Agile, *refactoring* is the vehicle for improving the design of classes and methods; in some cases refactoring may allow you to apply an appropriate *design pattern*. In contrast, for the Plan-and-Document lifecycles:

- The early design phase makes it easier to select a good initial software architecture and class designs.
- The specification is broken into problems and then into subproblems, where developers try to use patterns to solve them.
- As design precedes coding, *design reviews* can offer early feedback.
- One concern is whether the design must change once coding begins.

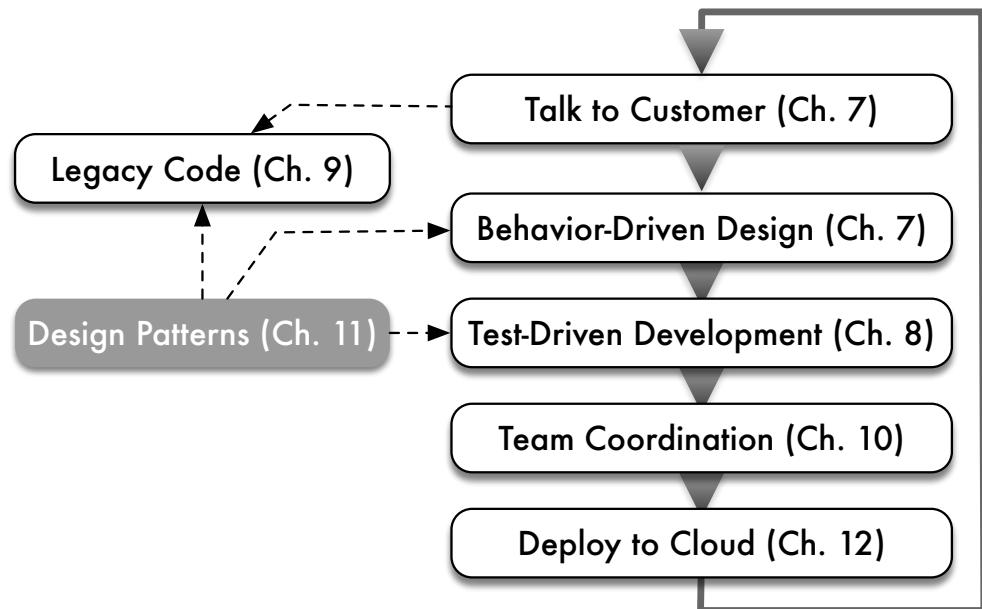


Figure 11.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter covers design patterns, which influence BDD and TDD for new apps and for enhancing legacy code.

11.1 Patterns, Antipatterns, and SOLID Class Architecture

In Chapter 3, we introduced the idea of a design pattern: a reusable structure, behavior, strategy, or technique that captures a proven solution to a collection of similar problems by *separating the things that change from those that stay the same*. Patterns play a major role in helping us achieve our goal throughout this book: producing code that is not only correct (TDD) and meets a customer need (BDD), but is also concise, readable, DRY, and generally beautiful. Figure 11.1 highlights the role of design patterns in the Agile lifecycle as covered in this chapter.

While we have already seen architectural patterns such as Client–Server and structural patterns such as Model–View–Controller, this chapter examines design patterns that apply to classes and class architecture. As Figure 11.2 shows, we will follow a similar approach as we did in Chapter 9. Rather than simply listing a catalog of design patterns, we'll motivate their use by starting from some guidelines about what makes a class architecture good or bad, identifying smells and metrics that indicate possible problem spots, and showing how some of these problems can be fixed by refactoring—both within classes and by moving code across classes—to eliminate the problems. In some cases, we can refactor to make the code match an existing and proven design pattern. In other cases, the refactoring doesn't necessarily result in major structural changes to the class architecture.

As with method-level refactoring, application of design patterns is best learned by doing, and the number of design patterns exceeds what we can cover in one chapter of one book. Indeed, there are entire books just on design patterns, including the seminal *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al. 1994), whose authors became known as the “Gang of Four” or GoF, and their catalog known as the “**GoF design patterns**”

Chapter 9	Chapter 11
Code smells warn of problems in methods of a class	Design smells warn of problems in relationships among classes
Many catalogs of code smells and refactorings; we use Fowler's as definitive	Many catalogs of design smells and design patterns; we use Ruby-specific versions of the Gang of Four (GoF) design patterns as definitive
ABC, Cyclomatic Complexity metrics complement code smells with quantitative warnings	LCOM (Lack of Cohesion of Methods) metric complements design smells with quantitative warnings
Refactoring by extracting methods and moving code within a class	Refactoring by extracting classes and moving code between classes
SOFA guidelines for good methods (Short, do One thing, Few arguments, single Abstraction level)	SOLID guidelines for good class architecture (Single responsibility, Open/Closed, Liskov substitution, dependency Injection, Demeter)
Some code smells don't apply in Ruby	Some design smells don't apply in Ruby or SaaS

Figure 11.2: The parallels between the warning symptoms and remedies introduced for individual classes and methods in Chapter 9 and those introduced for inter-class relationships in this chapter. For reasons explained in the text, whereas most books use the I in SOLID for Interface Segregation (a smell that doesn't arise in Ruby) and D for injecting Dependencies, we instead use I for Injecting dependencies and D for the Demeter principle, which arises frequently in Ruby.

terns.” The 23 GoF design patterns are divided into Creational, Structural, and Behavioral design patterns, as Figure 11.3 shows. As with Fowler’s original book on refactoring, the GoF design patterns book gave rise to other books with examples tailored to specific languages including Ruby (Olsen 2007).

The GoF authors cite two overarching principles of good object-oriented design that inform most of the patterns:

- Prefer Composition and Delegation over Inheritance.
- Program to an Interface, not an Implementation.

We will learn what these catch-phrases mean as we explore some specific design patterns.

In an ideal world, all programmers would use design patterns tastefully, continuously refactoring their code as Chapter 9 suggests, and all code would be beautiful. Needless to say, this is not always the case. An *antipattern* is a piece of code that seems to want to be expressed in terms of a well-known design pattern, but isn’t—often because the original (good) code has evolved to fill new needs without refactoring along the way. *Design smells*, similar to the code smells we saw in Chapter 9, are warning signs that your code may be headed towards an antipattern. In contrast to code smells, which typically apply to methods within a class, design smells apply to relationships between classes and how responsibilities are divided among them. Therefore, whereas refactoring a method involves moving code around *within* a class, refactoring a design involves moving code *between* classes, creating new classes or modules (perhaps by extracting commonality from existing ones), or removing classes that aren’t pulling their weight.

Similar to SOFA in Chapter 9, the mnemonic SOLID (credited to Robert C. Martin) stands for a set of five design principles that clean code should respect. As in Chapter 9, design smells and quantitative metrics can tell us when we’re in danger of violating one or more SOLID guidelines; the fix is often a refactoring that eliminates the problem by bringing the code in line with one or more design patterns.

Since the GoF design patterns evolved in the context of statically typed languages, some of them address problems that don’t arise in Ruby. For example, patterns that eliminate type signature changes that would trigger recompilation are rarely used in Ruby, which isn’t compiled and doesn’t use types to enforce contracts.

“Uncle Bob” Martin, an American software engineer and consultant¹ since 1970, is a founder of Agile/XP and a leading member of the **Software Craftsmanship** movement, which encourages programmers to see themselves as creative professionals learning a disciplined craft in an apprenticeship model.

Creational patterns

Abstract Factory, Factory Method: Provide an interface for creating families of related or dependent objects without specifying their concrete classes

Singleton: Ensure a class has only one instance, and provide a global point of access to it.

Prototype: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. As we've seen in Chapter 6, prototype-based inheritance is part of the JavaScript language.

Builder: Separate the construction of a complex object from its representation allowing the same construction process to create various representations

Structural patterns

Adapter, Proxy, Façade, Bridge: Convert the programming interface of a class into another (sometimes simpler) interface that clients expect, or decouple an abstraction's interface from its implementation, for dependency injection or performance

Decorator: Attach additional responsibilities to an object dynamically, keeping the same interface. Helps with "Prefer composition or delegation over inheritance."

Composite: Provide operations that work on both an individual object and a collection of that type of object

Flyweight: Use sharing to support large numbers of similar objects efficiently

Behavioral patterns

Template Method, Strategy: Uniformly encapsulate multiple varying strategies for same task

Observer: One or more entities need to be notified when something happens to an object

Iterator, Visitor: Separate traversal of a data structure from operations performed on each element of the data structure

Null Object: (Doesn't appear in GoF catalog) Provide an object with defined neutral behaviors that can be safely called, to take the place of conditionals guarding method calls

State: Encapsulate an object whose behaviors (methods) differ depending on which of a small number of internal states the object is in

Chain of Responsibility: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request, passing request up the chain until someone handles it

Mediator: Define an object that encapsulates how a set of objects interact without those objects having to refer to each other explicitly, allowing decoupling

Interpreter: Define a representation for a language along with an interpreter that executes the representation

Command: Encapsulate an operation request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

Figure 11.3: The 23 *GoF design patterns* spanning three categories, with *italics* showing a subset we'll encounter as we illustrate and fix SOLID violations and with closely-related patterns grouped into a single entry, as with *Abstract Factory* and *Factory Method*. Whenever we introduce a design pattern, we'll explain the pattern's goal, show a Unified Modeling Language representation (introduced in the next section) of the class architecture before and after refactoring to that pattern, and when possible, give an example of how the pattern is used "in the wild" in Rails itself or in a Ruby gem.

Principle	Meaning	Warning smells	Refactoring fix
Single Responsibility	A class should have one and only one reason to change	Large class, poor LCOM (Lack of Cohesion Of Methods) score, data clumps	Extract class, move methods
Open/Closed	Classes should be open for extension but closed for modification	Conditional complexity, case-based dispatcher	Use Strategy or Template Method, possibly combined with Abstract Factory pattern; use Decorator to avoid explosion of subclasses
Liskov Substitution	Substituting a subclass for a class should preserve correct program behavior	Refused bequest: subclass destructively overrides an inherited method	Replace inheritance with delegation
Injection of Dependencies	Collaborating classes whose implementation may vary at runtime should depend on an intermediate “injected” dependency	Unit tests that require <i>ad hoc</i> stubbing to create seams; constructors that hardwire a call to another class’s constructor, rather than allowing runtime determination of <i>which</i> other class to use	Inject a dependency on a shared interface to isolate the classes; use Adapter, Façade, or Proxy patterns as needed to make the interface uniform across variants
Demeter Principle	Speak only to your friends; treat your friends’ friends as strangers	Inappropriate intimacy, feature envy, mock trainwrecks	Delegate behaviors and call the delegate methods instead

Figure 11.4: The SOLID design guidelines and some smells that may suggest your code violates one or more of them. We diverge a little bit from standard usage of SOLID: we use I for Injecting dependencies and D for the Demeter principle, whereas most books use I for Interface Segregation (which doesn’t apply in Ruby) and D for injecting Dependencies.

Figure 11.4 shows the SOLID mnemonics and what they tell us about good composition of classes. In our discussion of selected design patterns, we’ll see violations of each one of these guidelines, and show how refactoring the bad code (in some cases, with the goal of applying a design pattern) can fix the violation. In general, the SOLID principles strive for a class architecture that avoids various problems that thwart productivity:

1. Viscosity: it’s easier to fix a problem using a quick hack, even though you know that’s not the right thing to do.
2. Immobility: it’s hard to be DRY and because the functionality you want to reuse is wired into the app in a way that makes extraction difficult.
3. Needless repetition: possibly as a consequence of immobility, the app has similar functionality duplicated in multiple places. As a result, a change in one part of the app often ripples to many other parts of the app, so that a small change in functionality requires a lot of little changes to code and tests, a process sometimes called *shotgun surgery*.
4. Needless complexity: the app’s design reflects generality that was inserted before it was needed.

As with refactoring and legacy code, seeking out design smells and addressing them by refactoring with judicious use of design patterns is a skill learned by doing. Therefore, rather than presenting “laundry lists” of design smells, refactorings, and design patterns, we focus our discussion around the SOLID principles and give a few representative examples of the overall process of identifying design smells and assessing the alternatives for addressing

them. As you tackle your own applications, perusing the more detailed resources listed in Section 11.11 is essential.

Summary of patterns, antipatterns and SOLID:

- Good code should accommodate evolutionary change gracefully. Design patterns are proven solutions to common problems that thwart this goal. They work by providing a clean way to separate the things that may change or evolve from those that stay the same and a clean way to accommodate those changes.
- Just as with individual methods, refactoring is the process of improving the structure of a class architecture to make the code more maintainable and evolvable by moving code across classes as well as refactoring within the class. In some cases, these refactorings lead us to one of the 23 “Gang of Four” (GoF) design patterns.
- Just as with individual methods, design smells and metrics can serve as early warnings of an *antipattern*—a piece of code that would be better structured if it followed a design pattern.

■ Elaboration: Other types of patterns

As we’ve emphasized since the beginning of this book, judicious use of patterns pervades good software engineering. To complement class-level design patterns, others have developed catalogs of architectural patterns for enterprise applications² (we met some in Chapter 3), parallel programming patterns, computational patterns (to support specific algorithm families such as graph algorithms, linear algebra, circuits, grids, and so on), **Concurrency patterns**, and user interface patterns³.

Grady Booch (1955–), internationally recognized for his work in software engineering and collaborative development environments, developed UML with Ivar Jacobson and James Rumbaugh.



Self-Check 11.1.1. True or false: one measure of the quality of a piece of software is the degree to which it uses design patterns.

- ◊ False: while design patterns provide proven solutions to some common problems, code that doesn’t exhibit such problems may not need those patterns, but that doesn’t make it poor code. The GoF authors specifically warn against measuring code quality in terms of design pattern usage. ■

11.2 Just Enough UML

The **Unified Modeling Language** or UML is not a textual language, but a set of graphical notation techniques that provide a “standard way to visualize the design of a [software] system.” UML evolved from 1995 to the present through the unification of previously-distinct modeling language standards and diagram types, which Figure 11.5 lists.

While this book focuses on more lightweight Agile modeling—indeed, UML-based modeling has been criticized as being too “bloated” and heavyweight—some types of UML diagrams are widely used even in Agile modeling. Figure 11.6 shows a UML **class diagram**, which depicts each actual class in the app, its most important class and instance variables and methods, and its relationship to other classes, such as has-many or belongs-to associations. Each end of the line connecting two associated classes is annotated with the minimum and

Structure diagrams	
Class	Describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
Component	Describes how a software system is split up into components and shows the dependencies among these components.
Composite structure	Describes the internal structure of a class and the collaborations that this structure makes possible.
Deployment	Describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
Object	Shows a complete or partial view of the structure of an example modeled system at a specific time.
Package	Describes how a system is split up into logical groupings by showing the dependencies among these groupings.
Profile	Describes reusable domain-specific "stereotype" objects from which specific object types can be derived for use in a particular application.
Interaction diagrams	
Communication	Shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.
Interaction overview	Provides an overview in which the nodes represent communication diagrams.
Sequence	Shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.
Timing	A specific type of interaction diagram where the focus is on timing constraints.
Behavior diagrams	
Activity	Describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
State machine	Describes the states and state transitions of the system.
Use Case	Describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.

Figure 11.5: The fourteen types of diagrams defined by UML 2.2 for describing a software system. These descriptions are based on the excellent Wikipedia summary of UML⁵, which also shows an example of each diagram type. Use case diagrams are similar to Agile user stories, but lack the level of detail that allows tools like Cucumber to bridge the gap between user stories and integration/acceptance tests.

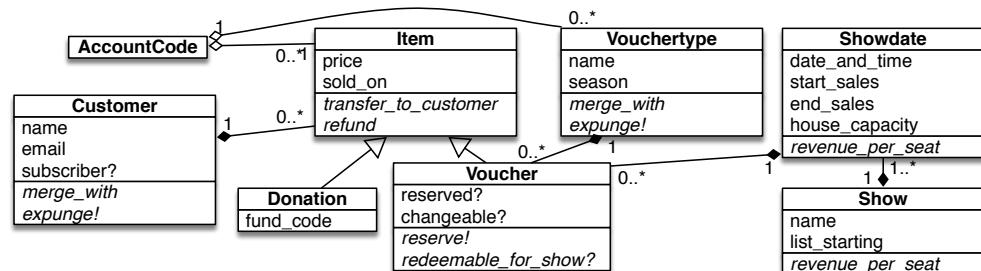


Figure 11.6: This UML *class diagram* shows a subset of the classes in the theater-ticketing app consistent with Figures 9.4 and 9.5. Each box represents a class with its most important methods and attributes (responsibilities). Inheritance is represented by an arrow. Classes with associations are connected by lines whose endpoints are annotated with a multiplicity and optionally a diamond—open for aggregations, filled for compositions, absent otherwise.

maximum number of instances that can participate in that “side” of the association, called the association’s *multiplicity*, using the symbol * for “unlimited”. For example, a multiplicity $1..*$ means “one or more”, $0..*$ means “zero or more”, and 1 means “exactly one.” UML distinguishes two kinds of “owning” (has-one or has-many) associations. In an *aggregation*, the owned objects survive destruction of the owning object. For example, *Course has many Students* is an aggregation because the students happily don’t get destroyed when the course is over! In a *composition*, the owned objects are usually destroyed when the owning object is destroyed. For example, *Movie has many Reviews* is a composition since deleting a Movie should cause all of its reviews to be deleted.

Class diagrams are popular even among software engineers who don’t use the other parts of UML. With this introduction to UML in hand, we can use class diagrams to illustrate “before and after” class architecture when we improve code using the SOLID guidelines and design patterns.

Summary of Unified Modeling Language (UML):

- UML comprises a family of diagram types to illustrate various aspects of a software design and implementation.
- UML class diagrams are widely used even by engineers who don’t use other UML features. They show a class’s name, its most important public and private methods and attributes, and its relationship to other classes.

■ Elaboration: When to use UML?

While heavyweight, UML is useful for modeling very large applications divided into subsystems being worked on by widely-distributed teams. Also, since UML notation is language-neutral, it can be helpful for coordinating international teams. Because of UML’s maturity, many tools support its use; the challenge is keeping the diagrams “in sync” with the code and the design, which is why most such tools try to go in both directions, synthesizing code skeletons from UML and extracting UML diagrams from code. One such tool useful for learning UML is **UMLe**⁶, a domain-specific language developed at the University of Ottawa for expressing class relationships. The Try UMLE⁷ web site can generate UML class diagrams from UMLE code, generate UMLE code from diagrams you draw yourself, or generate executable code in various programming languages corresponding to your UMLE code or UML diagrams. It’s a great tool for exploring UML and class diagrams, but we don’t recommend using the Ruby code it generates, which is non-DRY and somewhat non-idiomatic.



Self-Check 11.2.1. *In a UML class diagram depicting the relationship “University has many Departments,” what multiplicities would be allowable on each side of the association?*

- ◊ The University side has multiplicity 1, because a Department must belong to exactly one University. The Department side has multiplicity $1..*$, because one or more Departments can belong to a University. ■

Self-Check 11.2.2. *Should the relationship “University has many Departments” be modeled as an aggregation or a composition?*

- ◊ It should be a composition, since departments wouldn’t survive the closing of a university.



LCOM variant	Scores	Interpretation
Revised Henderson-Sellers LCOM	0 (best) to 1 (worst)	0 means all instance methods access all instance variables. 1 means any given instance variable is used by only one instance method, that is, the instance methods are fairly independent of each other.
LCOM-4	1 (best) to n (worst)	Estimates number of responsibilities n in your class as the number of connected components in a graph in which related methods' nodes are connected by an edge. $n > 1$ suggests that up to $n - 1$ responsibilities could be extracted into their own classes.

Figure 11.7: The “recommended” lack of cohesion of methods (LCOM) score depends heavily on which LCOM variant is used. The table shows two of the most widely-used variants.

11.3 Single Responsibility Principle

The **Single Responsibility Principle** (SRP) of SOLID states that a class should have one and only one responsibility—that is, only one reason to change. For example, in Section 5.2, when we added single sign-on to `RottenPotatoes`, we created a new `SessionsController` to handle the sign-on interaction. An alternate strategy would be to augment `MoviegoersController`, since sign-on is an action associated with moviegoers. Indeed, before the single sign-on approach described in Chapter 5, this was the recommended way to implementing password-based authentication in earlier versions of Rails. But such a scheme would require changing the `Moviegoer` model and controller whenever we wanted to change the authentication strategy, even though the “essence” of a `Moviegoer` doesn’t really depend on how they sign in. In MVC, each controller should specialize in dealing with one resource; an authenticated user session is a distinct resource from the user himself, and deserves its own RESTful actions and model methods. As a rule of thumb, if you cannot describe the responsibility of a class in 25 words or less, it may have more than one responsibility, and the new ones should be split out into their own classes.

In statically typed compiled languages, the cost of violating SRP is obvious: any change to a class requires recompilation and may also trigger recompilation or relinking of other classes that depend on it. Because we don’t pay this price in interpreted dynamic languages, it’s easy to let classes get too large and violate SRP. One tip-off is lack of **cohesion**, which is the degree to which the elements of a single logical entity, in this case a class, are related. Two methods are related if they access the same subset of instance or class variables or if one calls the other. The *LCOM* metric, for *Lack of Cohesion Of Methods*, measures cohesion for a class: in particular, it warns you if the class consists of multiple “clusters” in which methods within a cluster are related, but methods in one cluster aren’t strongly related to methods in other clusters. Figure 11.7 shows two of the most commonly used variants of the LCOM metric.

In Section 9.6, after successfully refactoring `convert_reek`, reported “low cohesion” in the `TimeSetter` class because we used class variables rather than instance variables for maintaining what was actually instance state, as that section described.

The *Data Clumps* design smell is one warning sign that a good class is evolving toward the “multiple responsibilities” antipattern. A Data Clump is a group of variables or values that are always passed together as arguments to a method or returned together as a set of results from a method. This “traveling together” is a sign that the values might really need their own class. Another symptom is that something that used to be a “simple” data value acquires new behaviors. For example, suppose a `Moviegoer` has attributes `phone_number` and `zipcode`, and you want to add the ability to check the zip code for accuracy or canonicalize the formatting of the phone number. If you add these methods to `Moviegoer`, they will reduce its cohesion because they form a “clique” of methods that only deal with specific instance variables. The alternative is to use the *Extract Class* refactoring to put these methods

<https://gist.github.com/a100b89bab3e1dfc3f0fbbe98fc820>

```

1 class Moviegoer
2   attr_accessor :name, :street, :phone_number, :zipcode
3   validates :phone_number, # ...
4   validates :zipcode, # ...
5   def format_phone_number ; ... ; end
6   def check_zipcode ; ... ; end
7   def format_address(street, phone_number, zipcode) # data clump
8     # do formatting, calling format_phone_number and check_zipcode
9   end
10 end
11 # After applying Extract Class:
12 class Moviegoer
13   attr_accessor :name
14   has_one :address
15 end
16 class Address
17   belongs_to :moviegoer
18   attr_accessor :phone_number, :zipcode
19   validates :phone_number, # ...
20   validates :zipcode, # ...
21   def format_address ; ... ; end # no arguments - operates on 'self'
22   private # no need to expose these now:
23   def format_phone_number ; ... ; end
24   def check_zipcode ; ... ; end
25 end

```

Figure 11.8: To perform Extract Class, we identify the group of methods that shares a responsibility distinct from that of the rest of the class, move those methods into a new class, make the “traveling together” data items on which they operate into instance variables of the class, and arrange to pass an instance of the class around rather than the individual items.

into a new **Address** class, as Figure 11.8 shows.

Summary of Single Responsibility Principle:

- A class should have one and only one reason to change, that is, one responsibility.
- A poor LCOM (Lack of Cohesion Of Methods) score and the Data Clump design smell are both warnings of possible SRP violations. The Extract Class refactoring can help remove and encapsulate additional responsibilities in a separate class.

■ Elaboration: Interface Segregation Principle

Related to SRP is the **Interface Segregation Principle** (ISP, and the original I in SOLID), which states that if a class’s API is used by multiple quite different types of clients, the API probably should be segregated into subsets useful to each type of client. For example, the **Movie** class might provide both movie metadata (MPAA rating, release date, and so on) and an interface for searching TMDb, but it’s unlikely that a client using one of those two sets of services would care about the other. The problem solved by ISP arises in compiled languages in which changes to an interface require recompiling the class, thereby triggering recompilation or relinking of classes that use that interface. While documenting separate interfaces for distinct sets of functionality is good style, ISP rarely arises in Ruby since there are no compiled classes, so we won’t discuss it further.

Self-Check 11.3.1. Draw the UML class diagrams showing class architecture before and after the refactoring in Figure 11.8.

