

Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing

Beta Edition 0.9.0

Armando Fox and David Patterson

August 23, 2012

Copyright 2012 Strawberry Canyon LLC. All rights reserved.
No part of this book or its related materials may be reproduced in any form without
the written consent of the copyright holder.

Book version: 0.9.0

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 A.D. This photo is from the half-mile (0.8 km) long, 92 feet (28 m) high above ground segment built using unmortared, granite blocks. The Roman designers followed the architectural principles in the ten volumes series *De Architectura* ("On Architecture"), written in 15 B.C. by Marcus Vitruvius Pollio. It was untouched until the 1500s, when King Ferdinand and Queen Isabella performed the first reconstruction of these arches. The aqueduct was in use and delivering water until recently.

Both the print book and ebook were prepared with LaTeX, tex4ht, and Ruby scripts employing Nokogiri. Additional Ruby scripts automatically keep the Pastebin excerpts and screencast URLs up-to-date in the text. The necessary Makefiles, style files and most of the scripts are available under the BSD License at <http://github.com/armandofox/latex2ebook>.

Arthur Klepchukov designed the covers and graphics for all versions.

Contents

1 Engineering Software is Different from Hardware

1.1 Introduction

1.2 Product Lifetimes: Independent Products vs. Continuous Improvement

1.3 Development Processes: Waterfall vs. Spiral vs. Agile

1.4 Assurance: Testing and Formal Methods

1.5 Productivity: Conciseness, Synthesis, Reuse, and Tools

1.6 Software as a Service

1.7 Service Oriented Architecture

1.8 Cloud Computing

1.9 Fallacies and Pitfalls

1.10 Guided Tour of the Book

1.11 How NOT to Read this Book

1.12 Concluding Remarks: Engineering Software is More Than Programming

1.13 To Learn More

1.14 Suggested Projects

2 SaaS Architecture

2.1 100,000 Feet: Client-Server Architecture

2.2 50,000 Feet: Communication—HTTP and URIs

2.3 10,000 feet: Representation—HTML and CSS

2.4 5,000 Feet: 3-Tier Architecture & Horizontal Scaling

2.5 [1,000 Feet: Model-View-Controller Architecture](#)

2.6 [500 Feet: Active Record for Models](#)

2.7 [500 feet: Routes, Controllers, and REST](#)

2.8 [500 feet: Template Views](#)

2.9 [Fallacies and Pitfalls](#)

2.10 [Concluding Remarks: Patterns, Architecture, and Long-Lived APIs](#)

2.11 [To Learn More](#)

2.12 [Suggested Projects](#)

3 [Ruby for Java Programmers](#)

3.1 [Overview and Three Pillars of Ruby](#)

3.2 [Everything is an Object](#)

3.3 [Every Operation is a Method Call](#)

3.4 [Classes, Methods, and Inheritance](#)

3.5 [All Programming is Metaprogramming](#)

3.6 [Blocks: Iterators, Functional Idioms, and Closures](#)

3.7 [Mix-ins and Duck Typing](#)

3.8 [Make Your Own Iterators Using Yield](#)

3.9 [Fallacies and Pitfalls](#)

3.10 [Concluding Remarks: Idiomatic Language Use](#)

3.11 [To Learn More](#)

3.12 [Suggested Projects](#)

4 Rails From Zero to CRUD

4.1 [Rails Basics: From Zero to CRUD](#)

4.2 [Databases and Migrations](#)

4.3 [Models: Active Record Basics](#)

4.4 [Controllers and Views](#)

4.5 [Debugging: When Things Go Wrong](#)

4.6 [Form Submission: New and Create](#)

4.7 [Redirection and the Flash](#)

4.8 [Finishing CRUD: Edit/Update and Destroy](#)

4.9 [Fallacies and Pitfalls](#)

4.10 [Concluding Remarks: Designing for SOA](#)

4.11 [To Learn More](#)

4.12 [Suggested Projects](#)

5 Validating Requirements: BDD and User Stories

5.1 [Introduction to Behavior-Driven Design and User Stories](#)

5.2 [SMART User Stories](#)

5.3 [Introducing Cucumber and Capybara](#)