◊ Figure 11.9 shows the UML diagrams. ■

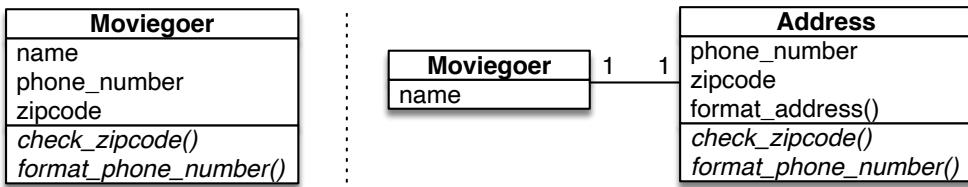


Figure 11.9: UML class diagrams before (left) and after (right) extracting the **Address** class from **Moviegoer**.

<https://gist.github.com/ce84aab55f40bdc7566e84a7512ede27>

```

1 class Report
2   def output
3     formatter =
4       case @format
5         when :html
6           HtmlFormatter.new(self)
7         when :pdf
8           PdfFormatter.new(self)
9         # ...etc
10        end
11      end
12    end

```

Figure 11.10: The **Report** class depends on a base class **Formatter** with subclasses **HtmlFormatter** and **PdfFormatter**. Because of the explicit dispatch on the report format, adding a new type of report output requires modifying **Report#output**, and probably requires changing other methods of **Report** that have similar logic—so-called *shotgun surgery*.

11.4 Open/Closed Principle

The **Open/Closed Principle** (OCP) of SOLID states that classes should be “open for extension, but closed against modification.” That is, it should be possible to extend the behavior of classes without modifying existing code on which other classes or apps depend.

While adding subclasses that inherit from a base class is one way to extend existing classes, it’s often not enough by itself. Figure 11.10 shows why the presence of case-based dispatching logic—one variant of the *Case Statement* design smell—suggests a possible OCP violation.

Depending on the specific case, various design patterns can help. One problem that the smelly code in Figure 11.10 is trying to solve is that the desired subclass of **Formatter** isn’t known until runtime, when it is stored in the `@format` instance variable. The **abstract factory pattern** provides a common interface for instantiating an object whose subclass may not be known until runtime. Ruby’s duck typing and metaprogramming enable a particularly elegant implementation of this pattern, as Figure 11.11 shows. (In statically-typed languages, to “work around” the type system, we have to create a factory method for each subclass and have them all implement a common interface—hence the name of the pattern.)

Another approach is to take advantage of the **Strategy pattern** or **Template Method pattern**. Both support the case in which there is a general approach to doing a task but many possible variants. The difference between the two is the level at which commonality is captured. With Template Method, although the implementation of each step may differ, the set of steps is the same for all variants; hence it is usually implemented using inheritance. With Strategy, the overall task is the same, but the set of steps may be different in each variant;

<https://gist.github.com/4d15a866491b82e0b6f19f1e97d5601b>

```

1 class Report
2   def output
3     formatter_class =
4       begin
5         @format.to_s.classify.constantize
6       rescue NameError
7         # ...handle 'invalid formatter type'
8       end
9     formatter = formatter_class.send(:new, self)
10    # etc
11  end
12 end

```

Figure 11.11: Ruby’s metaprogramming and duck typing enable an elegant implementation of the abstract factory pattern. `classify` is provided by Rails to convert `snake_case` to `UpperCamelCase`. `constantize` is syntactic sugar provided by Rails that calls the Ruby introspection method `Object#const_get` on the receiver. We also handle the case of an invalid value of the formatter class, which the bad code doesn’t.

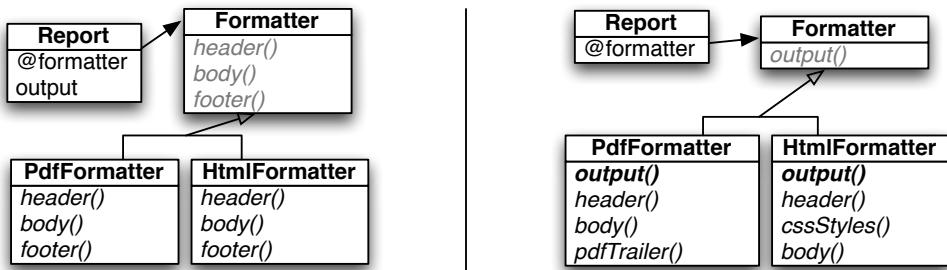


Figure 11.12: In Template Method (left), the extension points are `header`, `body`, and `footer`, since the `Report#output` method calls `@formatter.header`, `@formatter.body`, and so on, each of which delegates to a specialized counterpart in the appropriate subclass. (Light gray type indicates methods that just delegate to a subclass.) In Strategy (right), the extension point is the `output` method itself, which delegates the entire task to a subclass. Delegation is such a common ingredient of composition that some people refer to it as the *delegation pattern*.

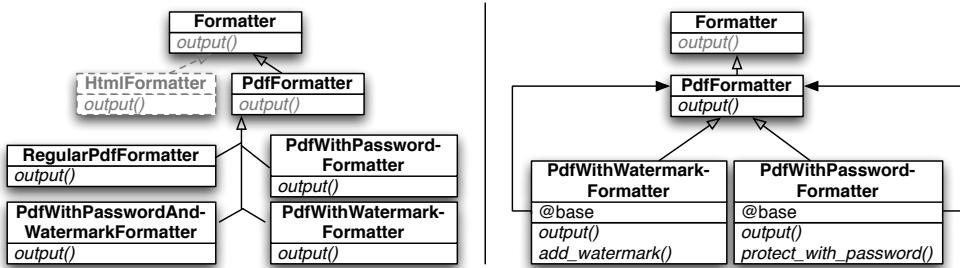


Figure 11.13: (Left) The multiplication of subclasses resulting from trying to solve the Formatter problem using inheritance shows why your class designs should “prefer composition over inheritance.” (Right) A more elegant solution uses the **Decorator** design pattern.

hence it is usually implemented using composition. Figure 11.12 shows how either pattern could be applied to the report formatter. If every kind of formatter followed the same high-level steps—for example, generate the header, generate the report body, and then generate the footer—we could use Template Method. On the other hand, if the steps themselves were quite different, it would make more sense to use Strategy.

An example of the Strategy pattern in the wild is OmniAuth (Section 5.2): many apps need third-party authentication, and the steps are quite different depending on the auth provider, but the API to all of them is the same. Indeed, OmniAuth even refers to its plug-ins as “strategies.”

A different kind of OCP violation arises when we want to *add* behaviors to an existing class and discover that we cannot do so without modifying it. For example, PDF files can be generated with or without password protection and with or without a “Draft” watermark across the background. Both features amount to “tacking on” some extra behavior to what `PdfFormatter` already does. If you’ve done a lot of object-oriented programming, your first thought might therefore be to solve the problem using inheritance, as the UML diagram in Figure 11.13 (left) shows, but there are four permutations of features so you’d end up with four subclasses with duplication across them—hardly DRY. Fortunately, the **decorator pattern** can help: we “decorate” a class or method by wrapping it in an enhanced version that has the same API, allowing us to compose multiple decorations as needed. Figure 11.14 shows the code corresponding to the more elegant decorator-based design of the PDF formatter shown in Figure 11.13 (right).

In the wild, the `ActiveSupport` module of Rails provides method-level decoration via `alias_method_chain`, which is very useful in conjunction with Ruby’s open classes, as Figure 11.15 shows. A more interesting example of Decorator in the wild is the Rack application server we’ve been using since Chapter 3. The heart of Rack is a “middleware” module that receives an HTTP request and returns a three-element array consisting of an HTTP response code, HTTP headers, and a response body. A Rack-based application specifies a “stack” of middleware components that all requests traverse: to add a behavior to an HTTP request (for example, to intercept certain requests as OmniAuth does to initiate an authentication flow), we decorate the basic HTTP request behavior. Additional decorators add support for SSL (Secure Sockets Layer), measuring app performance, and some types of HTTP caching.

Python’s “decorators”⁸ are, unfortunately, completely unrelated to the Decorator design pattern.

```
https://gist.github.com/4f46456d38c0dfb5743481ae76acb83d
1 class PdfFormatter
2   def initialize ; ... ; end
3   def output ; ... ; end
4 end
5 class PdfWithPasswordFormatter < PdfFormatter
6   def initialize(base) ; @base = base ; end
7   def protect_with_password(original_output) ; ... ; end
8   def output ; protect_with_password @base.output ; end
9 end
10 class PdfWithWatermarkFormatter < PdfFormatter
11   def initialize(base) ; @base = base ; end
12   def add_watermark(original_output) ; ... ; end
13   def output ; add_watermark @base.output ; end
14 end
15 end
16 # If we just want a plain PDF
17 formatter = PdfFormatter.new
18 # If we want a "draft" watermark
19 formatter = PdfWithWatermarkFormatter.new(PdfFormatter.new)
20 # Both password protection and watermark
21 formatter = PdfWithWatermarkFormatter.new(
22   PdfWithPasswordFormatter.new(PdfFormatter.new))
```

Figure 11.14: To apply Decorator to a class, we “wrap” class by creating a subclass (to follow the Liskov Substitution Principle, as we’ll learn in Section 11.5). The subclass delegates to the original method or class for functionality that isn’t changed, and implements the extra methods that extend the functionality. We can then easily “build up” just the version of `PdfFormatter` we need by “stacking” decorators.

<https://gist.github.com/d6e8a3aeb60516338d38ee95478be708>

```
1 # reopen Mailer class and decorate its send_email method.
2 class Mailer
3   alias_method_chain :send_email, :cc
4   def send_email_with_cc(recipient, body) # this is our new method
5     send_email_without_cc(recipient, body) # will call original method
6     copy_sender(body)
7   end
8 end
9 # now we have two methods:
10 send_email(...)           # calls send_email_with_cc
11 send_email_with_cc(...)   # same thing
12 send_email_without_cc(...) # call (renamed) original method
```

Figure 11.15: To decorate an existing method `Mailer#send_email`, we reopen its class and use `alias_method_chain` to decorate it. Without changing any classes that call `send_email`, all calls now use the decorated version that sends email and copies the sender.

Summary of Open/Closed Principle:

- To make a class open for extension but closed against modification, we need mechanisms that enable specific *extension points* at places we think extensions might be needed in the future. The *Case Statement* design smell is one symptom of a possible OCP violation.
- If the extension point takes the form of a task with varying implementations for the steps, the Strategy and Template Method patterns may apply. Both are often used in conjunction with the Abstract Factory pattern, since the variant to create may not be known until runtime.
- If the extension point takes the form of selecting different subsets of features that “add on” to existing class behaviors, the Decorator pattern may apply. The Rack application server is designed this way.

■ Elaboration: Closed against what?

“Open for extension but closed against modification” presupposes that you know in advance what the useful extension points will be, so you can leave the class open for the “most likely” changes and strategically close it against changes that might break its dependents. In our example, since we already had more than one way to do something (format a report), it seemed reasonable to allow additional formatters to be added later, but you don’t always know in advance what extension points you’ll want. Make your best guess, and deal with change as it comes.

Self-Check 11.4.1. Here are two statements about delegation:

1. A subclass delegates a behavior to an ancestor class
2. A class delegates a behavior to a descendant class

Looking at the examples of the Template Method, Strategy, and Decorator patterns (Figures 11.12 and 11.13), which statement best describes how each pattern uses delegation?

◊ In Template Method and Strategy, the ancestor class provides the “basic game plan” which is customized by delegating specific behaviors to different subclasses. In Decorator, each subclass provides special functionality of its own, but delegates back to the ancestor class for the “basic” functionality. ■

11.5 Liskov Substitution Principle

The **Liskov Substitution Principle** (LSP) is named for Turing Award winner Barbara Liskov, who did seminal work on subtypes that heavily influenced object-oriented programming. Informally, LSP states that a method designed to work on an object of type T should also work on an object of any subtype of T . That is, all of T ’s subtypes should preserve T ’s “contract.”

This may seem like common sense, but it’s subtly easy to get wrong. Consider the code in Figure 11.16, which suffers from an LSP violation. You might think a **Square** is just a

<https://gist.github.com/cca588f67ed62d38e96094ffff68a5418>

```

1 class Rectangle
2   attr_accessor :width, :height, :top_left
3   def initialize(width,height,top_left) ... ; end
4   def area ... ; end
5   def perimeter ... ; end
6 end
7 # A square is just a special case of rectangle...right?
8 class Square < Rectangle
9   # ooops...a square has to have width and height equal
10  attr_reader :width, :height, :side
11  def width=(w) ; @width = @height = w ; end
12  def height=(w) ; @width = @height = w ; end
13  def side=(w) ; @width = @height = w ; end
14 end
15 # But is a Square really a kind of Rectangle?
16 class Rectangle
17   def make_twice_as_wide_as_high(dim)
18     self.width = 2*dim
19     self.height = dim           # doesn't work!
20   end
21 end

```

Figure 11.16: Behaviorally, rectangles have some capabilities that squares don’t have—for example, the ability to set the lengths of their sides independently, as in `Rectangle#make_twice_as_wide_as_high`.

special case of `Rectangle` and should therefore inherit from it. But *behaviorally*, a square is *not* like a rectangle when it comes to setting the length of a side! When you spot this problem, you might be tempted to override `Rectangle#make_twice_as_wide_as_high` within `Square`, perhaps raising an exception since this method doesn’t make sense to call on a `Square`. But that would be a *refused bequest*—a design smell that often indicates an LSP violation. The symptom is that a subclass either destructively overrides a behavior inherited from its superclass or forces changes to the superclass to avoid the problem (which itself should indicate a possible OCP violation). The problem is that inheritance is all about implementation sharing, but if a subclass won’t take advantage of its parent’s implementations, it might not deserve to be a subclass at all.

The fix, therefore, is to again use composition and delegation rather than inheritance, as Figure 11.17 shows. Happily, because of Ruby’s duck typing, this use of composition and delegation still allows us to pass an instance of `Square` to most places where a `Rectangle` would be expected, even though it’s no longer a subclass; a statically-typed language would have to introduce an explicit interface capturing the operations common to both `Square` and `Rectangle`.

<https://gist.github.com/8c1a10862521a34e274c3f1cfa5c24d8>

```

1 # LSP-compliant solution: replace inheritance with delegation
2 # Ruby's duck typing still lets you use a square in most places where
3 # rectangle would be used - but no longer a subclass in LSP sense.
4 class Square
5   attr_accessor :rect
6   def initialize(side, top_left)
7     @rect = Rectangle.new(side, side, top_left)
8   end
9   def area      ; rect.area      ; end
10  def perimeter ; rect.perimeter ; end
11  # A more concise way to delegate, if using ActiveSupport (see text):
12  # delegate :area, :perimeter, :to => :rect
13  def side=(s) ; rect.width = rect.height = s ; end
14 end

```

Figure 11.17: As with some OCP violations, the problem arises from a misuse of inheritance. As Figure 11.18 shows, preferring composition and delegation to inheritance fixes the problem. Line 12 shows a concise syntax for delegation available to apps using ActiveSupport (and all Rails apps do); similar functionality for non-Rails Ruby apps is provided by the `Forwardable` module in Ruby's standard library.

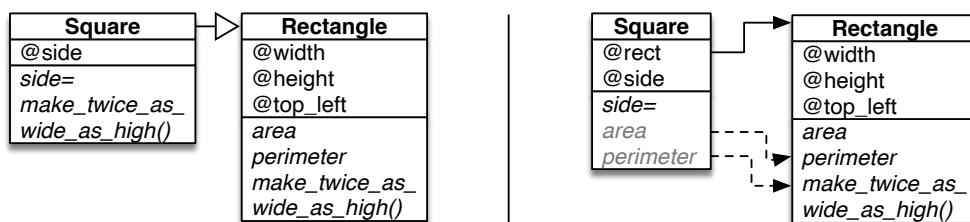


Figure 11.18: Left: The UML class diagram representing the original LSP-violating code in Figure 11.16, which destructively overrides `Rectangle#make_twice_as_wide_as_high`. Right: the class diagram for the refactored LSP-compliant code in Figure 11.17.

Summary of the Liskov Substitution Principle:

- LSP states that a method that operates on objects of some class should also work correctly on objects of any subclass of that class. When a subclass differs behaviorally from one of its parents, an LSP violation can arise.
- The *refused bequest* design smell, in which a subclass destructively overrides a parent behavior or forces changes to the parent class so that the behavior is not inherited—often signals an LSP violation.
- Many LSP violations can be fixed by using composition of classes rather than inheritance, achieving reuse through delegation rather than through subclassing.

Self-Check 11.5.1. Why is `Forwardable` in the Ruby standard library provided as a module rather than a class?

◊ Modules allow the delegation mechanisms to be mixed in to any class that wants to use them, which would be awkward if `Forwardable` were a class. That is, `Forwardable` is itself an example of preferring composition to inheritance! ■

11.6 Dependency Injection Principle

The dependency injection principle (DIP), sometimes also called dependency inversion, states that if two classes depend on each other but their implementations may change, it would be better for them to both depend on a separate abstract interface that is “injected” between them.

Suppose RottenPotatoes now adds email marketing—interested moviegoers can receive emails with discounts on their favorite movies. RottenPotatoes integrates with the external email marketing service MailerMonkey to do this job:

<https://gist.github.com/c722647142f471b8bb7806055a8c4765>

```

1  class EmailList
2    attr_reader :mailer
3    delegate :send_email, :to => :mailer
4    def initialize
5      @mailer = MailerMonkey.new
6    end
7  end
8  # in RottenPotatoes EmailListController:
9  def advertise_discount_for_movie
10    moviegoers = Moviegoer.interested_in params[:movie_id]
11    EmailList.new.send_email_to moviegoers
12  end

```

Suppose the feature is so successful that you decide to extend the mechanism so that moviegoers who are on the Amiko social network can opt to have these emails forwarded to their Amiko friends as well, using the new Amiko gem that wraps Amiko’s RESTful API for friend lists, posting on walls, messaging, and so on. There are two problems, however.

First, `EmailList#initialize` has a hardcoded dependency on `MailerMonkey`, but now we will sometimes need to use `Amiko` instead. This runtime variation is the problem solved by dependency injection—since we won’t know until runtime which type of mailer we’ll need, we modify `EmailList#initialize` so we can “inject” the correct value at runtime:

<https://gist.github.com/c9a3235dbf7be72738dcb4911ccb18e9>

```

1 class EmailList
2   attr_reader :mailer
3   delegate :send_email, :to => :mailer
4   def initialize(mailer_type)
5     @mailer = mailer_type.new
6   end
7 end
8 # in RottenPotatoes EmailListController:
9 def advertise_discount_for_movie
10  moviegoers = Moviegoer.interested_in params[:movie_id]
11  mailer = if Config.has_amiko? then AmikoAdapter else MailerMonkey end
12  EmailList.new(mailer).send_email_to moviegoers
13 end

```

You can think of DIP as injecting an additional seam between two classes, and indeed, in statically compiled languages DIP helps with testability. This benefit is less apparent in Ruby, since as we've seen we can create seams almost anywhere we want at runtime using mocking or stubbing in conjunction with Ruby's dynamic language features.

The second problem is that Amiko exposes a different and more complex API than the simple `send_email` method provided by `MailerMonkey` (to which `EmailList#send_email` delegates in line 3), yet our controller method is already set up to call `send_email` on the mailer object. The **Adapter pattern** can help us here: it's designed to convert an existing API into one that's compatible with an existing caller. In this case, we can define a new class `AmikoAdapter` that converts the more complex Amiko API into the simpler one that our controller expects, by providing the same `send_email` method that `MailerMonkey` provides:

<https://gist.github.com/411c249b2ba9e9f04613df32bc8862c1>

```

1 class AmikoAdapter
2   def initialize ; @mailer = Amiko.new(...) ; end
3   def send_email
4     @mailer.authenticate(...)
5     @mailer.send_message(...)
6   end
7 end
8 # Change the controller method to use the adapter:
9 def advertise_discount_for_movie
10  moviegoers = Moviegoer.interested_in params[:movie_id]
11  mailer = if Config.has_amiko? then AmikoAdapter else MailerMonkey end
12  EmailList.new(mailer).send_email_to moviegoers
13 end

```

When the Adapter pattern not only converts an existing API but also simplifies it—for example, the `Amiko` gem also provides many other Amiko functions unrelated to email, but `AmikoAdapter` only “adapts” the email-specific part of that API—it is sometimes called the **Façade pattern**.

Lastly, even in cases where the email strategy is known when the app starts up, what if we want to disable email sending altogether from time to time? Figure 11.19 (top) shows a naive approach: we have moved the logic for determining whichemailer to use into a new `Config` class, but we still have to “condition out” the email-sending logic in the controller method if email is disabled. But if there are other places in the app where a similar check must be performed, the same condition logic would have to be replicated there (shotgun surgery). A better alternative is the **Null Object pattern**, in which we create a “dummy” object that has all the same behaviors as a real object but doesn't do anything when those behaviors are called. Figure 11.19 (bottom) applies the Null Object pattern to this example, avoiding the proliferation of conditionals throughout the code.

Figure 11.20 shows the UML class diagrams corresponding to the various versions of our

ActiveRecord has been criticized for configuring the database at startup from `database.yml` rather than using DIP. Presumably the designers judged that the database wouldn't change while the app was running. While DIP-induced seams also help with stubbing and mocking, Chapter 8 shows that Ruby's open classes and metaprogramming let you insert test seams wherever needed.

<https://gist.github.com/3533bdccb7bcd3fb36b7f18fc79862>

```

1 class Config
2   def self.email_enabled? ; ... ; end
3   def self.emailer ; if has_amiko? then Amiko else MailerMonkey end ; end
4 end
5 def advertise_discount_for_movie
6   if Config.email_enabled?
7     moviegoers = Moviegoer.interested_in(params[:movie_id])
8     EmailList.new(Config.emailer).send_email_to(moviegoers)
9   end
10 end

```

<https://gist.github.com/4b333ccb9cbf7f5de75c1ca9d94e7ad6>

```

1 class Config
2   def self.emailer
3     if email_disabled? then NullMailer else
4       if has_amiko? then Amiko else MailerMonkey end
5     end
6   end
7 end
8 class NullMailer
9   def initialize ; end
10  def send_email_to(*args) ; true ; end
11 end
12 def advertise_discount_for_movie
13   moviegoers = Moviegoer.interested_in(params[:movie_id])
14   EmailList.new(Config.emailer).send_email_to(moviegoers)
15 end
16 end

```

Figure 11.19: Top: a naive way to disable a behavior is to “condition it out” wherever it occurs. Bottom: the Null Object pattern eliminates the conditionals by providing “dummy” methods that are safe to call but don’t do anything.

DIP example.

An interesting relative of the Adapter and Façade patterns is the **Proxy pattern**, in which one object “stands in” for another that has the same API. The client talks to the proxy instead of the original object; the proxy may forward some requests directly to the original object (that is, delegate them) but may take other actions on different requests, perhaps for reasons of performance or efficiency.

Two classic examples of this pattern are found in ActiveRecord itself. First, the object returned by ActiveRecord’s `all`, `where` and `find`-based methods quacks like a collection, but it’s actually a proxy object that doesn’t even do the query until you force the issue by asking for one of the collection’s elements. That is why you can build up complex queries with multiple `wheres` without paying the cost of doing the query each time. The second is when you use ActiveRecord’s associations (Section 5.4): the result of evaluating `@movie.reviews` quacks like an enumerable collection, but it’s actually a proxy object that responds to all the collection methods (`size`, `<<`, and so on), without querying the database except when it has to. Another example of a use for the proxy pattern would be for sending email while disconnected from the Internet. If the real Internet-based email service is accessed via a `send_email` method, a proxy object could provide a `send_email` method that just stores an email on the local disk until the next time the computer is connected to the Internet. This proxy shields the client (email GUI) from having to change its behavior when the user isn’t connected.

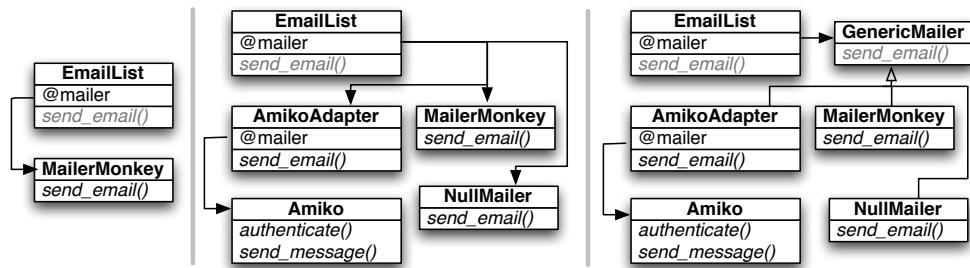


Figure 11.20: Left: Without dependency injection, `EmailList` depends directly on `MailerMonkey`. Center: With dependency injection, `@mailer` can be set at runtime to use any of `MailerMonkey`, `NullMailer` (which implements the Null Object pattern to disable email), or `AmikoAdapter` (which implements the Adapter/Façade pattern over `Amiko`), all of which have the same API. Right: In statically typed languages, the abstract superclass `GenericMailer` formalizes the fact that all three mailers have compatible APIs, but in Ruby this superclass is often omitted if it consists entirely of abstract methods (as is the case here), since abstract methods and classes aren't part of the language.

Summary of Dependency Injection:

- Dependency injection inserts a seam between two classes by passing in (injecting) a dependency whose value may not be known until runtime, rather than hardwiring a dependency into the source code.
- Because dependency injection is often used to vary which of a collection of implementations is used at runtime, it's often seen together with the Adapter pattern, in which a class converts one API into another that a client expects to use.
- Variations on Adapter include Façade, in which the API is not only adapted but also simplified, and Proxy, in which the API is exactly imitated but the behaviors changed to accommodate different usage conditions without the client (caller of the API) having to change its behavior.
- The Null Object pattern is another mechanism for replacing unwieldy conditionals with safe “neutral” behaviors as a way of disabling a feature.

■ Elaboration: Did injecting a dependency violate the Open/Closed Principle?

You might wonder whether our “fix” to add a second type of mailer service violates OCP, because adding support for a third mailer would then require modifying `advertise_discount_for_movie`. If you had reason to believe you might indeed need to add additional mailers later, you could combine this with the Abstract Factory pattern introduced in Section 11.4. This scenario is an example of making a judgment call about whether the possibility of handling additional mailers is an extension point you want to leave open, or a change you feel the app wouldn't accommodate well and should therefore be strategically closed against.

Self-Check 11.6.1. Why does proper use of DIP have higher impact in statically typed languages?

- ◊ In such languages, you cannot create a runtime seam to override a “hardwired” behavior as you can in dynamic languages like Ruby, so the seam must be provided in advance by

<https://gist.github.com/d3ea5309672163c64674b21033c9106c>

```

1 # This example is adapted from Dan Manges's blog, dcmanges.com
2 class Wallet ; attr_accessor :credit_balance ; end
3 class Moviegoer
4   attr_accessor :wallet
5   def initialize
6     # ...setup wallet attribute with correct credit balance
7   end
8 end
9 class MovieTheater
10  def collect_money(moviegoer, due_amount)
11    # VIOLATION OF DEMETER (see text)
12    if moviegoer.wallet.credit_balance < due_amount
13      raise InsufficientFundsError
14    else
15      moviegoer.wallet.credit_balance -= due_amount
16      @collected_amount += due_amount
17    end
18  end
19 end
20 # Imagine testing the above code:
21 describe MovieTheater do
22   describe "collecting money" do
23     it "should raise error if moviegoer can't pay" do
24       # "Mock trainwreck" is a warning of a Demeter violation
25       wallet = double('wallet', :credit_balance => 5.00)
26       moviegoer = double('moviegoer', :wallet => wallet)
27       expect { @theater.collect_money(moviegoer, 10.00) }.
28         to raise_error(...)
29     end
30   end
31 end

```

Figure 11.21: Line 12 contains a Demeter violation: while it’s reasonable for `MovieTheater` to know about `Moviegoer`, it also knows about the implementation of `Wallet`, since it “reaches through” the `wallet` attribute to manipulate the wallet’s `credit_balance`. Also, we’re handling the problem of “not enough cash” in `MovieTheater`, even though logically it seems to belong in `Wallet`.

injecting the dependency. ■

11.7 Demeter Principle

The name comes from the Demeter Project on adaptive and aspect-oriented programming, which in turn is named for the Greek goddess of agriculture to signify a “from the ground up” approach to programming.

The Demeter Principle or **Law of Demeter** states informally: “Talk to your friends—don’t get intimate with strangers.” Specifically, a method can call other methods in its own class, and methods on the classes of its own instance variables; everything else is taboo. Demeter isn’t originally part of the SOLID guidelines, as Figure 11.4 explains, but we include it here since it is highly applicable to Ruby and SaaS, and we opportunistically hijack the **D** in SOLID to represent it.

The Demeter Principle is easily illustrated by example. Suppose RottenPotatoes has made deals with movie theaters so that moviegoers can buy movie tickets directly via RottenPotatoes by maintaining a credit balance (for example, by receiving movie theater gift cards).

Figure 11.21 shows an implementation of this behavior that contains a Demeter Principle violation. A problem arises if we ever change the implementation of `Wallet`—for example, if we change `credit_balance` to `cash_balance`, or add `points_balance` to allow moviegoers to accumulate PotatoPoints by becoming top reviewers. All of a sudden, the `MovieTheater` class, which is “twice removed” from `Wallet`, would have to change.

Two design smells can tip us off to possible Demeter violations. One is **inappropri-**

<https://gist.github.com/6a636b98e53b84c9628eba411d3d4086>

```

1 # Better: delegate credit_balance so MovieTheater only accesses Moviegoer
2 class Moviegoer
3   def credit_balance
4     self.wallet.credit_balance # delegation
5   end
6 end
7 class MovieTheater
8   def collect_money(moviegoer,due_amount)
9     if moviegoer.credit_balance >= due_amount
10    moviegoer.credit_balance -= due_amount
11    @collected_amount += due_amount
12  else
13    raise InsufficientFundsError
14  end
15 end
16 end

```

<https://gist.github.com/8a6859cdd248ab77673f3eb5373e9a67>

```

1 class Wallet
2   attr_reader :credit_balance # no longer attr_accessor!
3   def withdraw(amount)
4     raise InsufficientFundsError if amount > @credit_balance
5     @credit_balance -= amount
6     amount
7   end
8 end
9 class Moviegoer
10  # behavior delegation
11  def pay(amount)
12    wallet.withdraw(amount)
13  end
14 end
15 class MovieTheater
16   def collect_money(moviegoer, amount)
17     @collected_amount += moviegoer.pay(amount)
18   end
19 end

```

Figure 11.22: (Top) If `Moviegoer` delegates `credit_balance` to its `wallet`, `MovieTheater` no longer has to know about the implementation of `Wallet`. However, it may still be undesirable that the payment behavior (subtract payment from credit balance) is exposed to `MovieTheater` when it should really be the responsibility of `Moviegoer` or `Wallet` only. (Bottom) Delegating the behavior of payment, rather than the attributes through which it's accomplished, solves the problem and eliminates the Demeter violation.

ate intimacy: the `collect_money` method manipulates the `credit_balance` attribute of `Wallet` directly, even though managing that attribute is the `Wallet` class's responsibility. (When the same kind of inappropriate intimacy occurs repeatedly throughout a class, it's sometimes called **feature envy**, because `Moviegoer` "wishes it had access to" the features managed by `Wallet`.) Another smell that arises in tests is the *mock trainwreck*, which occurs in lines 25–27 of Figure 11.21: to test code that violates Demeter, we find ourselves setting up a "chain" of mocks that will be used when we call the method under test.

Once again, delegation comes to the rescue. A simple improvement comes from delegating the `credit_balance` attribute, as Figure 11.22 (top) shows. But the best delegation is that in Figure 11.22 (bottom), since now the behavior of payment is entirely encapsulated within `Wallet`, as is the decision of when to raise an error for failed payments.

Inappropriate intimacy and Demeter violations can arise in any situation where you feel you are "reaching through" an interface to get some task done, thereby exposing yourself to dependency on implementation details of a class that should really be none of your business. Three design patterns address common scenarios that could otherwise lead to Demeter vio-

<https://gist.github.com/7d58ab75f108b42105b6c198040f831b>

```

1 class EmailList
2   observe Review
3   def after_create(review)
4     moviegoers = review.moviegoers # from has_many :through, remember?
5     self.email(moviegoers, "A new review for #{review.movie} is up.")
6   end
7   observe Moviegoer
8   def after_create(moviegoer)
9     self.email([moviegoer], "Welcome, #{moviegoer.name}!")
10    end
11    def self.email ; ... ; end
12  end

```

Figure 11.23: An email list subsystem observes other models so it can generate email in response to certain events. The **Observer pattern** is an ideal fit since it collects all the concerns about when to send email in one place.

lations. One is the Visitor pattern, in which a data structure is traversed and you provide a callback method to execute for each member of the data structure, allowing you to “visit” each element while remaining ignorant of the way the data structure is organized. Indeed, the “data structure” could even be materialized lazily as you visit the different nodes, rather than existing statically all at once. An example of this pattern in the wild is the Nokogiri⁹ gem, which supports traversal of HTML and XML documents organized as a tree: in addition to searching for a specific element in a document, you can have Nokogiri traverse the document and call a visitor method you provide at each document node.

A simple special case of Visitor is the **Iterator pattern**, which is so pervasive in Ruby (you use it anytime you use `each`) that many Rubyists hardly think of it as a pattern. Iterator separates the implementation of traversing a collection from the behavior you want to apply to each collection element. Without iterators, the behavior would have to “reach into” the collection, thereby knowing inappropriately intimate details of how the collection is organized.

The last design pattern that can help with some cases of Demeter violations is the **Observer pattern**, which is used when one class (the observer) wants to be kept aware of what another class is doing (the subject) without knowing the details of the subject’s implementation. The Observer design pattern provides a canonical way for the subject to maintain a list of its observers and notify them automatically of any state changes in which they have indicated interest, using a narrow interface to separate the concept of observation from the specifics of what each observer does with the information.

While the Ruby standard library includes a mixin¹⁰ called `Observable`, Rails’ ActiveSupport provides a more concise Observer that lets you observe any model’s ActiveRecord lifecycle hooks (`after_save` and so on), introduced in Section 5.1. Figure 11.23 shows how easy it is to add an `EmailList` class to RottenPotatoes that “subscribes” to two kinds of state changes:

1. When a new review is added, it emails all moviegoers who have already reviewed that same movie.
2. When a new moviegoer signs up, it sends her a “Welcome” email.

In addition to ActiveRecord lifecycle hooks, Rails caching, which we will encounter in Chapter 12, is another example of the Observer pattern in the wild: the cache for each type of ActiveRecord model observes the model instance in order to know when model instances

Observer was first implemented in the MVC framework of **Smalltalk**, from which Ruby inherits its object model.

become stale and should be removed from the cache. The observer doesn't have to know the implementation details of the observed class—it just gets called at the right time, like Iterator and Visitor.

To close out this section, it's worth pointing out an example that looks like it violates Demeter, but really doesn't. It's common in Rails views (say, for a Review) to see code such as:

<https://gist.github.com/c5ddf39643f6763c5b7ec377a8cdd5e7>

```
1 | <p> Review of: <%= @review.movie.title %>    </p>
2 | <p> Written by: <%= @review.moviegoer.name %> </p>
```

Aren't these Demeter violations? It's a judgment call: strictly speaking, a `review` shouldn't know the implementation details of `movie`, but it's hard to argue that creating delegate methods `Review#movie_title` and `Review#moviegoer_name` would enhance readability in this case. The general opinion in the Rails community is that it's acceptable for views whose purpose is to display object relationships to also expose those relationships in the view code, so examples like this are usually allowed to stand.

Summary of Demeter Principle:

- The Demeter Principle states that a class shouldn't be aware of the details of collaborator classes from which it is further away than “once removed.” That is, you can access instance methods in your own class and in the classes corresponding to your nearest collaborators, but not on *their* collaborators.
- The Inappropriate Intimacy design smell, which sometimes manifests as a Mock Trainwreck in unit tests, may signal a Demeter violation. If a class shows many instances of Inappropriate Intimacy with another class, it is sometimes said to have Feature Envy with respect to the other class.
- Delegation is the key mechanism for resolving these violations.
- Design patterns cover some common manipulations of classes without violating Demeter, including Iterator and Visitor (separating traversal of an aggregate from behavior) and Observer (separating notification of “interesting” events from the details of the class being observed).

■ Elaboration: Observers, Visitors, Iterators, and Mixins

Because of duck typing and mixins, Ruby can express many design patterns with far less code than statically-typed languages, as the Wikipedia entries for **Observer**, **Iterator** and **Visitor** clearly demonstrate by using Java-based examples. In contrast to Ruby's internal iterators based on `each`, statically-typed languages usually provide external iterators and visitors in which you set up the iterator over a collection and ask the iterator explicitly whether the collection has any more elements, sometimes requiring various contortions to work around the type system. Similarly, Observer usually requires modifying the subject class(es) so that they can implement an `Observable` interface, but Ruby's open classes allow us to skip that step, as Figure 11.23 showed: from the programmer's point of view, all of the logic is in the observing class, not the subjects.

Self-Check 11.7.1. Ben Bitdiddle is a purist about Demeter violations, and he objects to the

expression @movie.reviews.average_rating in the movie details view, which shows a movie's average review score. How would you placate Ben and fix this Demeter violation?

<https://gist.github.com/77f741c6c441b81be9c3220bf01a05ee>

```

1 # naive way:
2 class Movie
3   has_many :reviews
4   def average_rating
5     self.reviews.average_rating # delegate to Review#average_rating
6   end
7 end
8 # Rails shortcut:
9 class Movie
10  has_many :reviews
11  delegate :average_rating, :to => :review
12 end

```

Self-Check 11.7.2. *Notwithstanding that “delegation is the key mechanism” for resolving Demeter violations, why should you be concerned if you find yourself delegating many methods from class A to class B just to resolve Demeter violations present in class C?*

- ◊ You might ask yourself whether there should be a direct relationship between class C and class B, or whether class A has “feature envy” for class B, indicating that the division of responsibilities between A and B might need to be reengineered. ■

11.8 The Plan-And-Document Perspective on Design Patterns

A strength of Plan-and-Document is that careful upfront planning can result in a product with a good software architecture that uses design patterns well. This preplanning is reflected in the alternative catch phrase for these processes of ***Big Design Up Front***, as Chapter 1 mentions.

A Plan-and-Document development team starts with the ***Software Requirements Specification (SRS)*** (see Section 7.9), which the team breaks into a series of problems. For each one, the team looks for one or more architecture patterns that might solve the problem. The team then goes down to the next level of subproblems, and looks for design patterns that match them. The philosophy is to learn from the experience of others captured as patterns so as to avoid repeating the mistakes of your predecessors. Another way to get feedback from more experienced engineers is to hold a ***design review*** (see Section 10.7). Note that design reviews can be done before any code is written in Plan-and-Document processes.

Thus, compared to Agile, there is considerably more effort in starting with a good design in Plan-and-Document. As Martin Fowler points out in his article *Is Design Dead?*¹¹, a frequent critique of Agile is that it encourages developers to jump in and start coding without any design, and rely too much on refactoring to fix things later. As the critics sometimes say, you can build a doghouse by slapping stuff together and planning as you go, but you can't build a skyscraper that way.

Agile supporters counter that Plan-and-Document methods are just as bad: by disallowing any code until the design is complete, it's impossible to be confident that the design will be implementable or that it really captures the customer's needs. This critique especially holds when the architects/designers will not be writing the code or may be out of touch with current coding practices and tools. As a result, say Agile proponents, when coding starts, the design will have to change anyway.

Both sides have a point, but the critique can be phrased in a more nuanced way as “How much design makes sense up front?” For example, Agile developers plan for persistent storage as part of their SaaS apps, even though the first BDD and TDD tests they write will not touch the database. A more subtle example is horizontal scaling. As we alluded to in Chapter 3, and will discuss more fully in Chapter 12, designers of successful SaaS *must* think about horizontal scalability early on. Even though it may be months before scalability matters, design decisions early in the project can cripple scalability, and it may be difficult to change them without major rewriting and refactoring.

A possible solution to the conundrum is captured by a rule of thumb in Fowler’s article. If you have previously done a project that has some design constraint or element, it’s OK to plan for it in a new project that is similar, because your previous experience will likely lead to reasonable design decisions this time.

Summary: Plan-and-Document processes have an explicit design phase that is a natural fit to the use of design patterns in the software development process. One potential drawback is uncertainty as to whether the initial architecture and design patterns will need to change as the code is written and as the system evolves. In contrast, the Agile process relies on refactoring to incorporate design patterns as the code evolves, although experienced developers may lay plans for software architectures and design patterns that they expect to need based on previous, similar projects.

Self-Check 11.8.1. *True or False: Agile design is an oxymoron.*

- ◊ False. Although there is no separate design phase in Agile development, the refactoring that is the norm in Agile can incorporate design patterns. ■

11.9 6S: A Clean Code Checklist

A key message of this book is that when you write code, you’re writing for other developers who will maintain it after you’ve moved on. The many tools and techniques we introduced—software architectures such as model-view-controller and client-server, class-level and method-level design patterns, and tools for measuring and improving code quality through refactoring—are all there to help enhance code beauty and therefore maintainability.

While there is no fixed recipe for creating beautiful code (other than lots of practice), the information in this book on refactoring (Chapter 9) and design patterns (this chapter) can go a long way towards helping you produce beautiful code, and the frequent code reviews that are part of the pull request process (Chapter 10) can serve as a further cross-check. You have also learned in multiple contexts that while automated code quality tools are useful, they are no substitute for engineering judgment, a solid understanding of the codebase, and a team whose members trust and rely on each other for constructive criticism.

To help pull all of this advice together, in this section we present a “checklist” you and your team can use to evaluate your code. Since Agile is all about iterative refinement in the pursuit of higher code quality, we will call it the “6S List,” which we hope you will remember because it almost sounds like “success list.” Here is our proposed 6S list, ordered from highest to lowest level of abstraction, to use as a checklist before opening a pull request: Site, SOLID, SOFA, Smells, Style, and Sign-off.



The name is also a nod to **Six Sigma**, a technique for manufacturing-process improvement that uses statistical methods to find and remediate process problems that harm output quality.

Site. If you’re adding new code, is it in the right place architecturally? Does it properly belong directly in a model, view, controller, or helper? Or, at least as often, should it be separately modularized as a service object, form object, or other type of code (Section 5.8), with clear and explicit dependencies on other existing classes? Is the new code connected in the proper way to the rest of the app? For example, JavaScript code must be appropriately loaded by the asset pipeline, and may require DOM bindings (Section 6.4) on particular app pages.

SOLID. If you’re adding new classes or adding new code to existing classes, really scrutinizing your code against each of the SOLID principles (Sections 11.3–11.7) can reveal risk areas in which you are introducing unnecessary technical debt—changes that will make the code harder to understand, maintain, or modify later. This is the moment to identify possible refactorings, even simple ones such as extracting a class (Section 11.3). Automated tools such as CodeClimate can help identify trouble spots. Once technical debt is incurred and gets into the mainline codebase, it is *extremely resistant* to being paid back, since new features or other work always seem to take priority.

SOFA. Just as with SOLID, make an effort to critically apply each of the four SOFA criteria (Section 9.5) to any new or refactored methods within each class. Is each method **Short**, because it does only **One** thing requiring **Few** arguments while maintaining a single level of **Abstraction**? Just as one of the most frequent violations of SOLID is violating the Single Responsibility Principle, one of the most frequent violations of SOFA is violating “Methods should be Short.” Both violations are usually symptomatic of other violations likely lurking in the code.

Smells. Is the code relatively free of code and design smells (Chapters 9 and 11)? Tools like CodeClimate can find both language-independent smells (high cyclomatic complexity, deeply nested conditionals, meaningless variable names, arguments that travel around as a group) and language-specific or framework-specific problems (overly long controller actions, use of potentially confusing language idioms). Again, use automated tools in tandem with good engineering judgment, not as a substitute for it.

Style. Most languages and frameworks have low-level best practices regarding variable and function naming (Section 2.3), spacing and indentation, punctuation, and so on. Where such practices are unambiguous, your code should follow them; where there is flexibility in the practices, your code should follow the existing conventions of the codebase. All modern editors allow customizing the rules for automatic indentation and spacing, use of spaces rather than tabs, and so on; configure your editor to match the codebase’s conventions.

Sign-off. When you’ve checked all of the above, the last step is to open a pull request and invite team feedback until the code passes your team’s review with “LGTM” (“looks good to me”) or similar. Automated tools are wonderful, but the eyes and brains of your team can produce codebase-specific suggestions beyond what automated tools are able to do.

Summary:

- The six S's—site, SOLID, SOFA, smells, style, sign-off—can help you check your own work as you get more proficient at creating beautiful code.
- Automated tools can help with some of the S's, but in the end, there's no substitute for an informed team who trust and hold each other responsible for providing constructive feedback. Keeping the codebase clean is everyone's job!

■ Elaboration: Guidelines or Rules?

In a 2013 presentation¹² at the Barcelona Ruby Conference, Rubyist Sandi Metz, the author of one of the books we recommend on object-oriented design (Metz 2012), proposes a set of rules based on a seventh S, *size*. Her proposed rules are:

- Classes must be no longer than 100 lines
- Methods must be no longer than 5 lines, and take no more than 4 arguments
- Controller actions may name at most 2 other classes, and may set at most 1 instance variable to be consumed by the view

The rules may only be broken if you can convince your pair programming partner that it makes sense to do so in a given situation. Metz notes that while these thresholds are arbitrary, at least they force developers to think about *why* they might need to break the rules or whether they should refactor. We largely agree with her rules (even if the thresholds are a bit tight), and we hope the 6S approach will help identify *why* the rules are about to be broken and provide guidance for how to improve.

Self-Check 11.9.1.

◊ ■

11.10 Fallacies and Pitfalls

**Pitfall: Over-reliance or under-reliance on patterns.**

As with every tool and methodology we've seen, slavishly following design patterns is a pitfall: they can help point the way when your problem could take advantage of a proven solution, but they cannot by themselves ensure beautiful code. In fact, the GoF authors specifically warn *against* trying to evaluate the soundness of a design based on the number of patterns it uses. In addition, if you apply design patterns too early in your design cycle, you may try to implement a pattern in its full generality even though you may not need that generality for solving the current problem. That will complicate your design because most design patterns call for *more* classes, methods, and levels of indirection than the same code would require without this level of generality. In contrast, if you apply design patterns too late, you risk falling into antipatterns and extensive refactoring.

What to do? Develop taste and judgment through learning by doing. You will make some mistakes as you go, but your judgment on how to deliver working and maintainable code will quickly improve.



Pitfall: Over-reliance on UML or other diagrams.

A diagram's purpose is communication of intent. Reading UML diagrams is not necessarily easier than reading user stories or well-factored TDD tests. Create a diagram when it helps to clarify a class architecture; don't rely on them as a crutch.



Fallacy: SOLID principles aren't needed in dynamic languages.

As we saw in this chapter, some of the problems addressed by SOLID don't really arise in dynamically-typed languages like Ruby. Nonetheless, the SOLID guidelines still represent good design; in static languages, there is simply a much more tangible up-front cost to ignoring them. In dynamic languages, while the opportunity exists to use dynamic features to make your code more elegant and DRY without the extra machinery required by some of the SOLID guidelines, the corresponding risk is that it's easier to fall into sloth and end up with ugly antipattern code.



Pitfall: Lots of private methods in a class.

You may have already discovered that methods declared `private` are hard to test, because by definition they can only be called from within an instance method of that class—meaning they cannot be called directly from an RSpec test. Although you can use a hack to temporarily make the method public (`MyClass.send(:public,:some_private_method)`), private methods complex enough to need their own tests should be considered a smell: the methods themselves may be too long, violating the Short guideline of SOFA, and the class containing these methods may be violating the **Single Responsibility Principle**. In this case, consider extracting a collaborator class whose methods are public (and therefore easy to test and easy to shorten by refactoring) but are only called from the original class, thereby improving maintainability and testability.



Pitfall: Using `initialize` to implement factory patterns.

In Section 11.4, we showed an example of Abstract Factory pattern in which the correct subclass constructor is called directly. Another common scenario is one in which you have a class `A` with subclasses `A1` and `A2`, and you want calls to `A`'s constructor to return a new object of the correct subclass. You usually cannot put the factory logic into the `initialize` method of `A`, because that method must by definition return an instance of class `A`. Instead, give the factory method a different name such as `create`, make it a class method, and call it from `A`'s constructor:

<https://gist.github.com/f6cd62078436064577a5773617fccccd4>

```

1  class A
2    def self.create(subclass, *args) # subclass must be either 'A1' or 'A2'
3      return Object.const_get(subclass).send(:new, *args)
4    end
5  end

```

11.11 Concluding Remarks: Frameworks Capture Design Patterns

The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be

an aesthetic experience much like composing poetry or music.

—Donald Knuth

The idea of design patterns is inspired by Christopher Alexander's 1977 book *A Pattern Language: Towns, Buildings, Construction* describing design patterns for civil architecture. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (the "Gang Of Four" or GoF) published the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* in 1995 (Gamma et al. 1994). It described what are now called the 23 GoF Design Patterns focusing on class-level structures and behaviors. The original 23 design patterns from the Gang of Four have been expanded dramatically since their book appeared. There are numerous repositories of design patterns (Cunningham 2013; Noble and Johnson 2013), with some tailored to specific problem areas such as user interfaces (Griffiths 2013; Toxboe 2013).