5.4 [Running Cucumber and Capybara](#)

5.5 [Lo-Fi User Interface Sketches and Storyboards](#)

5.6 [Enhancing Rotten Potatoes](#)

5.7 [Explicit vs. Implicit and Imperative vs. Declarative Scenarios](#)

5.8 [Fallacies and Pitfalls](#)

5.9 [Concluding Remarks: Pros and Cons of BDD](#)

5.10 [To Learn More](#)

5.11 [Suggested Projects](#)

6 [Verification: Test-Driven Development](#)

6.1 [Background: A RESTful API and a Ruby Gem](#)

6.2 [FIRST, TDD, and Getting Started With RSpec](#)

6.3 [The TDD Cycle: Red–Green–Refactor](#)

6.4 [More Controller Specs and Refactoring](#)

6.5 [Fixtures and Factories](#)

6.6 [TDD for the Model](#)

6.7 [Stubbing the Internet](#)

6.8 [Coverage Concepts and Unit vs. Integration Tests](#)

6.9 [Other Testing Approaches and Terminology](#)

6.10 [Fallacies and Pitfalls](#)

6.11 [Concluding Remarks: TDD vs. Conventional Debugging](#)

6.12 [To Learn More](#)

6.13 [Suggested Projects](#)

7 [Improving Productivity: DRY and Concise Rails](#)

7.1 [DRYing Out MVC: Partials, Validations and Filters](#)

7.2 [Single Sign-On and Third-Party Authentication](#)

7.3 [Associations and Foreign Keys](#)

7.4 [Through-Associations](#)

7.5 [RESTful Routes for Associations](#)

7.6 [Composing Queries With Reusable Scopes](#)

7.7 [Fallacies and Pitfalls](#)

7.8 [Concluding Remarks: Languages, Productivity, and Beauty](#)

7.9 [To Learn More](#)

7.10 [Suggested Projects](#)

8 [Legacy Software, Refactoring, and Agile Methods](#)

8.1 [What Makes Code “Legacy” and How Can Agile Help?](#)

8.2 [Exploring a Legacy Codebase](#)

8.3 [Establishing Ground Truth With Characterization Tests](#)

8.4 [Metrics, Code Smells, and SOFA](#)

8.5 [Method-Level Refactoring: Replacing Dependencies With Seams](#)

8.6 [Fallacies and Pitfalls](#)

8.7 [Concluding Remarks: Continuous Refactoring](#)

8.8 [To Learn More](#)

8.9 [Suggested Projects](#)

9 [Working In Teams vs. Individually](#)

9.1 [It Takes a Team: Two-Pizza and Scrum](#)

9.2 [Points, Velocity, and Pivotal Tracker](#)

9.3 [Pair Programming](#)

9.4 [Design Reviews and Code Reviews](#)

9.5 [Version Control for the Two-Pizza Team: Merge Conflicts](#)

9.6 [Using Branches Effectively](#)

9.7 [Reporting and Fixing Bugs: The Five R's](#)

9.8 [Fallacies and Pitfalls](#)

9.9 [Concluding Remarks: Teams, Collaboration, and Four Decades of Version Control](#)

9.10 [To Learn More](#)

9.11 [Suggested Projects](#)

10 [SOLID Design Patterns for SaaS](#)

10.1 [Patterns, Antipatterns, and SOLID Class Architecture](#)

10.2 [Just Enough UML](#)

10.3 [Single Responsibility Principle](#)

10.4 [Open/Closed Principle](#)

10.5 [Liskov Substitution Principle](#)

10.6 [Dependency Injection Principle](#)

10.7 [Demeter Principle](#)

10.8 [Fallacies and Pitfalls](#)

10.9 [Concluding Remarks: Is Agile Design an Oxymoron?](#)

10.10 [To Learn More](#)

10.11 [Suggested Projects](#)

[11 Enhancing SaaS With JavaScript](#)

[11.1 JavaScript: The Big Picture](#)

[11.2 Client-Side JavaScript for Ruby Programmers](#)

[11.3 Functions and Prototype Inheritance](#)

[11.4 The Document Object Model and jQuery](#)