Despite design patterns' popularity as a tool, they have been the subject of some critique; for example, Peter Norvig, currently Google's Director of Research, has argued that some design patterns just compensate for deficiencies in statically-typed programming languages such as C++ and Java, and that the need for them disappears in dynamic languages such as Lisp or Ruby. Notwithstanding some controversy, patterns of many kinds remain a valuable way for software engineers to identify structure in their work and bring proven solutions to bear on recurring problems.

A problem for novice developers is that even if you read the Gang of Four book or study these repositories, it is hard to know which pattern to apply. If you don't have previous experience with a given design pattern, and you try to design for it in an anticipatory manner, you're more likely to get it wrong, so you should instead wait to add it later when and if it's really needed.

The good news is that frameworks like Rails encapsulate others' design experience to provide abstractions and design constraints that have been proven through reuse. For example, it may not occur to you to design your app's actions around REST, but it turns out that doing so results in a design that is more consistent with the scalability success stories of the Web. While the Gang of Four went out of their way to differentiate design patterns from frameworks to try to make it clear what design patterns are—more abstract, narrower in focus, and not targeted to a problem domain—today frameworks are a great way for a novice to get started with design patterns. By examining the patterns in a framework that are instantiated as code, you can gain experience on how to create your own code based on design patterns.

Design Patterns (Gamma et al. 1994) is the classic Gang of Four text on design patterns. While canonical, it's a bit slower reading than some other sources, and the examples are heavily oriented to C++. *Design Patterns in Ruby* (Olsen 2007) treats a subset of the GoF patterns in detail showing Ruby examples. It also discusses patterns made unnecessary by Ruby language features. *Clean Code* (Martin 2008) has a more thorough exposition of both the SOFA and SOLID guidelines that motivate the use of design patterns.

Rather than presenting a "laundry list" of patterns, we tried to motivate a subset of patterns by showing the design smells they fix. *Rails Antipatterns* (Pytel and Saleh 2010) gives great examples of how real-life code that starts with a good design can become cluttered over time, and how to beautify and streamline it by refactoring, often using one or more of the appropriate design patterns. Figure 11.24 shows a few examples of those refactorings¹³ and comprehensive book (Fields et al. 2009).



Smell	Description	Fix
Comment deodorant, inappropriate name	Obfuscated variable or method names make lots of comments necessary	Reduce need for comments through descriptive names and (as necessary) by addressing other smells within the offending code
Lazy class, data class	A class does too little, for example, providing nothing but getters and setters for some object but no other logic	Merge methods that encapsulate the data object into another class
Duplicated code, combinatorial explosion	Nearly the same code repeated with subtle changes in multiple methods, in same class	Extract common parts using DRY mechanisms like blocks and <code>yield</code> (Section 2.3), extracting helper methods (Section 9.6), using Template or Strategy design pattern (Section 11.4)
Parallel inheritance hierarchy	Nearly the same code repeated with subtle changes in different classes that inherit from different ancestors; for example, numerous pieces of code using slightly different combinations of data or behavior	Extract commonality into its own class and delegate to that class (Section 11.7). If classes with different ancestors need the functionality, try extracting it into a module that can be mixed in

Figure 11.24: Some smells are relatively easily fixed by a local modification. These are excerpted from Fowler's *Refactoring, Ruby Edition* (Fields et al. 2009).

W. Cunningham. Portland pattern repository, 2013. URL <http://c2.com/ppr/>.

J. Fields, S. Harvie, M. Fowler, and K. Beck. *Refactoring: Ruby Edition*. Addison-Wesley Professional, 2009. ISBN 0321603508.

E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN 0201633612.

R. Griffiths. HCI design patterns, 2013. URL <http://www.hcipatterns.org/patterns>.

R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN 9780132350884.

S. Metz. *Practical Object-Oriented Design in Ruby: An Agile Primer (Addison-Wesley Professional Ruby)*. Addison-Wesley Professional, 2012. ISBN 0321721330. URL <http://poodr.com>.

J. Noble and R. Johnson. Design patterns library, 2013. URL <http://hillside.net/patterns>.

R. Olsen. *Design Patterns in Ruby*. Addison-Wesley Professional, 2007. ISBN 9780321490452.

C. Pytel and T. Saleh. *Rails AntiPatterns: Best Practice Ruby on Rails Refactoring (Addison-Wesley Professional Ruby Series)*. Addison-Wesley Professional, 2010. ISBN 9780321604811.

A. Toxboe. UI patterns, 2013. URL <http://ui-patterns.com/>.

Notes

¹<http://cleancoder.com>

²<http://martinfowler.com/eaaCatalog>

³<http://ui-patterns.com>

⁴http://en.wikipedia.org/wiki/Unified_Modeling_Language

⁵http://en.wikipedia.org/wiki/Unified_Modeling_Language

⁶<http://cruise.site.uottawa.ca/umple/>

⁷<http://try.umple.org>

⁸http://en.wikipedia.org/wiki/Python_syntax_and_semantics#Decorators

9<http://nokogiri.org>10<http://www.ruby-doc.org/stdlib-1.9.3/libdoc/observer/rdoc/Observable.html>11<http://www.martinfowler.com/articles/designDead.html>12<https://youtu.be/np0G0mkxuio>13<http://martinfowler.com/refactoring/catalog>

12

Dev/Ops: Deployment, Performance, Reliability, and Practical Security

Ronald Rivest (1947–), Adi Shamir (1952–), and Leonard Adleman (1945–) received the 2002 Turing Award for making public-key cryptography useful in practice. In the eponymous RSA algorithm, the security properties of keypairs are based on the difficulty of factoring large integers and performing modular exponentiation, that is, determining m such that $C = m^E \bmod N$.



My response was “Congratulations, Ron, that should work.”

—Len Adleman, reacting to Ron Rivest’s encryption proposal, 1977

12.1 From Development to Deployment	373
12.2 Three-Tier Architecture	375
12.3 Responsiveness, Service Level Objectives, and Apdex	378
12.4 Releases and Feature Flags	382
12.5 Monitoring and Finding Bottlenecks	385
12.6 Improving Rendering and Database Performance With Caching	387
12.7 Avoiding Abusive Database Queries	391
12.8 CHIPS: Exploiting Caching and Indices	394
12.9 Security: Defending Customer Data in Your App	394
12.10 The Plan-And-Document Perspective on Operations	400
12.11 Fallacies and Pitfalls	402
12.12 Concluding Remarks: Beyond PaaS Basics	406

Prerequisites and Concepts

The big concept of this chapter is how to avoid the following headaches when your app is deployed: crashes, becoming unresponsive if it experiences a surge in popularity, or compromising customer data. Such non-functional characteristics can be more important than functional features since such headaches can drive users away. While deploying on a Platform-as-a-Service (PaaS) simplifies addressing these challenges, the techniques you must use to make your app take best advantage of PaaS are essentially the same ones you will need if you abandon PaaS.

Concepts:

For SaaS using an Agile lifecycle:

- Most SaaS servers follow the ***three-tier architecture*** pattern, which separates the responsibilities of different SaaS server components (Web server tier, application server tier, and database or storage tier) and enables ***horizontal scaling*** to accommodate millions of users.
- Deployment and maintenance headaches are greatly simplified by deploying your app on a ***Platform as a Service*** (PaaS), which manages much of the administration and scaling for you, including some aspects of horizontal scaling.
- Because the database cannot benefit from horizontal scaling in a 3-tier architecture as readily as the Web server or app server, database capacity is often the reason an app has to abandon the PaaS solution. But you can maximize your benefit from a PaaS-hosted database by using techniques that ease the load on the database, such as ***caching***, creating indices, and avoiding unnecessary and expensive database queries.
- Even with PaaS and horizontal scaling, the difficult ***performance*** challenge for 3-tier apps is ***latency***, which can be helped by ***overprovisioning*** in limited cases. The ***Apdex*** metric is a standard measure to see if an app is meeting its ***Service Level Objective (SLO)***.
- ***Releases*** are more challenging in SaaS since you normally need to deploy new versions without first taking down old ones. ***Feature flags*** make it easier to quickly deploy and remove new features should the need arise.
- ***Security*** can be enhanced by following the ***principles of least privilege*** and ***fail-safe defaults***, which limit access to assets on a “need-to-know” basis, and the ***principle of psychological acceptability***, which states that the user interface must not be more difficult with protection features than without them.
- ***Defensive programming*** anticipates flaws before they appear and can lead to systems that are both more reliable and more secure.

For Plan-and-Document lifecycles:

- Performance is just a possible non-functional requirement.

- Releases are less frequent, larger events than in Agile.
- The *Mean Time Between Failures* (MTBF) is a holistic measure of uptime, including errors by the hardware, the software, and the operators. Reducing *Mean Time to Repair* (MTTR) can be just as effective in improving uptime as trying to increase MTBF, and MTTR is easier to measure than MTBF.
- Security can be enhanced by making the system robust against software flaws that leave it open to attacks, such as *buffer overflows*, *arithmetic overflows*, and *data races*.

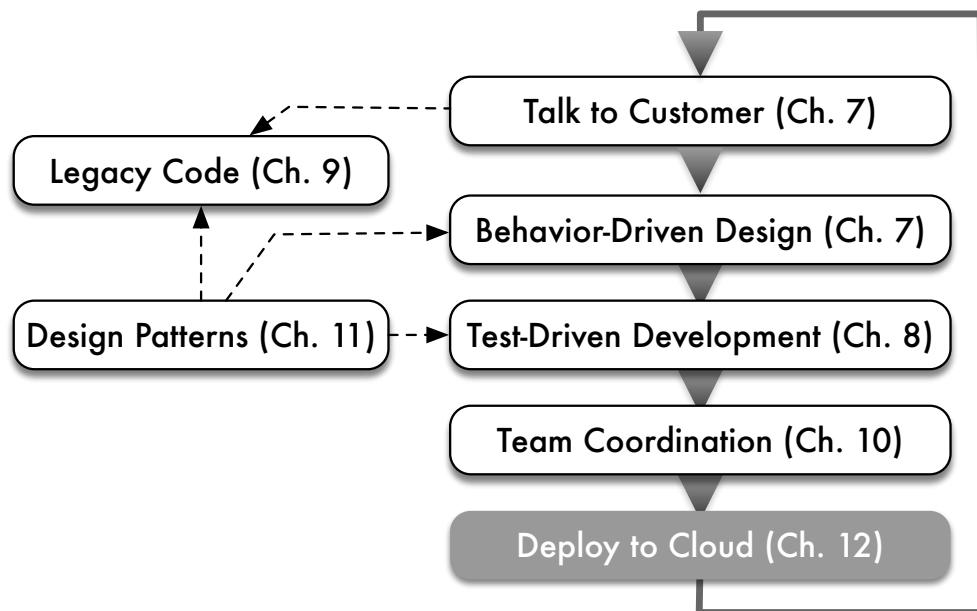


Figure 12.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter covers deploying the app into the cloud so that the customer can evaluate this Agile iteration.

12.1 From Development to Deployment

Users are a terrible thing. Systems would be infinitely more stable without them.

—Michael Nygard, *Release It!* (Nygard 2007)

The moment a SaaS app is deployed, its behavior changes because it has actual users. If it is a public-facing app, it is open to malicious attacks as well as unexpected success, but even private apps such as internal billing systems must be *designed for deployability and monitorability* in order to ensure smooth deployment and operations. In addition, there are necessarily differences between the environment in which you develop and test your app and the environment in which it is deployed, and these differences can sometimes result in unexpected (and usually undesirable) changes in app behavior between development and deployment that your tests didn't catch. Fortunately, as Figure 12.1 reminds us, deployment is part of every iteration in the Agile lifecycle—indeed, many Agile SaaS companies deploy several times *per day*—so you will quickly become practiced in “routine” deployments.

SaaS deployment is much easier than it used to be. Just a few years ago, SaaS developers had to learn quite a bit about system administration in order to manage their own production servers. For small sites they were typically hosted on shared Internet Service Providers (“managed-hosting ISP”), on virtual machines running on shared hardware (Virtual Private Server or VPS), or on one or more dedicated computers physically located at the ISP’s datacenter (“hosting service”). Today, the horizontal scaling enabled by cloud computing (Section 12.2) has given rise to companies like Heroku that provide a **Platform as a Service** (PaaS): a curated software stack ready for you to deploy your app, with much of the administration and scaling responsibility managed for you, making deployment much more developer-friendly. PaaS providers may either run their own datacenters or, increasingly, rely

on lower-level Infrastructure as a Service (IaaS) providers such as the Amazon public cloud, as Heroku does.

For early-stage and many mature SaaS apps, PaaS is now the preferred way to deploy: basic scaling issues and performance tuning are handled for you by professional SaaS administrators who are more experienced at operations than most developers. Of course, when a site becomes large enough or popular enough, its technical needs may outgrow what PaaS can provide, or economics may suggest bringing operations “in-house”, which as we will see is a major undertaking. Therefore one goal of this chapter is to help your app stay within the PaaS-friendly usage tier for as long as possible. Indeed, if your app is internally-facing, so that its maximum user base is bounded and it runs in a more protected and less hostile environment than public-facing apps, you may have the good fortune to stay in that tier indefinitely.

In general, though, performance, reliability, and security are systemwide concerns that must be constantly reviewed, rather than problems to be solved once and then set aside. While PaaS helps address some of these concerns, others must be confronted directly by the app developers, or PaaS cannot help you. For example, as we will see, a key to managing the growth of your app is controlling the demands placed on the database, which is harder to scale horizontally. One insight of this chapter is that the performance and security problems you face are the same for both small- and large-scale SaaS apps, but the solutions differ because PaaS providers can be very helpful in solving some of the problems, saving you the work of a custom-built solution.

Notwithstanding the title of this chapter, the terms **performance** and **security** are often overused and ill-defined. Here is a more focused list of key operational criteria we will address.

- Responsiveness: how long do most users wait before the app delivers a useful response? (Section 12.3)
- Release management: how can you deploy or upgrade your app “in place” without reducing availability and responsiveness? (Section 12.4)
- Availability: what percentage of the time is your app correctly serving requests? (Section 12.3)
- Scalability: as the number of users increases, either gradually and permanently or as a one-time surge of popularity, can your app maintain its steady-state availability and responsiveness without increasing the operational cost per user? As Section 12.2 explains, three-tier SaaS apps on cloud computing have excellent *potential* horizontal scalability, but good design alone doesn’t guarantee that your app will scale (though poor design guarantees that it won’t). Caching (Section 12.6) and avoiding abuse of the database (Section 12.7) can help.
- Privacy: is important customer data accessible only to authorized parties, such as the data’s owner and perhaps the app’s administrators?
- Authentication: can the app ensure that a given user is who they claim to be, by verifying a password or using third-party authentication such as Facebook Login or OpenID in such a way that an impostor cannot successfully impersonate another user without having obtained the user’s credentials?

- Data integrity: can the app prevent customer data from being tampered with, or at least detect that tampering has occurred or that data may have been compromised?

The first three items in the above list might be collectively referred to as *performance stability*, while the last three collectively address *security*, which Section 12.9 discusses.

Lastly, the opening of this section warned about unexpected problems due to differences between development and production environments, or a lack of so-called *development-production parity*. Because it is impossible to foresee and test every such possibility before deployment, the alternative is to make deployment itself as agile as possible by relying heavily on automation. If deployment of a new version of the software causes unexpected problems, how easily can you “roll back” to the previous version? If a schema migration or data migration causes unexpected problems, can you easily restore a database snapshot taken immediately before the migration was run? A good rule of thumb for managing your production environment is to assume that any task you need to do in that environment will fail the first time and will have to be repeated from scratch. If the task was automated, you just type one line to rerun the script. If the task consisted of manual steps, you must repeat the steps, which takes longer and is particularly error-prone when you’re operating under the psychological pressure caused by having broken the production server. More so than with any other aspect of SaaS, when it comes to deployment, *automate everything*.



Summary

- High availability and responsiveness, release management without downtime, and scalability without increasing per-user costs are three key *performance stability* concerns of SaaS apps, and defending your customers’ data is the app’s key security concern.
- Good PaaS providers can provide infrastructure mechanisms to automatically handle some of the details of maintaining performance stability and security, but as a developer you must also address these concerns in various aspects of your app’s design and implementation, using mechanisms we will discuss in this chapter.
- Compared to shrink-wrapped software, SaaS developer-operators are typically much more involved with deploying, releasing, and upgrading their apps and monitoring them for problems with performance or security.



rake is the Rails tool designed to automate deployment tasks that require access to the app’s classes, schema, and so on.

Self-Check 12.1.1. Which aspects of application scalability are not automatically handled for you in a PaaS environment?

- ◊ If your app “outgrows” the capacity of the largest database offered by the PaaS provider, you will need to manually build a solution to split it into multiple distinct databases. This task is highly app-specific so PaaS providers cannot provide a generic mechanism to do it. ■

12.2 Three-Tier Architecture

So far we have treated the overall SaaS server as a “black box”: whereas Chapter 3 considered the software architecture of SaaS and SOA generally, and Chapter 4 examined the software

architecture of SaaS applications using patterns such as Model–View–Controller, we have been oblivious to how the other parts of the server are organized. For example, SaaS apps use HTTP to communicate, yet you haven’t had to write any of the code that handles the details of such communication. We also have largely ignored how SaaS software components are deployed on actual hardware in production. While PaaS hides much of the hardware details from you, a high-level understanding of the hardware architecture is key to making good decisions about scalability in your software architecture. To that end, this section explains how SaaS servers typically follow a ***three-tier architecture***, how the logical boundaries separating those tiers are in place whether you run a development server on your own computer or deploy on a public cloud facility such as Heroku, how the components in the three-tier architecture typically map onto the cloud hardware, and what the resulting implications are for scaling up a SaaS app, that is, allowing it to serve more and more users.

Figure 12.2 shows the canonical three-tier architecture. The *presentation tier* usually consists of an *HTTP server* (or simply “**Web server**”), which accepts HTTP requests from the outside world (i.e. users) and handles the serving of static assets such as images, stylesheets, files of JavaScript code, and so on.

The web server forwards requests for dynamic content to the ***logic tier***, where your actual application runs. The application is typically supported by an ***application server*** whose job is to hide the low-level mechanics of these HTTP interactions from the app writer. We’ve been using the Rack application server, which ships with the Rails framework. If you were writing in PHP, Python, or Java, you would use an application server that handles code written in frameworks that use those languages, such as Django for Python or Node.js for JavaScript.

LAMP. Early SaaS sites were created using the Perl or PHP scripting languages, whose availability coincided with the early success of Linux, an open-source operating system, and MySQL, an open-source database. Thousands of sites are still powered by the *LAMP Stack*—Linux, Apache, MySQL, and PHP or Perl.

Cowboy, Heroku’s custom-built web server, is written in **Erlang**, a language optimized for “event driven” apps such as serving Web content. Separate tiers mean that each tier’s software can use the best tool for the job.

Finally, since HTTP is stateless (Chapter 3), application data that must remain stored across HTTP requests, such as users’ login and profile information, is stored in the ***persistence tier***. Popular choices for the persistence tier have traditionally been databases such as the open-source MySQL or PostgreSQL, although prior to their proliferation, commercial databases such as Oracle or IBM DB2 were also popular choices.

The “tiers” in the three-tier model are *logical* tiers. On a site with little content and low traffic, the software in all three tiers might run on a single physical computer: when you run `rails server` to do local development, the simple single-user WEBrick Web server fulfills the role of the presentation tier, and a single-user database called SQLite, which stores its information directly in files on your local computer, serves for persistence. In production, it’s more common for each tier to span one or more physical computers and to use highly specialized software. As Figure 12.2 shows, in a typical site, incoming HTTP requests are directed to one of several Web servers, possibly Apache¹ or Microsoft Internet Information Server², either of which can be deployed on hundreds of computers efficiently serving many copies of the same site to millions of users. When a Web server receives a request for your app, it selects one of several available application servers to handle dynamic-content generation, allowing computers to be added or removed from each tier as needed to handle demand.

However, as the Fallacies and Pitfalls section explains, making the persistence tier “shared-nothing” is much more complicated. Figure 12.2 shows one approach: a ***primary/replica*** or primary/secondary configuration, used when the database is read much more frequently than it is written. In this approach, any replica can perform reads, only the primary can perform writes, and the primary updates the replicas with the results of writes as quickly as possible. However, in the end, this and other techniques only postpone the scaling problem

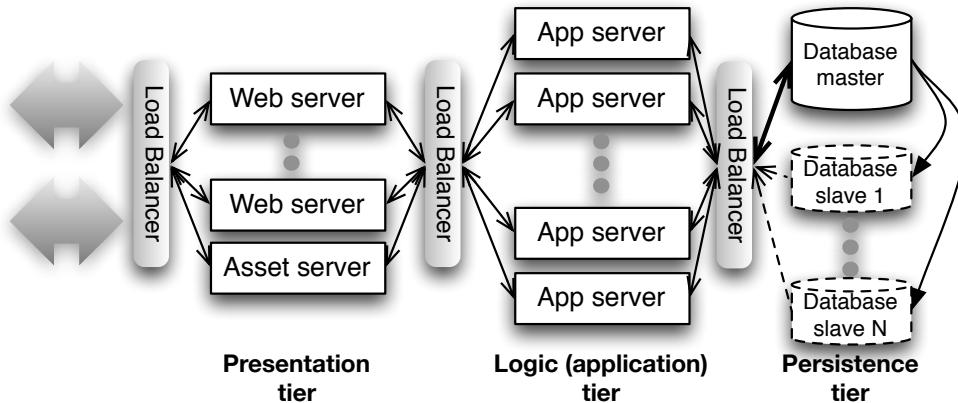


Figure 12.2: The 3-tier *shared-nothing* architecture, so called because entities within a tier generally do not communicate with each other, allows adding computers to each tier independently to match demand. *Load balancers*, which distribute workload evenly, can be either hardware appliances or specially-configured Web servers. The statelessness of HTTP makes *shared-nothing* possible: since all requests are independent, any server in the presentation or logic tier can be assigned to any request. However, scaling the persistence tier is much more challenging, as the text explains.

rather than solving it. As one of Heroku's³ founders wrote:

A question I'm often asked about Heroku is: "How do you scale the SQL database?" There's a lot of things I can say about using caching, sharding, and other techniques to take load off the database. But the actual answer is: we don't. SQL databases are fundamentally non-scalable, and there is no magical pixie dust that we, or anyone, can sprinkle on them to suddenly make them scale.

—Adam Wiggins, Heroku, in 2009

Summary

- The three-tier architecture includes a presentation tier, which renders views and interacts with the user; a logic tier, which runs SaaS app code; and a persistence tier, which stores app data.
- HTTP's statelessness allows the presentation and logic tiers to be ***shared-nothing***, so cloud computing can be used to add more computers to each tier as demand requires. However, the persistence tier is harder to scale.
- Depending on the scale (size) of the deployment, more than 1 tier may be hosted on a single computer, or a single tier may require many computers.

■ Elaboration: Why Databases?

While the earliest Web apps sometimes manipulated files directly for storing data, there are two reasons why databases overwhelmingly took over this role very early. First, databases have historically provided high *durability* for stored information—the guarantee that once something has been stored, unexpected events such as system crashes or transient data corruption won’t cause data loss. For a Web app storing millions of users’ data, this guarantee is critical. Second, databases store information in a structured format—in the case of ***relational databases***, by far the most popular type, each kind of object is stored in a table whose rows represent object instances and whose columns represent object properties. This organization is a good fit for the structured data that many Web apps manipulate. Interestingly, today’s largest Web apps, such as Facebook, have grown so far beyond the scale for which relational databases were designed that they are being forced to look at alternatives to the long-reigning relational database.

Self-Check 12.2.1. *Explain why cloud computing might have had a lesser impact on SaaS if most SaaS apps didn’t follow the shared-nothing architecture.*

- ◊ Cloud computing allows easily adding and removing computers while paying only for what you use. The shared-nothing architecture takes advantage of this ability to rapidly “absorb” new computers into a running app and “release” them when no longer needed. ■

Self-Check 12.2.2. *Which tier(s) of three-tier SaaS apps can be scaled just by adding more computers and why?*

- ◊ The presentation and logic tiers. Because neither HTTP (Web) servers nor app servers maintain any of the state associated with user sessions, any computer in those tiers can in principle satisfy any user’s request. ■

12.3 Responsiveness, Service Level Objectives, and Apdex

Speed is a feature.

—Adam De Boor, Gmail software engineer, Google

Performance is a feature.

—Jeff Atwood, co-founder of StackOverflow

The best performance improvement is the transition from the nonworking state to the working state.

—John Ousterhout, designer of *magic* and *Tcl/Tk*

As we learned in Section 1.7, ***availability*** refers to the fraction of time your site is available and working correctly. For example, Gmail guarantees⁴ an availability of “three nines” or 99.9% during any given month for its enterprise customers. (Nygard wryly notes (Nygard 2007) that less-disciplined sites provide closer to “two eights” or 88.0%).

Why might your app be unavailable? For one thing, it may have crashed because of an unexpected error. While most PaaS services automatically restart a crashed app, restarting can induce delays and harm availability. One way to improve the reliability of software is to make it more robust. ***Defensive programming*** is a philosophy that tries to anticipate potential software flaws and write code to handle them. Here are three examples:

- *Check input values.* A common cause of problems is for the user to input values that the developer doesn't expect. Checking that the input is in a reasonable range for individual values, that it is not too big for a series of data, and that the collection of inputs are logically consistent can reduce the chances of outages.
- *Check input data type.* Another mistake users can make is to enter an unexpected type of data in response to a query. Making sure the user enters a valid type of data increases the chances of success for the app.
- *Catch exceptions.* Modern programming languages offer the ability to execute code when exceptions occur, such as arithmetic overflow. Offering code that can catch any exception increases the chances of the app continuing to run well even when unexpected events occur.



Another availability challenge is a bug that leads to outages but only appears after a long time or under heavy load. A classic example is a resource leak: a long-running process eventually runs out of a resource, such as memory, because it cannot reclaim 100% of the unused resource due to either an application bug or the inherent design of a language or framework. **Software rejuvenation** is a long-established way to alleviate a resource leak: the Apache web server runs a number of identical worker processes, and when a given worker process has “aged” enough, that process stops accepting requests and dies, to be replaced by a fresh worker. Since only one worker ($1/n$ of total capacity) is “rejuvenated” at a time, this process is sometimes called *rolling reboot*, and most PaaS platforms employ some variant of it.

Overprovisioning is often used in anticipation of crash recovery and rolling reboot. The idea is to provide more servers in a tier at any given time than you think you'll need. For example, by deploying $n + 1$ servers in a tier, temporarily losing one server degrades performance by only $1/n$. Good values for n can sometimes be determined empirically by monitoring, as the rest of this chapter describes. However, at large scale, systematic overprovisioning is both economically unattractive and may be insufficient by itself. For example, in an app whose database queries are poorly constructed, the database will quickly become the bottleneck, and as Section 12.2 reminds us, databases are generally *not* amenable to shared-nothing horizontal scaling. In such a situation, overprovisioning the other tiers won't help. The lesson is that in the end, there's no substitute for a design that is free of gratuitous bottlenecks to scalability. Later in this chapter we identify some common bottlenecks and how to avoid them.

Of course, it's not much good if your app is technically “available” but so sluggish that users don't want to use it. **Responsiveness** is the perceived delay between when a user takes an action such as clicking on a link and when the user perceives a response, such as new content appearing on the page. Technically, responsiveness has two components: **latency**, the initial delay to start receiving new content, and **throughput**, the time it takes for all the content to be delivered. As recently as the mid-1990s, many home users connected to the Internet using telephone modems that took 100 ms (milliseconds) to deliver the first packet of information and show part of the Web page. Telephone modems could sustain at most 56 Kbps (56×10^3 bits per second), so loading a complete Web page or image 50 KBytes (400 KBits) in size could take more than eight seconds. Since today's home customers increasingly use broadband connections whose throughput is 1–50 Mbps, responsiveness for Web pages is dominated by latency rather than throughput.

Since responsiveness has such a large effect on user behavior, SaaS operators carefully monitor the responsiveness of their sites. Of course, in practice, not every user interaction with the site takes the same amount of time, so evaluating performance requires appropriately characterizing a distribution of response times. Consider a site on which 8 out of 10 requests complete in 100 ms, 1 out of 10 completes in 250 ms, and the remaining 1 out of 10 completes in 850 ms. If the user satisfaction threshold T for the latency of this site is 200 ms, it is true that the *average* response time of $(8(100) + 1(250) + 1(850))/10 = 190$ ms is below the satisfaction threshold. But on the other hand, 20% of requests (and therefore, up to 20% of users) are receiving unsatisfactory service. Two definitions are used to measure latency in a way that makes it impossible to ignore the bad experience of even a small number of users:

SLA vs. SLO: A *service level agreement* (SLA) is a contract between a service provider and its customers that provides for customer consideration if the SLO is not met.

- A **service level objective** (SLO) usually takes the form of a quantitative statement about the quantiles of the latency distribution over a time window of a given width. For example, “95% of requests within any 5-minute window should have a latency below 100 ms.” In statistical terms, the 95th quantile of the latency distribution must not exceed 100 ms.
- The **Apdex** score (Application Performance Index) is an open standard⁵ that computes a simplified SLO as a number between 0 and 1 inclusive representing the fraction of satisfied users. Given a user satisfaction threshold latency T selected by the application operator, a request is *satisfactory* if it completes within time T , *tolerable* if it takes longer than T but less than $4T$, and *unsatisfactory* otherwise. The Apdex score is then $(\text{Satisfactory} + 0.5(\text{Tolerable})) / (\text{Number of samples})$. In the example above, the Apdex score would be $(8 + 0.5(1))/10 = 0.85$.

Google believes⁶ that because many aspects of time-to-glass are independent of the specific service, it is even more important for the service to be responsive, so that getting a response from any Google service is no slower than contacting the service to begin with.

Of course, the total response time perceived by the users includes many factors beyond your SaaS app’s control. It includes DNS lookup, time to set up the TCP connection and send the HTTP request to the server, and Internet-induced latency in receiving a response containing enough content that the browser can start to draw something (so-called “time to glass,” a term that will soon seem as quaint as “counterclockwise”). Especially when using curated PaaS, SaaS developer/operators have the most control over the code paths in their own apps: routing and dispatch, controller actions, model methods, and database access. We will therefore focus on measuring and improving responsiveness in those components.

For small sites, a perfectly reasonable way to mitigate latency is to *overprovision* (provide excess resources relative to steady-state) at one or more tiers, as Section 12.2 describes for the presentation and logic tiers. A few years ago, overprovisioning meant purchasing additional hardware that might sit idle, but pay-as-you-go cloud computing lets you “rent” the extra servers for pennies per hour only when needed. Indeed, technologies like RightScale⁷ offer just this service on top of Amazon EC2.

As we will see, a key insight that helps us is that *the same problems that push us out of the “PaaS-friendly” tier are the ones that will hinder scalability of our post-PaaS solutions*, so understanding what kinds of problems they are and how to solve them will serve you well in either situation.

What are the thresholds for user satisfaction on responsiveness? A classic 1968 study from the human-computer interaction literature (Miller 1968) found three interesting thresholds: if a computer system responds to a user action within 100 ms, it’s perceived as instantaneous; within 1 second, the user will still perceive a cause-and-effect connection between their action and the response, but will perceive the system as sluggish; and after about 8 seconds, the user’s attention drifts away from the task while waiting for a response. Surprisingly,

more than thirty years later, a scholarly study in 2000 (Bhatti et al. 2000) and another by independent firm Zona Research in 2001 affirmed the “eight second rule.” While many believe that a faster Internet and faster computers have raised users’ expectations, the eight-second rule is still used as a general guideline. New Relic, whose monitoring service we introduce later, reported in March 2012 that the average page load for all pages they monitor worldwide is 5.3 seconds and the average Apdex score is 0.86.

Summary

- Availability measures the percentage of time over a specified window that your app is correctly responding to user requests. Availability is usually measured in “nines” with the gold standard of 99.999% (“five nines”, corresponding to five minutes of downtime per year) set by the US telephone network and rarely matched by SaaS apps.
- While PaaS services usually restart crashed SaaS apps, the time required to do so can harm availability. App developers can mitigate this harm using **defensive programming**, which adds code to handle common classes of potential flaws before they cause the app to crash. PaaS providers can mitigate it using **software rejuvenation**, which proactively restarts members of a set of identical processes on a rotating schedule to neutralize resource leaks.
- Responsiveness measures how “snappy” an interactive app feels to users. Given today’s high-speed Internet connections and fast computers, responsiveness is dominated by latency. Service Level Objectives (SLOs) quantify responsiveness goals with statements such as “99% of requests within any 5-minute window should have a latency below 100 ms.”
- Overprovisioning helps improve latency by making more computers available to handle requests in a given tier and helps with availability by dealing gracefully with server crashes. However, at large scale, systematic overprovisioning is both economically unattractive and insufficient by itself.
- The Apdex score is a simple SLO measure between 0.0 and 1.0 in which a site gets “full credit” for requests that complete within a site-specific latency threshold T , “half credit” for requests that complete within $4T$, and no credit for requests taking longer than that.
- The problems that threaten availability and responsiveness are the same whether we use PaaS or not, but it’s worth trying to stay within the PaaS tier because it provides machinery to help mitigate those problems. Part of “scaling gracefully” is avoiding problems that lead to intrinsic scalability bottlenecks, some of which we discuss in the rest of this chapter.

Self-Check 12.3.1. *For a SaaS app to scale to large numbers of users, it must maintain its _____ and _____ as the number of users increases, without increasing the _____.*

◊ Availability; responsiveness; cost per user ■

Self-Check 12.3.2. *True or False: From the perspective of responsiveness, faster is always*

better.

- ◊ False. Faster than 100 ms is not perceptible to people, and people abandon sites only when responsiveness slows to 8 seconds or worse. ■

12.4 Releases and Feature Flags

As we discussed way back in Section 1.2, prior to SaaS, software releases were major and infrequent milestones after which product maintenance responsibility passed largely to the Quality Assurance or Customer Service department. In contrast, Many Agile companies deploy new versions frequently (sometimes several times *per day*) and the developers stay close to operations and to customer needs.



In Agile development, making deployment a *non-event* requires complete automation, so that typing one command triggers all the actions to deploy a new version of the software, including cleanly aborting the deploy without modifying the released version if anything goes wrong. As with iteration-based TDD and BDD, by deploying frequently you become good at it, and by automating deployment you ensure that it's done consistently every time.

Although deployment is a non-event, there is still a role for release milestones: they reassure the customer that new work is being deployed. For example, a customer-requested feature may require multiple commits to implement, each of which may include a deployment, but the overall feature remains “hidden” in the user interface until all changes are completed. “Turning on” the feature would be a useful release milestone. For this reason, many continuous-deployment workflows assign distinct and often whimsical labels to specific release points (such as “Bamboo” and “Cedar” for Heroku’s software stacks), but just use the Git commit-id to identify deployments that don’t include customer-visible changes.

Of course, deployment can only be successful if the app is well tested and stable in development. Although we’ve already focused heavily on testing in this book, making deployment a true non-event requires meeting two additional challenges: deployment testing and incremental feature rollout.

Beyond traditional CI, deployment testing must account for differences between the development and production environments, such as the type of database used or the need for JavaScript-intensive apps to work correctly on a variety of browser versions. Deployment testing should also test the app in ways it was *never* meant to be used—users submitting nonsensical input, browsers disabling cookies or JavaScript, miscreants trying to turn your site into a distributor of **malware** (as we describe further in Section 12.9)—and ensuring that it survives those conditions without compromising customer data or responsiveness.

The second challenge is the rollout of complex features that may require several code pushes, especially features that require database schema changes. In particular, a challenge arises when the new code does not work with the old schema and vice-versa. To make the example concrete, suppose RottenPotatoes currently has a `moviegoers` table with a `name` column, but we want to change the schema to have separate `first_name` and `last_name` columns instead. If we change the schema before changing the code, the app will break because methods that expect to find the `name` column will fail. If we change the code before changing the schema, the app will break because the new methods will look for `first_name` and `last_name` columns that don’t exist yet.

We could try to solve this problem by deploying the code and migration **atomically**: take the service offline, apply the migration to perform the schema change and copy the data from the existing `name` column into the two new columns, and bring the service back online. This

<https://gist.github.com/0131ef743244fae7b9ac5fa466a7e582>

```

1  /* in code paths for functionality that searches the database: */
2  if (featureflag is on)
3      results = union(query using old schema, query using new schema)
4  else /* featureflag is off */
5      results = (query using old schema)
6  end
7
8  /* in code paths that write to the database */
9  if (featureflag is on)
10     if (data to be written is still using old schema)
11         (convert existing record from old to new schema)
12         (mark record as converted)
13     end
14     (update data according to new schema)
15 else
16     (update data according to old schema)
17 end

```

Figure 12.3: Pseudocode for using a feature flag to help migrate data from an older to a newer schema incrementally. After an initial migration creates any necessary new schema elements, each function that reads or updates the affected data implements two code paths, corresponding to the older and newer schema respectively. If the feature flag is off, only the old code path is ever used; but when the feature flag is on, the new code path contributes results to searches and causes old data to be incrementally migrated to the new schema. Once all data has been migrated, a subsequent migration and code push can remove unused columns or tables from the old schema and remove the alternate code paths protected by the feature flag.

is the simplest solution, but may cause unacceptable unavailability: a complex migration on a database of hundreds of thousands of rows can take tens of minutes or even hours to run.

The second option is to split the change across multiple deployments using a *feature flag*—a configuration variable whose value can be changed *while the app is running* to control which code paths in the app are executed. Notice that each step in Figure 12.3 is nondestructive: as we did with refactoring in Chapter 9, if something goes wrong at a given step, the app is still left in a working intermediate state. Figure 12.3 illustrates schematically how to do this:

1. Create a migration that makes *only* those changes to the schema that *add* new tables or columns, including a column indicating whether the current record has been migrated to the new schema or not.
2. Create version $n+1$ of the app in which every code path affected by the schema change is split into two code paths, of which one or the other is executed based on the value of a *feature flag*. Critical to this step is that correct code will be executed regardless of the feature flag’s value at any time, so the feature flag’s value can be changed without stopping and restarting the app; typically this is done by storing the feature flag in a special database table.
3. Deploy version $n+1$, which may require pushing the code to multiple servers, a process that can take several minutes.
4. Once deployment is complete (all servers have been updated to version $n + 1$ of the code), while the app is running set the feature flag’s value to True. Essentially, each record will be migrated to the new schema the next time it’s modified for any reason. If you wanted to speed things up, you could also run a low-traffic background job that opportunistically migrates a few records at a time to minimize the additional load on the app, or migrates many records at a time during hours when the app is lightly

loaded, if any. If something goes wrong at this step, turn off the feature flag; the code will revert to the behavior of version n , since the new schema is a proper superset of the old schema and the `before_save` callback is nondestructive (that is, it correctly updates the user's name in both the old and new schemata).

5. If all goes well, once all records have been migrated, deploy code version $n + 2$, in which the feature flag is removed and only the code path associated with the new schema remains.
6. Finally, apply a new migration that removes the old name column and the temporary migrated column (and therefore the index on that column).

What about a schema change that modifies a column's name or format rather than adding or removing columns? The strategy is the same: add a new column, remove the old column, and if necessary rename the new column, using feature flags during each transition so that every deployed version of the code works with both versions of the schema.

Besides handling destructive migrations, feature flags have other uses as well:

- Preflight checking: roll out a feature to a small percentage of users only, in order to make sure the feature doesn't break anything or have a negative effect on overall site performance.
- A/B testing: roll out two different versions of a feature to two different sets of users to see which version most improves user retention, purchases, and so on.
- Complex feature: sometimes the complete functionality associated with a feature may require multiple incremental deployment cycles such as the one described above. In this case, a separate feature flag can be used to keep the feature hidden from the user interface until 100% of the new feature code has been deployed.

The `rollout`⁸ gem supports the use of feature flags for all these cases.



Summary

- In general, SaaS deployment should be so automated and straightforward that it can be done frequently, up to several times a day, while remaining a non-event.
- One way to ensure a smooth deployment is to include additional deployment tests that must run before a deploy is attempted, to test differences between the development and production environments and to stress the app by deliberately simulating user misbehaviors.
- To perform a complex upgrade that changes both the app code and the schema, use a feature flag whose value can be changed while the app is running. The feature flag's value selectively enables certain code paths at runtime, and can be immediately turned off if a bug is observed after deployment. Otherwise, once all data has been incrementally migrated as a result of changing the feature flag's value, you can deploy a new migration and code push that eliminate the old code paths and schema elements.

■ Elaboration: Continuous Deployment

The extreme version of making deployment a non-event is **continuous deployment**, in which every successful CI run (continuous integration, discussed in Section 10.4) *automatically* triggers a deployment to staging or production. CD can result in multiple deployments per day, many of which include changes not visible to the customer that “build towards” a feature that will be unveiled at a release milestone.

Self-Check 12.4.1. Which of the following are appropriate places to store the value of a simple Boolean feature flag and why: (a) a YAML file in the app’s config directory, (b) a column in an existing database table, (c) a separate database table?

- ◊ The point of a feature flag is to allow its value to be changed at runtime without modifying the app. Therefore (a) is a poor choice because a YAML file cannot be changed without touching the production servers while the app is running. ■

12.5 Monitoring and Finding Bottlenecks

If you’re not monitoring it, it’s probably broken.

—variously attributed

Given the importance of responsiveness and availability, how can we measure them, and if they’re unsatisfactory, how can we identify what parts of our app need attention? *Monitoring* consists of collecting app performance data for analysis and visualization. In the case of SaaS, application performance monitoring (APM) refers to monitoring the Key Performance Indicators (KPIs) that *directly impact business value*. KPIs are by nature app-specific—for example, an e-tailer’s KPIs might include responsiveness of adding an item to a shopping cart and percentage of user searches in which the user selects an item that is in the top 5 search results.

There are various techniques for monitoring SaaS apps, and we can characterize them in terms of three axes:

1. Is the monitoring active or passive? In active monitoring, an external stimulus is deliberately applied to the app (even if the app would be otherwise idle) in order to ensure it’s working. In passive monitoring, no monitoring data is collected until some external user asks the app to do something.
2. Is the monitoring external or internal? External monitoring can only report on the behavior of an app as seen from the outside—for example, reporting that some types of requests take longer than other types. Internal monitoring can “hook” into the code of the app server or the app itself, so it can provide better *attribution*—how long did a request spend in each tier of the SaaS stack and in different parts of your app (for example, the controllers, the models, the database, or the view rendering)?
3. Is the monitoring focused on app performance or user behavior? For example, you might want to know what fraction of users who added an item to their shopping cart ended up purchasing the item, and what actions were taken instead by the users who didn’t end up completing the purchase. Such questions can be critical for a business even though they have little to do with performance. (Of course, performance monitoring might reveal the *reasons* some users don’t complete the purchase flow!)

Regardless of which combination of the above axes is provided by a particular monitoring solution, we must also address the issue of how the collected monitoring data is stored and how it is presented or reported to the app’s dev/ops team.

Before cloud computing and the prominence of SaaS and highly-productive frameworks, internal monitoring required installing programs that collected metrics periodically, manually inserting extra code into your app, or both. Today, the combination of hosted PaaS, Ruby’s dynamic language features, and well-factored frameworks such as Rails allows internal monitoring without modifying your app’s source code or installing software. For example, New Relic⁹ unobtrusively collects instrumentation about your app’s controller actions, database queries, and so on, making use of metaprogramming in Rails to do this without requiring changes to your app’s code. Because the data is sent back to New Relic’s SaaS site where you can view and analyze it, this architecture is sometimes called RPM for Remote Performance Monitoring. New Relic provides both internal passive monitoring and external active monitoring, in which you can set up HTTP “probe” requests with fixed URIs and test for the presence of particular strings in the apps’ responses.

Internal monitoring can also occur during development, when it is often called ***profiling***. New Relic and other monitoring solutions can be installed in development mode as well. How much profiling should you do? If you’ve followed best practices in writing and testing your app, it may be most productive to just deploy and see how the app behaves under load, especially given the unavoidable differences between the development and production environments, such as the lack of real user activity and the use of a development-only database such as SQLite rather than a highly tuned production database such as PostgreSQL. After all, with agile development, it’s easy to deploy incremental fixes such as implementing basic caching (Section 12.6) and fixing abuses of the database (Sections 12.7).

In external monitoring (sometimes called probing or active monitoring), a separate site makes live requests to your app to check availability and response time. Why would you need external monitoring given the detailed information available from internal monitoring that has access to your code? Internal monitoring may be unable to reveal that your app is sluggish or unavailable if the problem is due to factors other than your app’s code—for example, performance problems in the presentation tier or other parts of the software stack beyond your app’s boundaries. External monitoring, like an integration test, is a true end-to-end test of a limited subset of your app’s code paths as seen by actual users “from the outside.”

Once a monitoring tool has identified the slowest or most expensive requests, ***stress testing*** or ***longevity testing*** on a staging server can quantify the level of demand at which those requests become bottlenecks. The free and widely-used command line tool ***httpperf***, maintained by Hewlett-Packard Laboratories¹⁰, can simulate a specified number of users requesting simple sequences of URIs from an app and while recording metrics about the response times. Whereas tools like Cucumber let you write expressive scenarios and check arbitrarily complex conditions, ***httpperf*** can only follow simple sequences of URIs and only checks whether a successful HTTP response was received from the server. In a typical stress test, the test engineer will set up several computers running ***httpperf*** against the staging site and gradually increase the number of simulated users until some resource becomes the bottleneck.

Finally, monitoring can also help you understand your customers’ behavior:

- Clickstreams: what are the most popular sequences of pages your users visit?
- Think times/dwell times: how long does a typical user stay on a given page?



- Abandonment: if your site contains a flow that has a well-defined termination, such as making a sale, what percentage of users “abandon” the flow rather than completing it and how far do they get?

Google Analytics provides free basic analytics-as-a-service: you embed a small piece of JavaScript in every page on your site (for example, by embedding it on the default layout template) that sends Google Analytics information each time a page is loaded. To help you use this information, Google’s “Speed is a Feature”¹¹ site links to a breathtakingly comprehensive collection of articles about all the different ways you can speed up your SaaS apps, including many optimizations to reduce the overall size of your pages and improve the speed at which Web browsers can render them.

Summary

- As with testing, no single type of monitoring will alert you of all performance problems: use a combination of internal and external (end-to-end) monitoring.
- Hosted monitoring such as Pingdom and PaaS-integrated monitoring such as New Relic greatly simplify monitoring compared to the early days of SaaS.
- Stress testing and longevity testing can reveal the bottlenecks in your SaaS app and frequently expose bugs that would otherwise remain hidden.
- User-centric analytics can provide information about the behavior of users as they navigate your site, which can be extremely valuable business data even though unrelated to performance *per se*.

■ Elaboration: Request tracing

A finer-grained approach to internal monitoring is request tracing, which is used in conjunction with metric aggregation to pinpoint and diagnose slow requests. Request tracing follows a request through every software component in every tier and timestamping it along the way, often at every function call entry. Google has used request tracing to identify obstacles to keeping their massively-parallel systems highly responsive (Barroso and Dean 2012).

Self-Check 12.5.1. Which of the following key performance indicators (KPIs) would be relevant for Application Performance Monitoring: CPU utilization of a particular computer; completion time of slow database queries; view rendering time of 5 slowest views.

◊ Query completion times and view rendering times are relevant because they have a direct impact on responsiveness, which is generally a Key Performance Indicator tied to business value delivered to the customer. CPU utilization, while useful to know, does not directly tell us about the customer experience. ■

12.6 Improving Rendering and Database Performance With Caching

There are only two hard things in computer science: cache invalidation and naming things.

—Phil Karlton

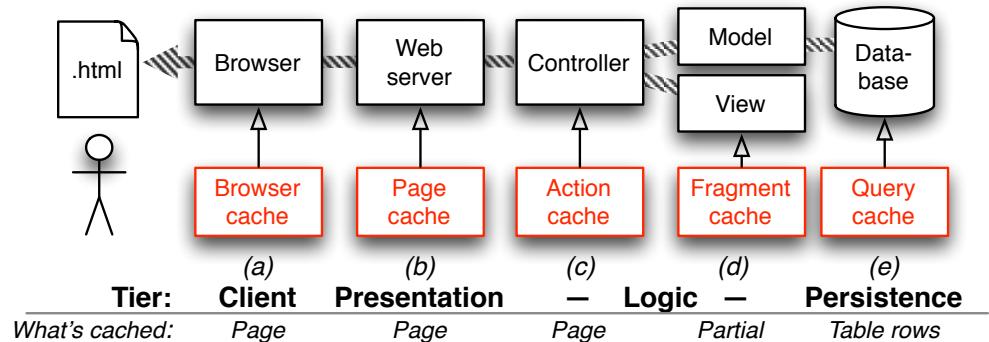


Figure 12.4: The goal of multiple levels of caching is to satisfy each HTTP request as close to the user as possible. (a) A Web browser that has previously visited a page can reuse the copy in its local cache after verifying with the server that the page hasn't changed. (b) Otherwise, the Web server may be able to serve it from the page cache, bypassing Rails altogether. (c) Otherwise, if the page is generated by an action protected by a before-filter, Rails may be able to serve it from the action cache without querying the database or rendering any templates. (d) Otherwise, some of the fragments comprised by the view templates may be in the fragment cache. (e) As a last resort, the database's query cache serves the results of recent queries whose results haven't changed, such as `Movie.all`.

The idea behind caching is simple: information that hasn't changed since the last time it was requested can simply be regurgitated rather than recomputed. In SaaS, caching can help two kinds of computation. First, if information needed from the database to complete an action hasn't changed, we can avoid querying the database at all. Second, if the information underlying a particular view or view fragment hasn't changed, we can avoid re-rendering the view (recall that rendering is the process of transforming Erb with embedded Ruby code and variables into HTML). In any caching scenario, we must address two issues:

1. **Naming:** how do we specify that the result of some computation should be cached for later reuse, and name it in a way that ensures it will be used only when that exact same computation is called for?
2. **Expiration:** how do we detect when the cached version is out of date (stale) because the information on which it depends has changed, and how do we remove it from the cache? The variant of this problem that arises in microprocessor design is often referred to as *cache invalidation*.

Figure 12.4 shows how caching can be used at each tier in the 3-tier SaaS architecture and what Rails entities are cached at each level. The simplest thing we could do is cache the entire HTML page resulting from rendering a particular controller action. For example, the `MoviesController#show` action and its corresponding view depend only on the attributes of the particular movie being displayed (the `@movie` variable in the controller method and view template). Figure 12.5 shows how to cache the entire HTML page for a movie, so that future requests to that page neither access the database nor re-render the HTML, as in Figure 12.4(b).

Of course, this is unsuitable for controller actions protected by before-filters, such as pages that require the user to be logged in and therefore require executing the controller filter. In such cases, changing `caches_page` to `caches_action` will still execute any filters but allow Rails to deliver a cached page without consulting the database or re-rendering views, as in Figure 12.4(c). Figure 12.7 shows the benefits of page and action caching for this simple

<https://gist.github.com/178cb5d9f527c2d6e5b045edd418550b>

```
1 # In Gemfile, include gems for page and action caching
2 gem 'actionpack-page_caching'
3 gem 'actionpack-action_caching'
4 gem 'rails-observers'
```

<https://gist.github.com/c14b8bbe5c710a570401d3c520bd093d>

```
1 class MoviesController < ApplicationController
2   caches_page :show
3   cache_sweeper :movie_sweeper
4   def show
5     @movie = Movie.find(params[:id])
6   end
7 end
```

<https://gist.github.com/436e15467dbc9b41d9a36d87d47c2481>

```
1 class MovieSweeper < ActionController::Caching::Sweeper
2   observe Movie
3   # if a movie is created or deleted, movie list becomes invalid
4   # and rendered partials become invalid
5   def after_save(movie); invalidate; end
6   def after_destroy(movie); invalidate; end
7   private
8   def invalidate
9     expire_action :action => ['index', 'show']
10    expire_fragment 'movie'
11  end
12 end
```

Figure 12.5: (Top) As of Rails 4, caching and observers are provided by separate gems, which must be included in the `Gemfile`. (Middle) Line 2 specifies that Rails should cache the result of the `:show` action. Action caching is implemented as a before-filter that checks whether a cached version should be used and an around-filter that captures and caches the rendered output, making it an example of the Decorator design pattern (Section 11.4). (Bottom) This “sweeper,” referenced by line 3 of the controller, uses the Observer design pattern (Section 11.7) to add ActiveRecord lifecycle hooks (Section 5.1) to expire any objects that might become stale as a result of updating a particular movie.

example. Note that in Rails page caching, the name of the cached object *ignores* embedded parameters in URIs such as `/movies?ratings=PG+G`, so parameters that affect how the page would be displayed should instead be part of the RESTful route, as in `/movies/ratings/PG+G`.

An in-between case involves action caching in which the main page content doesn’t change, but the layout does. For example, your `app/views/layouts/application.html.erb` may include a message such as “Welcome, Alice” containing the name of the logged-in user. To allow action caching to work properly in this case, passing `:layout=>false` to `caches_action` will result in the layout getting fully re-rendered but the action (content part of the page) taking advantage of the action cache. Keep in mind that since the controller action won’t be run, any such dynamic content appearing in the layout must be set up in a before-filter.

Page-level caching isn’t useful for pages whose content changes dynamically. For example, the list of movies page (`MoviesController#index` action) changes when new movies are added or when the user filters the list by MPAA rating. But we can still benefit from caching by observing that the index page consists largely of a collection of table rows, each of which depends only on the attributes of one specific movie. Indeed, that observation allowed us to factor out the code for one row into a partial, as Figure 5.1 (Section 5.1) showed. Figure 12.6 shows how a trivial change to that partial caches the rendered HTML fragment corresponding to each movie.

<https://gist.github.com/f7e6de0cc4bb0fa21e34da8432bd413e>

```

1 <% cache(movie) do %>
2   <div class="row">
3     <div class="col-8"> <%= link_to movie.title, movie_path(movie) %> </div>
4     <div class="col-2"> <%= movie.rating %> </div>
5     <div class="col-2"> <%= movie.release_date.strftime('%F') %> </div>
6   </div>
7 <% end %>
```

Figure 12.6: Compared to Figure 5.1 in Section 5.1, only two lines have been added, to “wrap” the rendered content with a call to `cache`. Rails will generate a name for the cached fragment based on the pluralized resource name and primary key, for example, `movies/23`.

A convenient shortcut provided by Rails is that if the argument to `cache` is an ActiveRecord object whose table includes an `updated_at` or `updated_on` column, the cache will auto-expire a fragment if its table row has been updated since the fragment was first cached. Nonetheless, for clarity, line 10 of the sweeper in Figure 12.5 shows how to explicitly expire a fragment whose name matches the argument of `cache` whenever the underlying `movie` object is saved or destroyed.

Unlike action caching, which avoids running the controller action at all, checking the fragment cache occurs *after* the controller action has run. Given this fact, you may already be wondering how fragment caching helps reduce the load on the database. For example, suppose we add a partial to the list of movies page to display the `@top_5` movies based on average review scores, and we add a line to the `index` controller action to set up the variable:

<https://gist.github.com/3727b1cf67ba36339f999c4be9623034>

```

1 <!-- a cacheable partial for top movies -->
2 <%- cache('top_moviegoers') do %>
3   <ul id="topmovies">
4     <%- @top_5.each do |movie| %>
5       <li> <%= movie.name %> </li>
6     <% end %>
7   </ul>
8 <% end %>
```

<https://gist.github.com/090b252fc85756ad441e38d5106a5906>

```

1 class MoviesController < ApplicationController
2   def index
3     @movies = Movie.all
4     @top_5 = Movie.joins(:reviews).group('movie_id').
5                   order("AVG(potatoes) DESC").limit(5)
6   end
7 end
```

Action caching is now less useful, because the `index` view may change when a new movie is added *or* when a review is added (which might change what the top 5 reviewed movies are). If the controller action is run before the fragment cache is checked, aren’t we negating the benefit of caching, since setting `@top_5` in lines 4–5 of the controller method causes a database query?

Surprisingly, no. In fact, lines 4–5 *don’t* cause a query to happen: they construct an object that *can* do the query if it’s ever asked for the result! This is called **lazy evaluation**, an enormously powerful programming-language technique that comes from the **lambda calculus** underlying functional programming. Lazy evaluation is used in Rails’ ActiveRelation (ARel) subsystem, which is used by ActiveRecord. The actual database query doesn’t happen until `each` is called in line 4 of the partial, because that’s the first time the ActiveRelation object is asked to produce a value. But since that line is inside the `cache` block starting on line 2, if



No cache	Action cache	Speedup vs. no cache	Page cache	Speedup vs. no cache	Speedup vs. action cache
449 ms	57 ms	8x	21ms	21x	3x

Figure 12.7: For a PostgreSQL shared database on Heroku containing 1K movies and over 100 reviews per movie, the table shows the time in milliseconds to retrieve a list of the first 100 reviews sorted by creation date, with and without page and action caching. The numbers are from the log files visible with heroku logs.

the fragment cache hits, the line will never be executed and therefore the database will never be queried. Of course, you must still include logic in your cache sweeper to correctly expire the top-5-movies fragment when a new review is added.

In summary, both page- and fragment-level caching reward our ability to separate things that change (non-cacheable units) from those that stay the same (cacheable units). In page or action caching, split controller actions protected by before-filters into an “unprotected” action that can use page caching and a filtered action that can use action caching. (In an extreme case, you can even enlist a **content delivery network** (CDN) such as Amazon CloudFront to replicate the page at hundreds of servers around the world.) In fragment caching, use partials to isolate each cacheable entity, such as a single model instance, into its own partial that can be fragment-cached.

Earlier versions of Rails lacked lazy query evaluation, so controller actions had to explicitly check the fragment cache to avoid needless queries—very non-DRY.



Summary:

To maximize the benefits of caching, separate cacheable from non-cacheable units: controller actions can be split into cacheable and non-cacheable versions depending on whether a before-filter must be run, and partials can be used to break up views into cacheable fragments.

■ Elaboration: Where are cached objects stored?

In development, cached objects are generally stored in the local file system. Heroku offers add-ons such as Memcached that store cached content in the in-memory database memcached (pronounced *mem-cash-dee*; the suffix *-d* reflects the Unix convention for naming **daemon** processes that run constantly in the background). Rails cache stores must implement a common API so that different stores can be used in different environments—a great example of Dependency Injection, which we encountered in Section 11.6.

Self-Check 12.6.1. We mentioned that passing `:layout=>false` to `caches_action` provides most of the benefit of action caching even when the page layout contains dynamic elements such as the logged-in user’s name. Why doesn’t the `caches_page` method also allow this option?

- ◊ Since page caching is handled by the presentation tier, not the logic tier, a hit in the page cache means that Rails is bypassed entirely. The presentation tier has a copy of the whole page, but only the logic tier knows what part of the page came from the layout and what part came from rendering the action. ■

12.7 Avoiding Abusive Database Queries

As we saw in Section 12.2, the database will ultimately limit horizontal scaling—not because you run out of space to store tables, but more likely because a single computer can no longer

```
https://gist.github.com/c1527e55d6197620bc9bcc56168e4299
1 # assumes class Moviegoer with has_many :movies, :through => :reviews
2
3 # in controller method:
4 @fans = Moviegoer.where("zip = ?", code) # table scan if no index!
5
6 # in view:
7 - @fans.each do |fan|
8   - fan.movies.each do |movie|
9     // BAD: each time thru this loop causes a new database query!
10    %p= movie.title
11
12 # better: eager loading of the association in controller.
13 # Rails automatically traverses the through-association between
14 # Moviegoers and Movies through Reviews
15 @fans = Moviegoer.where("zip = ?", code).includes(:movies)
16 # GOOD: preloading movies reviewed by fans avoids N queries in view.
17
18 # BAD: preload association but don't use it in view:
19 - @fans.each do |fan|
20  %p= @fan.name
21  // BAD: we never used the :movies that were preloaded!
```

Figure 12.8: The query in the controller action (line 4) accesses the database once to retrieve rows of `@fans`, but each pass through the loop in lines 8–10 causes another separate database access, resulting in $n + 1$ accesses for a fan who has reviewed n movies. Line 15, in contrast, performs a single *eager load* query that also retrieves all the movies, which is nearly as fast as line 4 since most of the overhead of small queries is in performing the database access.

sustain the necessary number of queries per second while remaining responsive. When that happens, you will need to turn to techniques such as sharding and replication, which are beyond the scope of this book (but see To Learn More for some suggestions).

Even on a single computer, database performance tuning is enormously complicated. The widely-used open source database MySQL has dozens of configuration parameters, and most database administrators (DBAs) will tell you that at least half a dozen of these are “critical” to getting good performance. Therefore, we focus on how to keep your database usage within the limit that will allow it to be hosted by a PaaS provider: Heroku, Amazon Web Services, Microsoft Azure, and others all offer hosted relational databases managed by professional DBAs responsible for baseline tuning. Many useful SaaS apps can be built at this scale—for example, all of Pivotal Tracker¹² fits in a database on a single computer.

One way to relieve pressure on your database is to avoid needlessly expensive queries. Two common mistakes for less-experienced SaaS authors arise in the presence of associations:

1. The *n+1 queries* problem occurs when traversing an association performs more queries than necessary.
2. The *full table scan* problem occurs when your tables lack the proper **indices** to speed up certain queries. (This problem can occur even in the absence of associations, but is extremely common when associations are used.)

Lines 1–17 of Figure 12.8 illustrate the so-called $n+1$ queries problem when traversing associations, and also show why the problem is more likely to arise when code creeps into your views: there would be no way for the view to know the damage it was causing. Of course, just as bad is eager loading of information you won’t use, as in lines 18–21 of Figure 12.8. The bullet¹³ gem helps detect both problems.



<https://gist.github.com/02eafcc5a821c1dacc355f4df7300ac0>

```

1 class AddEmailIndexToMoviegoers < ActiveRecord::Migration
2   def change
3     add_index 'moviegoers', 'email', :unique => true
4     # :unique is optional - see text for important warning!
5     add_index 'moviegoers', 'zip'
6   end
7 end

```

Figure 12.9: Adding an index on a column speeds up queries that match on that column. The index is even faster if you specify `:unique`, which is a promise you make that no two rows will have the same value for the indexed attribute; to avoid errors in case of a duplicate value, use this in conjunction with a uniqueness validation as described in Section 5.1.

# of reviews:	2000	20,000	200,000	200,000	
Read 100, no indices	0.94	1.33	5.28	Create 1K, no indices	9.69
Read 100, FK indices	0.57	0.63	0.65	Create 1K, all indices	11.30
Performance	166%	212%	808%	Performance	-17%

Figure 12.10: For a PostgreSQL shared database on Heroku containing 1K movies, 1K moviegoers, and 2K to 200K reviews, this table shows the benefits and penalties of indexing. The first part compares the time in seconds to read 100 reviews with no indices vs. with foreign key (FK) indices on `movie_id` and `moviegoer_id` in the `reviews` table. The second part compares the time to create 1,000 reviews in the absence of indices and in the presence of indices over *every possible pair* of reviews columns, showing that even in this pathological case, the penalty for using indices is slight.

Another database abuse to avoid is queries that result in a ***full table scan***. Consider line 4 of Figure 12.8: in the worst case, the database would have to examine every row of the `moviegoers` table to find a match on the `zip` column, so the query will run more and more slowly as the table grows, taking time $O(n)$ for a table with n rows. The solution is to add a ***database index*** on the `moviegoers.zip` column, as Figure 12.9 shows. An index is a separate data structure maintained by the database that uses ***hashing*** techniques over the column values to allow constant-time access to any row when that column is used as the constraint. You can have more than one index on a given table and even have indices based on the values of multiple columns. Besides obvious attributes named explicitly in `where` queries, *foreign keys* (the subject of the association) should usually be indexed. For example, in Figure 12.8, the `moviegoer_id` field in the `reviews` table would need an index in order to speed up the query implied by `fan.movies`.

Of course, indices aren't free: each index takes up space proportional to the number of table rows, and since every index on a table must be updated when table rows are added or modified, updates to heavily-indexed tables may be slowed down. However, because of the read-mostly behavior of typical SaaS apps and their relatively simple queries compared to other database-backed systems such as Online Transaction Processing (OLTP), your app will likely run into many other bottlenecks before indices begin to limit its performance. Figure 12.10 shows an example of the dramatic performance improvement provided by indices.

Summary:

- The $n + 1$ queries problem, in which traversing a 1-to- n association results in $n + 1$ short queries rather than a single large query, can be avoided by judicious use of eager loading.
- Full-table scans in queries can be avoided by judicious use of database indices, but each index takes up space and slows down update performance. A good starting point is to create indices for all foreign key columns and all columns referenced in the `where` clause of frequent queries.

■ Elaboration: SQL EXPLAIN

Many SQL databases, including MySQL and PostgreSQL (but not SQLite), support an `EXPLAIN` command that describes the *query plan*: which tables will be accessed to perform a query and which of those tables have indices that will speed up the query. Unfortunately, the output format of `EXPLAIN` is database-specific. Starting with Rails 3.2, `EXPLAIN` is automatically run¹⁴ on queries that take longer than a developer-specified threshold in development mode, and the query plan is written to `development.log`. The `query_reviewer`¹⁵ gem, which currently works only with MySQL, runs `EXPLAIN` on all queries generated by ActiveRecord and inserts the results into a `div` at the top of every page view in development mode.

Self-Check 12.7.1. An index on a database table usually speeds up ____ at the expense of ____ and ____.

- ◊ Query performance at the expense of space and table-update performance ■

12.8 CHIPS: Exploiting Caching and Indices



CHIPS 12.8: The benefits of caching in SaaS

<https://github.com/saasbook/hw-indices-performance>

Enhance RottenPotatoes' performance by adding caching and database indices as appropriate, and measure the performance improvement from each enhancement.

12.9 Security: Defending Customer Data in Your App

As security is its own field in computing, there is no shortage of material to review or topics to study. Perhaps as a result, security experts have boiled down their advice into principles that developers can follow. Here are three:

- The **principle of least privilege** states that a user or software component should be given no more privilege—that is, no further access information and resources—than what is necessary to perform its assigned task. This is analogous to the “need-to-know” principle for classified information. One example of this principle in the Rails world is that the Unix processes corresponding to your Rails app, your database, and the Web

server (presentation tier) should run with low privilege and in an environment where they cannot even create new files in the file system. Good PaaS providers, including Heroku, offer a deployment environment configured in just this way.

- The *principle of fail-safe defaults* states that unless a user or software component is given explicit access to an object, it should be denied access to the object. That is, the default should be denial of access. Proper use of strong parameters as described in Section 5.2 follows this principle.
- The *principle of psychological acceptability* states that the protection mechanism should not make the app harder to use than if there were no protection. That is, the user interface needs to be easy to use so that the security mechanisms are routinely followed.

The rest of this section covers six specific security vulnerabilities that are particularly relevant for SaaS applications: protecting data using encryption, cross-site request forgery, SQL injection and cross-site scripting, clickjacking, prohibiting calls to private controller methods, and self-denial-of-service.

Protecting Data Using Encryption. Since competent PaaS providers make it their business to stay abreast of security-related issues in the infrastructure itself, developers who use PaaS can focus primarily on attacks that can be thwarted by good coding practices. Data-related attacks on SaaS attempt to compromise one or more of the three basic elements of security: privacy, authenticity, and data integrity. The goal of **Transport Layer Security** (TLS) and its predecessor Secure Sockets Layer (SSL) is to **encrypt** all HTTP traffic by transforming it using cryptographic techniques driven by a *secret* (such as a password) known only to the two communicating parties. Running HTTP over such a secure connection is called HTTPS.

Establishing a shared secret with a site you've never visited before is a challenging problem whose practical solution, **public key cryptography**, is credited to Ron Rivest, Adi Shamir and Len Adleman (hence **RSA**). A **principal** or communicating entity generates a **keypair** consisting of two matched parts, one of which is made public (accessible to everyone in the world) and the other of which is kept secret.

A keypair has two important properties:

1. A message encrypted using the private key can only be decrypted using the public key, and vice-versa.
2. The private key cannot be deduced from the public key, and vice-versa.

Property 1 provides the foundation of public-key encryption: if you receive a message that is decryptable with Bob's public key, only someone possessing Bob's private key could have created it. A variation is the digital signature: to attest to a message, Bob generates a one-way digest of the message (a short "fingerprint" that would change if the message were altered) and encrypts the digest using his private key as a way of attesting "I, Bob, vouch for the information in the message represented by this digest."

To offer secure access to his site `rottenpotatoes.com`, Bob generates a keypair consisting of a public part *KU* and a private part *KP*. He proves his identity using conventional means such as government-issued IDs to a **certificate authority** (CA) such as VeriSign. The CA then uses its own private key *CP* to sign a **public key certificate** that states, in effect, "`rottenpotatoes.com` has public key *KU*." Bob installs the certificate on his server and

Alice and Bob are the **archetypal principals** who appear in security scenarios, along with eavesdropper Eve, malicious Mallory, and other colorful characters.

`force_ssl` is implemented as a top-level before-action that causes an immediate redirect from `http://site/route` to `https://site/route`.

enables his SaaS stack to accept secure connections—usually trivial in a PaaS environment. Finally, he enables secure connections in his Rails app by adding `config.force_ssl=true` to his `config/environments/production.rb`, which turns on secure connections in production but not for development or testing.

The CA’s public key CU is built into most Web browsers, so when Alice’s browser first connects to `https://rottenpotatoes.com` and requests the certificate, it can verify the CA’s signature and obtain Bob’s public key KU from the certificate. Alice’s browser then chooses a random string as the secret, encrypts it using KU , and sends it to `rottenpotatoes.com`, which alone can decrypt it using KP . This shared secret is then used to encrypt HTTP traffic using much faster ***symmetric-key cryptography*** for the duration of the session. At this point, any content sent via HTTPS is reasonably secure from eavesdroppers, and Alice’s browser believes the server it’s talking to is the genuine RottenPotatoes server, since only a server possessing KP could have completed the key exchange step.

It’s important to recognize that this is the limit of what a secure HTTP connection can do. In particular, the server knows nothing about Alice’s identity, and no guarantees can be made about Alice’s data other than its privacy during transmission to RottenPotatoes.

Cross-site request forgery. A CSRF attack (sometimes pronounced “sea-surf”) involves tricking the user’s browser into visiting a different web site for which the user has a valid cookie, and performing an illicit action on that site as the user. For example, suppose Alice has recently logged into her MyBank.com account, so her browser now has a valid cookie for MyBank.com showing that she is logged in. Now Alice visits a chat forum where malicious Mallory has posted a message with the following embedded “image”:

`https://gist.github.com/4c84e87311f79ce0187b23f915f05e8e`

```
1 | <p>Here's a risqué picture of me:
2 |   
3 | </p>
```

When Alice views the chat message, or if she receives an email with the “image” link embedded in it, her browser will try to fetch the image from this RESTful URI, which happens to transfer \$5000 into Mallory’s account. Alice will see a “broken image” icon without realizing the damage. CSRF is often combined with Cross-site Scripting (see below) to perform more sophisticated attacks.

There are two steps to thwarting such attacks. The first is to ensure that RESTful actions performed using the GET HTTP method have no side effects. An action such as bank withdrawal or completing a purchase should be handled by a POST. This makes it harder for the attacker to deliver the “payload” using embedded asset tags like IMG, which browsers *always* handle using GET. The second step is to insert a randomly-generated string based on the current session into every page view and arrange to include its value as a hidden form field on every form. This string will look different for Alice than it will for Bob, since their sessions are distinct. When a form is submitted without the correct random string, the submission is rejected. Rails automates this defense: all you need to do is render `csrf_meta_tags` in every such view and add `protect_from_forgery` to any controller that might handle a form submission. Indeed, when you use `rails new` to generate a new app, these defenses are included in `app/views/layouts/application.html.erb` and `app/controllers/application_controller.rb` respectively.

SQL injection and cross-site scripting. Both of these attacks exploit SaaS apps that handle attacker-provided content unsafely. Defending against both can be summarized by the same advice: *sanitize any content coming from the user*. In ***SQL injection***, Mallory enters form data that she hopes will be interpolated directly into a SQL query statement executed

<https://gist.github.com/9181ed068de114441d4bc1a1c8c2aa74>

```

1 class MoviesController
2   def search
3     movies = Movie.where("name = '#{params[:title]}'"') # UNSAFE!
4     # movies = Movie.where("name = ?", params[:title])      # safe
5   end
6 end

```

Figure 12.11: Code that is vulnerable to a SQL injection attack. Uncommenting line 4 and deleting line 3 would thwart the attack using a *prepared statement*, which lets ActiveRecord “sanitize” malicious input before inserting it in the query.

params[:title]	SQL statement
Aladdin	SELECT "movies".* FROM "movies" WHERE (title='Aladdin')
'); DROP TABLE "movies"; --	SELECT "movies".* FROM "movies" WHERE (title='); DROP TABLE "movies"; --

Figure 12.12: If Mallory enters the text in the second row of the table as a movie title, line 3 of Figure 12.11 becomes a dangerous SQL statement that deletes the whole table. (The final --, the SQL comment character, avoids executing any SQL code that might have come after DROP TABLE.) SQL injection was often successful against early frameworks such as PHP, in which queries were hand-coded by programmers.

by the app. Figure 12.11 shows an example and its defense: using prepared statements, in which “dangerous” characters in parts of the SQL statement are properly escaped. In **cross-site scripting** (XSS), Mallory prepares a fragment of JavaScript code that performs a harmful action. Her goal is to get RottenPotatoes to render that fragment as part of a displayed HTML page, triggering execution of the script. Figure 12.13 shows how Mallory might try to do this, by creating a movie whose `title` attribute is a simple piece of JavaScript that will display an alert; real examples often include JavaScript code that steals Alice’s valid cookie and transmits it to Mallory, who can now “hijack” Alice’s session by passing Alice’s cookie as her own. Worse, even if the XSS attack only succeeds in reading the page content from another site and not the cookie, the page content might contain the CSRF-prevention token generated by `csrf_meta_tags` corresponding to Alice’s session, so XSS is often used to enable CSRF. Fortunately, the Rails Erb renderer always escapes “dangerous” HTML characters by default, as the figure shows; to prevent Erb from escaping a string `s`, you must render `raw(s)`, and if you do so, you’d better have a good reason for believing it is safe, such as having separately sanitized `s` when it was first received from Mallory.

<https://gist.github.com/729b807f39577bb830a15c0a4192e0cd>

```

1 <h2><%= movie.title %></h2>
2 <p>Released on <%= movie.release_date %>. Rated <%= movie.rating %>.</p>

```

<https://gist.github.com/eca0ede43a6bf8123440c5eadf8bc311>

```

1 <h2><script>alert("Danger!");</script></h2>
2 <p>Released on 1992-11-25 00:00:00 UTC. Rated G.</p>

```

<https://gist.github.com/7542d15d9c33b025050327a1af802b11>

```

1 <h2>&lt;script&gt;alert("Danger!");&lt;/script&gt;</h2>
2 <p> Released on 1992-11-25 00:00:00 UTC. Rated G.</p>

```

Figure 12.13: Top: a fragment of a view template that Mallory hopes to exploit. Middle: Mallory creates a new movie whose “title” is the string `<script>alert("Danger!");</script>`, hoping that RottenPotatoes will send an HTML page that causes the JavaScript code to be executed. Bottom: What RottenPotatoes actually sends; the Erb renderer automatically sanitizes any strings interpolated into HTML, thwarting Mallory’s attack.

Amazon is well protected against clickbait attacks; we use it only as an example.

Clickjacking or *UI redress attacks* are aimed at getting the user to take a UI action they normally wouldn't take, by obfuscating that action in the UI. Like XSS, they rely on deceiving the user regarding which site is actually displaying what they're seeing. For example, suppose you want to get many people to buy your widget on Amazon. First, create an unrelated page that has a "bait button" on it, such as "Click here for a free gift card." Craft that page so that it loads the Amazon product page for your widget into an HTTP `iframe`, and uses CSS to make the framed Amazon page transparent (invisible) but layered logically on top of the bait page, so that the "invisible" page is actually the one whose UI elements receive click events. Then position the framed page (more CSS) such that the Amazon "Buy Now With 1-Click" button is positioned directly over the bait button. The user thinks they're clicking the bait button, but in fact it's the Amazon button that receives the event and is activated. Of course, the user must be signed into Amazon for this to work, but there are many sites on which users have selected "remember me" so they don't have to login every time. Clickjacking was famously used in 2010 to garner many illegitimate Likes¹⁶ for a particular Facebook page.

The most effective defense against clickjacking is to ensure your site's pages cannot be framed on another site. All modern browsers observe the `X-Frame-Options` HTTP header; if the value is `SAMEORIGIN`, framing of a page is only allowed by other pages from the same site. Rails 4 and later set this header by default, but in earlier versions, the `secure_headers` gem was necessary to set it explicitly.

Prohibiting calls to private controller methods. It's not unusual for controllers to include "sensitive" helper methods that aren't intended to be called by end-user actions, but only from inside an action. Use `protected` for any controller method that isn't the target of a user-initiated action and check `rake routes` to make sure no routes include wildcards that could match a nonpublic controller action.

Self-denial-of-service. A malicious denial-of-service attack seeks to keep a server busy doing useless work, preventing access by legitimate users. You can inadvertently leave yourself open to these attacks if you allow arbitrary users to perform actions that result in a lot of work for the server, such as allowing the upload of a large file or generating an expensive report. For this reason, "expensive" actions are usually handled by a separate background process. For example, with Heroku, your app can queue the action using a simple queue system such as Redis, and a Heroku background worker¹⁷ can be triggered to pull jobs off the queue and run them while the main app server remains available to respond to interactive requests. Uploading files also carries other risks, so you should "outsource" that responsibility to other services; for example, many PaaS providers provide plugins for SaaS apps in popular languages facilitate the safe upload of files to external cloud-based storage such as Amazon Simple Storage Service (S3).

A final warning about security is in order. The "arms race" between SaaS developers and evildoers is ongoing, so even a carefully maintained site isn't 100% safe. In addition to defending against attacks on customer data, you should *also* be careful about handling sensitive data. Don't store passwords in cleartext; store them encrypted, or better yet, rely on third-party authentication as described in Section 5.2, to avoid embarrassing incidents¹⁸ of¹⁹ password²⁰ theft.²¹ Don't even *think* of storing credit card numbers, even encrypted. The Payment Card Industry association imposes an audit burden costing tens of thousands of dollars per year to any site that does this (to prevent credit²² card²³ fraud²⁴), and the burden is only slightly less severe if your code ever manipulates a credit card number even if you don't store it. Instead, offload this responsibility to sites like PayPal²⁵ or Stripe²⁶ that specialize in meeting these heavy burdens.

Attack	Rails Defenses
Eavesdropping	Install SSL certificate and configure SaaS app to redirect all insecure HTTP connections to HTTPS
Cross-site request forgery (CSRF)	Render <code>csrf_meta_tags</code> in all views (for example, by including it in main layout) and specify <code>protect_from_forgery</code> in <code>ApplicationController</code>
Cross-site scripting (XSS)	Sanitize HTML during rendering (many modern view-rendering systems do this automatically)
SQL injection	Use prepared queries with placeholders, rather than interpolating strings directly into queries
Executing protected actions	Use before-filters to guard sensitive public methods in controllers
Self-denial-of-service, pathologically slow clients	Use separate background workers to perform long-running tasks, rather than tying up the app server

Figure 12.14: Some common attacks against SaaS apps and the Rails mechanisms that defend against them.

Summary of defending customer data:

- Following the principles of *least privilege*, *fail-safe defaults*, and *psychological acceptability* can lead to more secure systems.
- Secure HTTP connections using TLS (formerly SSL) keep data private as it travels between the browser and server, and assure the browser of the server's identity, but provide no other guarantees. If the Certificate Authority that originally issued the certificate for the server's identity has been compromised, all bets are off, since it becomes possible to create counterfeit certificates that allow a rogue server to impersonate the legitimate server.
- Developers who deploy on a well-curated PaaS should focus primarily on attacks that can be thwarted by good coding practices. Figure 12.14 summarizes some common attacks on SaaS apps and the Rails mechanisms that thwart them.
- In addition to deploying app-level defenses, particularly sensitive customer data should either be stored in encrypted form or not at all, by outsourcing its handling to specialized services.

■ Elaboration: Keeping secrets: Encryption at rest

If your SaaS app's data is particularly sensitive, some PaaS providers offer **encryption at rest**, which encrypts the database file itself in a way that is transparent to any legitimate connection to the database. For truly **end-to-end encryption**, you can also encrypt specific data by using a strong symmetric encryption algorithm with a large key size, such as **AES-128**, and make the encryption key available only as an environment variable. (Each PaaS provider has a way to set environment variables whose values are available to a running app, so that the key value appears nowhere in the program text.) While end-to-end encryption for conventional databases prevents search queries from working, recent research such as the work of Dr. Raluca Ada Popa at UC Berkeley²⁷ has made great strides in computing on encrypted data, especially in encrypted-at-rest databases.

Self-Check 12.9.1. *True or false: If a site has a valid public key certificate, Cross-Site Request Forgery (CSRF) and SQL Injection attacks are harder to mount against it.*

- ◊ False. The security of the HTTP channel is irrelevant to both attacks. CSRF relies only on a site erroneously accepting a request that has a valid cookie but originated elsewhere. SQL injection relies only on the SaaS server code unsafely interpolating user-entered strings into a SQL query. ■

Self-Check 12.9.2. *Why can't CSRF attacks be thwarted by checking the Referer: header of an HTTP request?*

- ◊ The header can be trivially forged. ■

12.10 The Plan-And-Document Perspective on Operations

Non-functional requirements can be more important than adding new features, as violations can cause loss of millions of dollars, millions of users, or both. For example, sales for Amazon.com in the fourth quarter of 2012 was \$23.3B, so the loss of income due to Amazon being down just one hour would average \$10M. That same year a break-in of the Nebraska Student Information System²⁸ revealed social security numbers of anyone who applied to the University of Nebraska since 1985, estimated as 650,000 people. If customers can't trust a SaaS app, they will stop using it no matter what the set of features.

Performance. Performance is not a topic of focus in conventional software engineering, in part because it has been the excuse for bad practices and in part because it is well covered elsewhere. Performance can be part of the non-functional requirements and then later in acceptance-level testing to ensure the performance requirement is met.

Release Management. Plan-and-Document processes often produce software products that have major releases and minor releases. Using the Rails as an example, the last number of version 3.2.12 is a minor release, the middle number is a major release, and the first number is such a large change that it breaks APIs so that apps need to be ported again to this version. A release includes everything: code, configuration files, any data, and documentation. Release management includes picking dates for the release, information on how it will be distributed, and documenting everything so that you know what exactly is in the release and how to make it again so that it is easy to change when you have to make the next release. Release management is considered a case of configuration management in Plan-and-Document processes, which we review in Section 10.7.

Reliability. The main tool in our bag to make a system dependable is redundancy. By having more hardware than the absolute minimum needed to run the app and store the data, the system has the potential to continue even if a component fails. As all physical hardware has a non-zero failure rate, one redundancy guideline is to make sure there is *no single point of failure*, as it can be the Achilles' Heel of a system. Generally, the more redundancy the lower the chance of failure. As highly redundant systems can be expensive, it is important to have an adult conversation with the customer to see how dependable the app must be.

Dependability is holistic, involving the software and the operators as well as the hardware. No matter how dependable the hardware is, errors in the software and mistakes by the operators can lead to outages that reduce the **mean time between failures** (MTBF). As dependency is a function of the weakest link in the chain, it may be more effective to train operators how to run the app or to reduce the flaws in the software than to buy more redundant hardware to run the app. Since “to err is human,” systems should include safeguards to tolerate and prevent operator errors as well as hardware failures.

A foundational assumption of Plan-and-Document processes is that an organization can make the production of software predictable and repeatable by honing its process of software development, which should also lead to more reliable software. Hence, organizations commonly record everything they can from projects to learn what they can do to improve their process. For example, the ISO 9001 standard is granted if companies have processes in place, a method to see if the process is being followed, and record the results for each project so as to make improvements in their process. Surprisingly, standardization approval is not about the quality of the resulting code, it is just about the development process.

Finally, like performance, reliability can be measured. We can improve availability either taking longer between failures (MTBF) or by making the app reboot faster—the **mean time to repair** (**MTTR**)—as this equation shows:

$$\text{unavailability} \approx \frac{\text{MTTR}}{\text{MTBF}} \quad (12.1)$$

While it is hard to measure improvements in MTBF, as it can take a long time to record failures, we can easily measure MTTR. We just crash a computer and see how long it takes the app to reboot. And what we can measure, we can improve. Hence, it may be much more cost-effective to try to improve MTTR than to improve MTBF since it is easier to measure progress. However, they are not mutually exclusive, so developers can try to increase dependability by following both paths.

Security. While reliability can depend on probability to calculate availability—it is unlikely that several disks will fail simultaneously if the storage system is designed without hidden dependencies—this is not the case for security. Here there is a human adversary who is probing the corner cases of your design for weaknesses and then taking advantage of them to break into your system. The Common Vulnerabilities and Exposures database²⁹ lists common attacks to help developers understand the difficulty of security challenges.

Fortunately, defensive programming to make your system more robust against failures can also help make your system more secure. For example, in a **buffer overflow attack**, the adversary sends too much data to a buffer to overwrite nearby memory with their own code hidden inside the data. Checking the inputs to ensure that the user is not sending too much data can prevent such attacks. Similarly, the basis of **arithmetic overflow attack** might be to supply such an unexpectedly large number that when added to another number it will look small due to the wraparound nature of overflow with 32-bit arithmetic. Checking input

values or catching exceptions might prevent this attack. As computers today normally have multiple processors (“multicore”), an increasingly common attack is a **data race attack** where the program has non-deterministic behavior depending on the input. These concurrent programming flaws are much harder to detect and correct.

Testing security is much more challenging, but one approach is to use a **tiger team** as the adversaries who perform **penetration tests**. The team reports back to the developers the uncovered vulnerabilities.

Summary

Given the importance of keeping users’ trust, non-functional features can be more important than functional features, especially for SaaS apps.

- The Plan-and-Document processes speak little about performance, except as a potential piece of the Software Requirements Specification that is later validated as part of the Top-Level Test Plan.
- Releases, considered part of Configuration Management, are significant events in Plan-and-Document processes. A release wraps up everything about the project at that time, including documentation about how the release was made as well as the code, configuration files, data, and product documentation.
- Redundancy is the key to dependable systems, with highly available systems aiming to have no single point of failure. The **Mean Time Between Failures** (MTBF) is a function of the whole system, including hardware and operators along with the software. Another way to improve availability that is easier to measure than MTBF is to concentrate on reducing Mean Time to Repair (**MTTR**).
- Unlike the probabilistic basis for failures in dependability analysis, security is based on an intelligent adversary who is purposely exploiting unexpected events, such as buffer overflows.

Self-Check 12.10.1. Besides buffer overflows, arithmetic overflows, and data races, list another potential bug that can lead to security problems by violating one of the three security principles listed above.

◊ One example is improper initialization, which could violate the principle of fail-safe defaults. ■

12.11 Fallacies and Pitfalls



Fallacy: All the extra effort for testing very rare conditions in Continuous Integration tests is more trouble than it’s worth.

At 1 million hits per day, a “rare” one-in-a-million event is statistically likely every day. 1 million hits per day was Slashdot’s volume in 2010. At 8 billion (8×10^9) hits per day, which was Facebook’s volume in 2010³⁰, 8,000 “one-in-a-million” events can be expected per day. This is why code reviews at companies such as Google often focus on corner cases: at large scale, astronomically-unlikely events happen all the time (Brewer 2012). The extra resilience

Activity	Added latency	Measured effect
Amazon.com page view	100 ms	1% drop in sales
Yahoo.com page view	400 ms	5–9% drop in full-page traffic
Google.com search results	500 ms	20% fewer searches performed
Bing.com search results	2000 ms	4.3% lower revenue per user

Figure 12.15: The measured effects of added latency on users' interaction with various large SaaS apps, from Yahoo performance engineer Nicole Sullivan's "Design Fast Websites" presentation³³ and a joint presentation at the Velocity 2009 conference³⁴ by Jake Brutlag of Google and Eric Schurman of Amazon.

provided by error-handling code will help you sleep better at night.



Pitfall: Hidden assumptions that differ between development and production environments.

Section 2.6 explained how Bundler and the Gemfile automate the management of your app's dependencies on external libraries, and Section 4.2 explained how migrations automate making changes to your database. Heroku relies on these mechanisms for successful deployment of your app. If you manually install gems rather than listing them in your Gemfile, those gems will be missing or have the wrong version on Heroku. If you change your database manually rather than using migrations, Heroku won't be able to make the production database match your development database. Other dependencies of your app include the type of database (Heroku uses PostgreSQL), the versions of Ruby and Rails, the specific Web server used as the presentation tier, and more. While frameworks like Rails and deployment platforms like Heroku go to great lengths to shield your app from variation in these areas, using automation tools like migrations and Bundler, rather than making manual changes to your development environment, maximizes the likelihood that you've documented your dependencies so you can keep your development and production environments in sync. If it can be automated and recorded in a file, it should be!



Fallacy: We don't have to worry about performance because 3-tier SaaS apps can scale horizontally and cloud computing is cheap.

If you're using well-curated PaaS, *and* following the advice in this chapter for being kind to your database and leveraging caching, there is some truth to this statement up to a point. However, if your app "outgrows" PaaS, the fundamental problems of scalability and load balancing are now passed on to you. In other words, with PaaS you are *not* spared having to understand and avoid such problems, but you are *temporarily* spared from rolling your own solutions to them. When you start to set up your own system from scratch, it doesn't take long to appreciate the value of PaaS.

In Chapter 1 we argued for trading today's extra compute power for more productive tools and languages. However, it's easy to take this argument too far. In 2008, performance engineer Nicole Sullivan reported on experiments conducted by various large SaaS operators about how additional latency affected their sites. Figure 12.15 clearly shows that when extra processor time becomes extra latency (and therefore reduced responsiveness) for the end user, processor cycles aren't free at all.



Fallacy: The app is still in development, so we can ignore performance.

Knuth has said that premature optimization is the root of all evil "...about 97% of the

time.” But the quote continues: “Yet we should not pass up our opportunities in that critical 3%.” Blindly ignoring design issues such as lack of indices or needless repeated queries at design time is just as bad as focusing myopically on performance at design time. Being alert for, and avoiding, truly egregious performance mistakes will enable you to steer a happy path between two extremes.



Pitfall: Optimizing without measuring.

Some customers are surprised that Heroku doesn’t automatically add Web server capacity when a customer app is slow (van Hardenberg 2012). The reason is that without instrumenting and measuring your app, you don’t know *why* it’s slow, and the risk is that adding Web servers will make the problem worse. For example, if your app suffers from a database problem such as lack of indices or $n + 1$ queries, or if it relies on a separate service like Google Maps that is temporarily slow, adding servers to accept requests from *more* users will only make things worse. Without measuring, you won’t know what to fix.



Pitfall: Abusing continuous deployment, leading to cruft accumulation.

As we have already seen, evolving apps may grow to a point where a design change or architectural change would be the cleanest way to support new functionality. Since continuous deployment focuses on small incremental steps and tells us to avoid worrying about any functionality we don’t need immediately, the app has the potential to accumulate a lot of **cruft** as more code is bolted onto an obsolete design. The increasing presence of code smells (Chapter 9) is often an early symptom of this pitfall, which can be avoided by periodic design and architecture reviews when smells start to creep in.



Pitfall: Bugs in naming or expiration logic, leading to silently-wrong caching behavior.

As we noted, the two problems you must tackle with any kind of caching are naming and expiration. If you inadvertently reuse the same name for different objects—for example, a non-RESTful action that delivers different content depending on the logged-in user, but is always named using the same URI—then a cached object will be erroneously served when it shouldn’t be. If your sweepers don’t capture all the conditions under which a set of cached objects could become invalid, users could see stale data that doesn’t reflect the results of recent changes, such as a movie list that doesn’t contain the most recently added movies. Unit tests should cover such cases (“Caching system when new movie is added should immediately reflect new movie on the home page list”). Follow the steps in the Rails Caching Guide³⁵ to turn on caching in the testing and development environments, where it’s off by default to simplify debugging.



Pitfall: Slow external servers in an SOA that can adversely affect your own app’s performance.

If your app communicates with external servers in an SOA, you should be prepared for the possibility that those external servers are slow or unresponsive. The easy case is handling an unresponsive server, since a refused HTTP connection will result in a Ruby exception that you can catch. The hard case is a server that is functioning but very slow: by default, the call to the server will block (wait until the operation is complete or the TCP “slow timeout” expires, which can take up to three minutes), making *your* app slow down as well. Even if you are using a multi-threaded Rails app server such as unicorn, if each of N Web servers

<https://gist.github.com/6f8966c1f952ae9615df8170d1bb4663>

```

1 require 'timeout'
2 # call external service, but abort if no answer in 3 seconds:
3 Timeout::timeout(3.0) do
4   begin
5     # potentially slow operation here
6     rescue Timeout::Error
7       # what to do if timeout occurs
8   end
9 end

```

Figure 12.16: Using timeouts around calls to an external service protects your app from becoming slow if the external service is slow.

(“dynos” in Heroku’s terminology) is feeding requests to an app server with T threads, it takes only $N \times T$ simultaneous requests to hang your application completely. The solution is to use Ruby’s `timeout` library to “protect” the call, as the code in Figure 12.16 shows. Most modern app servers have some version of this mechanism built in, and allow it to be configured as part of the app server setup.



Fallacy: My app is secure because it runs on a secure platform and uses firewalls and HTTPS.

There’s no such thing as a “secure platform.” There are certainly *insecure* platforms, but no platform by itself can assure the security of your app. Security is a systemwide and ongoing concern: Every system has a weakest link, and as new exploits and software bugs are found, the weakest link may move from one part of the system to the other. The “arms race” between evildoers and legitimate developers makes it increasingly compelling to use professionally-curated PaaS infrastructure, so you can focus on securing your app code.



Fallacy: My app isn’t a target for attackers because it serves a niche audience, experiences low volume, and doesn’t store valuable information.

Malicious attackers aren’t necessarily after your app; they may be seeking to compromise it as a vehicle to a further end. For example, if your app accepts blog-style comments, it will become the target of blog spam, in which automated agents (bots) post spammy comments containing links the spammer hopes users will follow, either to buy something or cause malware to be installed. If your app is open to SQL injection attacks, one motive for such an attack might be to influence the code that is displayed by your views so as to incorporate a cross-site scripting attack, for example to cause malware to be downloaded onto an unsuspecting user’s machine. Even without malicious attackers, if any aspect of your app goes “viral” and becomes suddenly popular, you’ll be suddenly inundated with traffic. The lesson is: *If your app is publicly deployed, it is a target.*



Fallacy: Rails doesn’t scale (or Django, or PHP, or other frameworks).

With the shared-nothing 3-tier architecture depicted in Figure 12.2, the Web server and app server tiers (where Rails apps would run) can be scaled almost arbitrarily far by adding computers in each tier using cloud computing. The challenge lies in scaling the database, as the next Pitfall explains.



Pitfall: Putting all model data in an RDBMS on a single server computer, thereby limiting scalability.

The power of RDBMSs is a double-edged sword. It's easy to create database structures prone to scalability problems that might not emerge until a service grows to hundreds of thousands of users. Some developers feel that Rails compounds this problem because its Model abstractions are so productive that it is tempting to use them without thinking of the scalability consequences. Unfortunately, unlike with the presentation and logic tiers, we cannot "scale our way out" of this problem by simply deploying many copies of the database, because this might result in different values for different copies of the same item (the **data consistency** problem). Although techniques such as primary/replica and database **sharding** help make the database tier more like the shared-nothing presentation and logic tiers, extreme database scalability remains an area of both research and engineering effort.



Pitfall: Prematurely focusing on per-computer performance of your SaaS app.

Although the shared-nothing architecture makes horizontal scaling easy, we still need physical computers to do it. Adding a computer used to be expensive (buy the computer), time-consuming (configure and install the computer), and permanent (if demand subsides later, you'll be paying for an idle computer). With cloud computing, all three problems are alleviated, since we can add computers instantly for pennies per hour and release them when we don't need them anymore. Hence, until a SaaS app becomes large enough to require hundreds of computers, SaaS developers should focus on *horizontal scalability* rather than per-computer performance.

12.12 Concluding Remarks: Beyond PaaS Basics

The database abuses described in Section 12.7 reveal that object-relational mapping layers such as ActiveRecord, like most abstractions, are *leaky*: they try to hide implementation details for the sake of productivity, but concerns about security and performance sometimes require the developer to have some understanding of how the abstractions are implemented. For example, the $n + 1$ queries problem is not obvious from looking at ActiveRecord queries, nor are queries that would be speeded up by eager loading of associations.

In Chapter 4 we emphasized the importance of keeping your development and production environments as similar as possible. This is still good advice, but obviously if your production environment involves multiple servers and a huge database, it may be impractical to replicate in your development environment. So should you keep track of database performance in development if your production environment will be different? Absolutely. Heroku and other PaaS sites do a great job at tuning the baseline performance of their databases and software stack, but no amount of tuning can compensate for an app that forces the database to do inefficient queries or fails to use caching to ease the load on the database.

Given limited space, we focused on aspects of operations that every SaaS developer should know, even given the availability of PaaS. An excellent and more detailed book that focuses on challenges specific to SaaS and is laced with real customer stories is Michael Nygard's *Release It!* (Nygard 2007), which focuses more on the problems of "unexpected success" (sudden traffic surges, stability issues, and so on) than on repelling malicious attacks.

Understanding what happens during deployment and operations (especially automated deployment) is a prerequisite to debugging more complex performance problems. The vast majority of SaaS apps today, including those hosted on Windows servers, run in an environ-

ment based on the original Unix model of processes and input/output, so an understanding of this environment is crucial for debugging any nontrivial performance problems. *The Unix Programming Environment* (Kernighan and Pike 1984), coauthored by one of Unix’s creators, offers a high-bandwidth, learn-by-doing tour (using C!) of the Unix architecture and philosophy.

Sharding and **replication** are powerful techniques for scaling a database that require a great deal of design thinking up front. While most frameworks have libraries to help with both, these techniques usually also require database-level configuration changes, which many PaaS providers do not support. Sharding and replication have become particularly important with the emergence of “NoSQL” databases, which trade the expressiveness and data format independence of SQL for better scalability. *The NoSQL Ecosystem*, a chapter contributed by Adam Marcus to *The Architecture of Open Source Applications* (Marcus 2012), has a good treatment of these topics.

Security is an extremely broad topic; our goal has been to help you avoid basic mistakes by using built-in mechanisms to thwart common attacks against your app and your customers’ data. Of course, an attacker who can’t compromise your app’s internal data can still cause harm by attacking the infrastructure on which your app relies. Distributed **denial of service** (DDoS) floods a site with so much traffic that it becomes unresponsive for its intended users. A malicious client can leave your app server or Web server “hanging on the line” as it consumes output pathologically slowly, unless your Web server (presentation tier) has built-in timeouts. **DNS spoofing** tries to steer you to an impostor site by supplying an incorrect IP address when a browser looks up a host name, and is often combined with a **person-in-the-middle attack** (formerly “man-in-the-middle”) that falsifies the certificate attesting to the server’s identity. The impostor site then looks and behaves like the real site, and can “prove” its falsified identity to your browser, but can now collect sensitive information from users. Nonetheless, despite occasional vulnerabilities, curated PaaS sites are more likely to employ experienced professional system administrators who stay abreast of the latest techniques for avoiding such vulnerabilities, making them the best first line of defense for your SaaS apps.

Today’s best practices in SaaS security call for thinking in terms of DevSecOps, in which security is a consideration throughout the entire process of development rather than a “safety fence” around an existing app. Indeed, DevSecOps is the approach we advocate in this book; its key recommendations include encapsulation of microservices, automated tests for security-related features (input validation, login/authentication, and so on) in your test suite, and automated patching of security vulnerabilities arising from libraries or other app dependencies. This last service is provided by the free GitHub Dependabot³⁶, which scans your code for dependency vulnerabilities on each push and can even open pull requests automatically to update the vulnerable dependencies to a patched version.

Finally, at some point the unthinkable will happen: your production system will enter a state where some or all users receive no service. Whether the app has crashed or is “hung” (unable to make forward progress), from a business perspective the two conditions look the same, because the app is not generating revenue. In this scenario, the top priority is to restore service, which may require rebooting servers or doing other operations that destroy the post-mortem state you want to examine to determine what caused the problem in the first place. Generous logging can help, as the logs provide a semi-permanent record you can examine closely after service is restored.

L. Barroso and J. Dean. The tail at scale: Tolerating variability in large-scale online services. *Communications of the ACM*, 2012.

- N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *9th International World Wide Web Conference (WWW-9)*, pages 1–16, 2000.
- E. Brewer. Personal communication, May 2012.
- B. W. Kernighan and R. Pike. *Unix Programming Environment (Prentice-Hall Software Series)*. Prentice Hall Ptr, 1984. ISBN 013937681X.
- A. Marcus. The NoSQL ecosystem. In A. Brown, editor, *The Architecture of Open Source Applications*. lulu.com, 2012. ISBN 1257638017. URL <http://www.aosabook.org/en/nosql.html>.
- R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM. doi: 10.1145/1476589.1476628. URL <http://doi.acm.org/10.1145/1476589.1476628>.
- M. T. Nygard. *Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers)*. Pragmatic Bookshelf, 2007. ISBN 0978739213.
- P. van Hardenberg. Personal communication, April 2012.

Notes

- ¹https://projects.apache.org/project.html?httpd-http_server
- ²<http://www.iis.net>
- ³<http://heroku.com>
- ⁴https://workspace.google.com/terms/partner_sla.html
- ⁵<http://www.apdex.org>
- ⁶<https://developers.google.com/speed>
- ⁷<https://www.flexera.com/products/cloud-management-platform.html>
- ⁸<https://github.com/jamesgolick/rollout>
- ⁹<http://newrelic.com>
- ¹⁰<http://www.hpl.hp.com/research/linux/httpperf>
- ¹¹<https://www.thinkwithgoogle.com/future-of-marketing/digital-transformation/the-google-gospel-of-speed-urs-hoelzle>
- ¹²<http://pivotatracker.com>
- ¹³<https://github.com/flyerhzm/bullet>
- ¹⁴<http://weblog.rubyonrails.org/2011/12/6/what-s-new-in-edge-rails-explain>
- ¹⁵https://github.com/nesquena/query_reviewer
- ¹⁶<https://nakedsecurity.sophos.com/2010/05/31/viral-clickjacking-like-worm-hits-facebook-users/>
- ¹⁷<https://devcenter.heroku.com/articles/background-jobs-queueing>
- ¹⁸http://www.huffingtonpost.co.uk/2012/06/08/lastfm-hit-by-password-leak_n_1580012.html?ref=uk
- ¹⁹<http://www.zdnet.com/blog/btl/26000-email-addresses-and-passwords-leaked-check-this-list-to-see-if-youre-included/50424>
- ²⁰<http://www.neowin.net/news/main/09/10/05/thousands-of-hotmail-passwords-leaked-online>
- ²¹<http://hothardware.com/News/55000-Twitter-Accounts-Hacked-You-Should-Probably-Change-Your-Password/>
- ²²http://www.businessweek.com/technology/content/jul2009/tc2009076_891369.htm
- ²³<http://www.msnbc.msn.com/id/17853440/#.T9JsqxztEmY>
- ²⁴http://redtape.msnbc.msn.com/_news/2012/03/30/10940640-global-payments-under-15-million-account-numbers-hacked?lite
- ²⁵<http://paypal.com>

²⁶<http://stripe.com>

²⁷<https://people.eecs.berkeley.edu/~raluca/>

²⁸<https://newsroom.unl.edu/announce/todayatunl/1336/7831>

²⁹<http://cvedetails.com/>

³⁰<http://pingdom.com/blog/facebook-twitter-myspace-page-views>

³¹<http://www.slideshare.net/stubbornella/designing-fast-websites-presentation>

32<http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html>33<http://www.slideshare.net/stubbornella/designing-fast-websites-presentation>34<http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html>35http://guides.rubyonrails.org/caching_with_rails.html36<https://github.com/dependabot>

13

Afterword

Alan Kay (1940–) received the 2003 Turing Award for pioneering many of the ideas at the root of contemporary object-oriented programming languages. He led the team that developed the Smalltalk language, from which Ruby inherits its approach to object orientation. He also invented the “Dynabook” concept, the precursor of today’s laptops and tablet computers, which he conceived as an educational platform for teaching programming.



The best way to predict the future is to invent it.

—Alan Kay

13.1 Looking Backwards	412
13.2 Looking Forwards	413
13.3 Essential Readings	415
13.4 Last Words	416

Prerequisites and Concepts

Concepts:

- Agile is not the right fit for every project, nor is SaaS the right architecture for every project. But because of the “virtuous triangle” of SaaS, Agile, and Cloud Computing, this particular combination of ingredients has benefited all three areas, revolutionizing the future of software and making software development easier to learn.
- In this book you’ve used a successful distributed architecture (SaaS) and frameworks (Rails and jQuery). As an advanced software engineer, you’ll likely need to create such frameworks or extend existing ones. Paying careful attention to principles that made these frameworks successful, and the lessons from the past that informed their design, can help.
- Some of these lessons can be found in the wisdom captured in books about the software world. As George Santayana said, “Those who do not know history are condemned to repeat it.” We provide some suggestions of “classics in the field” that we believe all software engineers would benefit from reading.

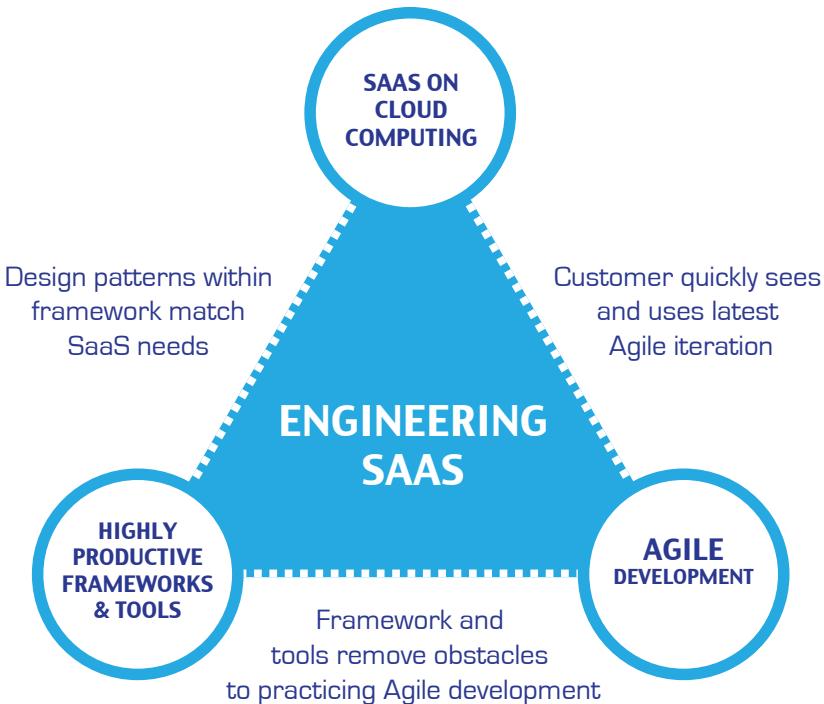


Figure 13.1: The Virtuous Triangle of Engineering SaaS is formed from the three software engineering crown jewels of (1) SaaS on Cloud Computing, (2) Highly Productive Framework and Tools, and (3) Agile Development.

13.1 Looking Backwards

Figure 13.1, first seen in Chapter 1, shows the three “crown jewels” on which the material in this book is based. Each pair of “jewels” forms synergistic bonds that support each other, as Figure 13.1 shows. In particular, the tools and related services of Rails makes it much easier to follow the Agile lifecycle. Figure 13.2 shows our oft-repeated Agile iteration, but this time it is decorated with the tools and services that we use in this book. These 14 tools and services support *both* following the Agile lifecycle *and* developing SaaS apps. Similarly, Figure 13.3 summarizes the relationship between phases of Plan-and-Document lifecycles and their Agile equivalents, showing how the techniques described in detail in this book play similar roles to those in earlier software process models.

Rails is very powerful but has evolved tremendously since version 1.0, which was originally *extracted* from a specific application. Indeed, the Web itself evolved from specific details to more general architectural patterns:

- From static documents in 1990 to dynamic content by 1995;
- From opaque URIs in the early 1990s to REST by the early 2000s;
- From session “hacks” (fat URIs, hidden fields, and so on) in the early 1990s to cookies and real sessions by the mid 1990s; and



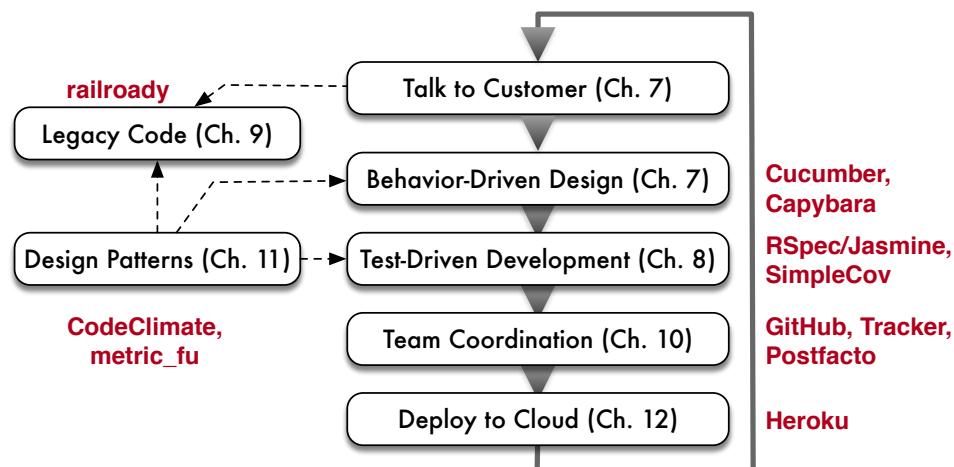


Figure 13.2: An iteration of the Agile software lifecycle and its relationship to the chapters in this book, with the supporting tools and services (red/bold letters) identified with each step.

Waterfall/Spiral	Agile	Chapter
Requirements gathering and analysis	BDD with short iterations so customer participates in design	7
Periodic code reviews	Pair programming (pairs constantly reviewing each others' code)	10
Periodic design reviews	Pull requests drive discussions about design changes	10
Test entire design after building	TDD to test continuously as you design	8
Post-implementation Integration Testing	Continuous integration testing	12
Infrequent major releases	Continuous deployment	12

Figure 13.3: While Agile methods aren't appropriate for all projects, the Agile lifecycle does embrace the same process steps as traditional models such as Waterfall and Spiral, but reduces each step in scope to a single iteration so that they can be done repeatedly and frequently, constantly refining a working version of the product.

- From setting up and administering your own *ad hoc* servers in 1990 to deployment on “curated” cloud platforms in the 2000s.

The programming languages Java and Ruby offer another demonstration that good incremental ideas can be embraced quickly but great radical ideas take time before they are accepted. Java and Ruby are the same age, both appearing in 1995. Within a few years Java became one of the most popular programming languages, while Ruby remained primarily of interest to the programming languages literati. Ruby's popularity came a decade later with the release of Rails. Ruby and Rails demonstrate that big ideas in programming languages really can deliver productivity through extensive software reuse. Comparing Java and its frameworks to Ruby and Rails, (Stella et al. 2008) and (Ji and Sedano 2011) found factors of 3 to 5 reductions in number of lines of code, which is one indication of productivity.



13.2 Looking Forwards

I've always been more interested in the future than in the past.

—Grace Murray Hopper

Given this history of rapidly-evolving tools, patterns, and development methodologies, what might software engineers look forward to in the next few years?

One software engineering technique that we expect to become popular in the next few years is *delta debugging* (Zeller 2002). It uses divide-and-conquer to automatically find the smallest input change that will cause a bug to appear. Debuggers usually use program analysis to detect flaws in the code itself. In contrast, delta debugging identifies changes to the program *state* that lead to the bug. It requires two runs, one with the flaw and one without, and it looks at the differences between the sets of states. By repeatedly changing the inputs and rerunning the program using a binary search strategy and automated testing, delta debugging methodically narrows the differences between the two runs. Delta debugging discovers dependencies that form a cause-effect chain, which it expands until it identifies the smallest set of changes to input variables that causes the bug to appear. Although it requires many runs of the program, this analysis is done at full program speed and without the intervention of the programmer, so it saves development time.

Program synthesis may be ready for a breakthrough. The state of the art today is that given incomplete segments of programs, program synthesis tools can often supply the missing code. One of the most interesting uses of this technology is in Microsoft Office Excel 2013, called the *Flash Fill feature*, which does programming by example (Gulwani et al. 2012). You give examples of what you want to do to rows or columns of code, and Excel will attempt to repeat and generalize what you do. Moreover, you can correct its attempts to steer it to what you want (Gantentbein 2012).

This split between Plan-and-Document and Agile development may become more pronounced with the advances in practicality of formal methods. The size of programs that can be formally verified is growing over time, with improvements in tools, faster computers, and wider understanding of how to write formal specifications. If the work of careful specification in advance of coding could be rewarded by not needing to test and yet have thoroughly verified programs, then the trade-offs would be crisp around change. For formal methods to work, clearly change needs to be rare. When change is commonplace, Agile is the answer, for change is the essence of Agile.

While Agile works better than other software methodologies for some types of apps today, it is surely not the final answer in software development. If a new methodology could simplify including a good software architecture and good design patterns while maintaining Agile's ease of change, it could become more popular. Historically, a new methodology comes along every decade or two, so it may soon be time for a new one.

This book itself was developed during the dawn of the **Massive Open Online Course** (MOOC) movement, which is another trend that we predict will become more significant in the next few years. Like many other advances in this modern world, we wouldn't have MOOCs without SaaS and cloud computing. The enabling components were:

- Scalable video distribution via services like YouTube.
- Sophisticated autograders running on cloud computing that evaluate assignments immediately yet can scale to tens of thousands of students.
- Discussion forums as a scalable solution to asking questions and getting answers from both other students and the staff.

These components combine to form a wonderful, low-cost vehicle for students around the world. For example, it will surely improve continuing education of professionals in our fast changing field, enable gifted pre-college students to go beyond what their schools can teach, and let dedicated students around the world who do not have access to great universities still get a good education. MOOCs may even have the side effect of raising the quality bar for traditional courses by providing viable alternatives to ineffective lecturers. If MOOCs deliver on only half of these opportunities, they will still be a potent force in higher education.

13.3 Essential Readings

Software tools change rapidly: languages and frameworks go in and out of vogue every few years. Software engineering methodologies change over time as well: Agile wasn't the first methodology and won't be the last, and variations of Agile continue to evolve. It may therefore seem perilous to recommend a list of readings that *all* aspiring software engineers should read, much less a list of online sources. Nonetheless, some of the field's bedrock ideas and acquired wisdom has stood the test of time, and we believe all software engineers would benefit by reading them. With some trepidation, we offer suggestions here, reminding the reader that we have *no formal or financial connection* to any of these works, although we do have professional or academic relationships with some of the authors.

Software design and architecture. We have mentioned Unix numerous times in this book; it is arguably the most influential production operating system ever created. While many of our readers were probably first exposed to it as Linux, that is only the latest and most widely adopted implementation of the original *kernel* or “core” of Unix, which chose and refined some of the best ideas from pioneering experimental systems such as *Multics* while greatly simplifying and streamlining some of its other aspects. Multics was an acronym for Multiplexed Information and Computing Service, with Multiplexed indicating that it was designed to serve multiple users simultaneously; the designers of Unix joked that their much smaller operating system might only be suitable for a single user at a time, so they named it Unics, later shortened to Unix.

The structure of Unix, and its approach to program design and to the management of processes and machine resources, are pervasive. We can suggest no better book than the one written by two of its designers: *The Unix Programming Environment* by Brian Kernighan and Rob Pike. Even many non-Unix operating systems borrow heavily from Unix’s models of process and resource management, and from a practical perspective, strong Unix toolsmithing skills are vital when you need to quickly produce some shell scripts to automate an otherwise tedious task.

Software project management. When Turing Award winner Frederick P. Brooks Jr. wrote *The Mythical Man-Month* (Brooks 1995), there was no such thing as “the software industry.” Software was generally written by programmers working for the companies that made the hardware, but the processes for estimating effort, coordinating the work of multiple team members, and performing quality control were far less evolved than they were for hardware design, which had a multiple-decade head start. Brooks’s account of managing the OS/360 project—the operating system for the groundbreaking IBM System/360, and far and away the most complex piece of commercial software ever written up to that time—still holds valuable lessons for software project management, even if the economics of the industry have changed.

If OS/360 was the face of software development in the 1960s, then collaboratively-



authored open-source development, exemplified by projects such as Linux, can be said to be at least part of the face of software development today. Developer Eric S. Raymond's *The Cathedral and the Bazaar*¹ (Raymond 2001), while controversial, is a good starting point for understanding how collaborative open-source development came about and how it compares to traditional in-house closed-source (proprietary) development. As of this writing, both models are vital to the software industry, with some companies embracing both. For example, Facebook and Twitter do not generally release the source code to their products, but they have released open-source tools such as React and Bootstrap originally developed for internal use.

The history of software. The evolution from proprietary early software to shrink-wrapped consumer software to SaaS is beautifully described in Martin Campbell-Kelly's *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry* (Campbell-Kelly 2003). For those interested in the corresponding history of hardware and the computing field generally, Paul Ceruzzi's *A History of Modern Computing* (Ceruzzi 2003) provides an outstanding overview; for the impatient, we recommend the much shorter and less detailed *Computing: A Concise History* by the same author (Ceruzzi 2012).

The modern business of software. Today, software is a business, and as Chapter 10 emphasized, most often a team effort. Building and running a successful software enterprise requires balancing technical expertise with great strategies for recruiting, hiring, team building, and retention. Joel Spolsky, creator of the project management tool Trello and co-creator of StackOverflow, for several years wrote a blog called *Joel On Software*², virtually every article of which is worth reading. You can read them online for free, or purchase the two books that collect and organize many of the posts by topic, *Joel On Software* and *More Joel On Software* (Spolsky 2004a,b). And if you're going to be managing software engineers (or anyone else for that matter), the actionable advice in *The One Minute Manager* (Blanchard and Johnson 1982) is hard to beat for clarity and conciseness.



Software as craftsmanship. Throughout the book we've affirmed the value of beautiful, well-tested code. Code that is hard to understand or poorly covered by tests is resistant to enhancement, and so is likely to be short-lived. Steve McConnell's *Code Complete* (McConnell 1993) justifiably remains a classic on practical software construction techniques. Robert C. "Uncle Bob" Martin's *Clean Code: A Handbook of Agile Software Craftsmanship* (Martin 2008) is an excellent and more recent companion to *Code Complete* that emphasizes taking a craftsman's pride in beauty and elegance in the code you write. Both should be on every developer's bookshelf. Reading and rereading these books periodically will help keep their suggestions at top of mind when you're actually at work.

Finally, to recommend a particular book is not to devalue any other particular book; no list of suggested readings can be definitive, complete, or fully objective. Nonetheless, we believe that these suggested "classics of the genre," which combine historical perspective with modern best practices, form a great starting point for software engineers wishing to really polish their skills while being fully aware of the work of those on whose shoulders they stand.

13.4 Last Words

Ultimately, it comes down to taste. It comes down to exposing yourself to the best things that humans have done, and then try to bring those things into what you're doing.

—Steve Jobs

Software helped put humans on the moon, led to the invention of lifesaving CAT scans, and enables eyewitness citizen journalism. By working as a software developer, you become part of a community that has the power to change the world.

But with great power comes great responsibility. Faulty software caused the loss of the Ariane V rocket³ and Mars Observer⁴ as well as the deaths of several patients due to radiation overdoses from the Therac-25 machine⁵.

While the early stories of computers and software are dominated by “frontier narratives” of lone geniuses working in garages or at startups, software today is too important to be left to any one individual, however talented. As we said in Chapter 10, software development is now a team sport.

We believe the concepts in this book increase the chances of you being both a responsible software developer *and* a part of a winning team. There’s no textbook for getting there; just keep writing, learning, and refactoring to apply your lessons as you go.

And as we said in the first chapter, we look forward to becoming passionate fans of the beautiful and long-lasting code that you and your team create!



K. H. Blanchard and S. Johnson. *The One Minute Manager*. William Morrow, Cambridge, MA, 1982.

F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading, MA, Anniversary edition, 1995. ISBN 0201835959.

M. Campbell-Kelly. *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. MIT Press, Cambridge, MA, 2003.

P. Ceruzzi. *A History of Modern Computing*. MIT Press, Cambridge, MA, 2003.

P. Ceruzzi. *Computing: A Concise History*. MIT Press, Cambridge, MA, 2012.

D. Gantenbein. Flash fill gives Excel a smart charge, Feb 2012. URL <http://research.microsoft.com/en-us/news/features/flashfill-020613.aspx>.

S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

F. Ji and T. Sedano. Comparing extreme programming and waterfall project results. *Conference on Software Engineering Education and Training*, pages 482–486, 2011.

R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN 9780132350884.

S. McConnell. *Code Complete (Microsoft Programming Series)*. Microsoft Press, 1993. ISBN 1556154844.

E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, Inc., 2001.

J. Spolsky. *Joel On Software*. Apress, 2004a.

J. Spolsky. *More Joel On Software*. Apress, 2004b.

L. Stella, S. Jarzabek, and B. Wadhwa. A comparative study of maintainability of web applications on J2EE, .NET and Ruby on Rails. *10th International Symposium on Web Site Evolution*, pages 93–99, October 2008.

A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, Nov 2002. ACM. doi: 10.1145/587051.587053. URL <http://www.st.cs.uni-saarland.de/papers/fse2002/p201-zeller.pdf>.

Notes

¹<http://www.catb.org/~esr/writings/cathedral-bazaar>

²<https://joelonsoftware.com>

³http://en.wikipedia.org/wiki/Ariane_5_Flight_501

⁴http://en.wikipedia.org/wiki/Mars_Observer

⁵<http://en.wikipedia.org/wiki/Therac-25>