[11.5 Events and Callbacks](#)

[11.6 AJAX: Asynchronous JavaScript And XML](#)

[11.7 Fallacies and Pitfalls](#)

[11.8 Concluding Remarks: JavaScript Past, Present and Future](#)

[11.9 To Learn More](#)

[11.10 Suggested Projects](#)

[12 Performance, Upgrades, Practical Security](#)

[12.1 From Development to Deployment](#)

[12.2 Quantifying Availability and Responsiveness](#)

[12.3 Continuous Integration and Continuous Deployment](#)

[12.4 Upgrades and Feature Flags](#)

[12.5 Monitoring and Finding Bottlenecks](#)

[12.6 Improving Rendering and Database Performance With Caching](#)

[12.7 Avoiding Abusive Queries](#)

[12.8 Defending Customer Data](#)

[12.9 Fallacies and Pitfalls](#)

[12.10 Concluding Remarks: Performance, Security, and Leaky Abstractions](#)

[12.11 To Learn More](#)

[12.12 Suggested Projects](#)

[13 Looking Backwards and Looking Forwards](#)

[13.1 Perspectives on SaaS and SOA](#)

[13.2 Looking Backwards](#)

[13.3 Looking Forwards](#)

[13.4 Evaluating the Book in the Classroom](#)

[13.5 Last Words](#)

[13.6 To Learn More](#)

[Appendix A Using the Bookware](#)

[A.1 Alpha Edition Guidance](#)

[A.2 Overview of the Bookware](#)

[A.3 Using the Bookware VM With VirtualBox](#)

[A.4 Using the Bookware VM on Amazon Elastic Compute Cloud](#)

[A.5 Working With Code: Editors and Unix Survival Skills](#)

[A.6 Getting Started With Git for Version Control](#)

[A.7 Getting Started With GitHub or ProjectLocker](#)

[A.8 Deploying to the Cloud Using Heroku](#)

[A.9 Fallacies and Pitfalls](#)

[A.10 To Learn More](#)

Foreword

Warning!

This book is an opinionated path through the bewildering array of methodologies, languages, tools, and artifact types that collectively make up “software engineering.” The goal is to instill good software habits in students—testability, software architecture, modularity, and reusability—while providing them the gratification of building a working deployed artifact that they themselves (and their peers) would use and find compelling.

This book is neither a step-by-step tutorial nor a reference book. Plenty of both are available online and in print; check saasbook.info for some suggestions. Instead, our goal is to bring a diverse set of topics together into a *single narrative*, help you understand the most important ideas by giving concrete examples, and teach you enough about each topic to get you started in the field. Throughout the book we recommend dozens of other excellent books that go into great depth about each topic; readers can decide for themselves which ones they will find most useful. The particular choice we make is to teach agile development using a methodology similar to extreme programming (XP) in the context of building and deploying a software-as-a-service (SaaS) application implemented using Ruby on Rails. Each choice has a good reason.

Why Agile?

We use the [**Software Engineering Body of Knowledge**](#) (SWEBOK), stewarded by the Institute of Electrical and Electronics Engineers, to introduce agile ideas and methods and apply them to the creation of SaaS, which we believe is important to the future of software. While agile might not be suitable for building safety-critical systems, its iteration-based, short-planning-cycle approach is a great fit for the reality of crowded undergraduate schedules and fast-paced courses. Busy students will by nature procrastinate and then pull several all-nighters to get a demo cobbled together and working by the project deadline; agile not only thwarts this tactic (since students are evaluated on progress being made each iteration) but in our experience actually leads to real progress using responsible discipline on a more regular basis.

Within each iteration, we are able to address the major issues of the software lifecycle in microcosm—requirements gathering with the customer, transforming requirements to user stories, driving the class-level architecture of the software using behavior-driven development, driving the implementation using test-driven development, and evaluating both unit/regression test coverage and acceptance/integration test results. That is, rather than first evaluating students on requirements gathering, then on good design, then on development and finally on test coverage and a working demo, *all* of these elements are evaluated on *every iteration*, encouraging the students to see the concepts and techniques as part of an

integrated ongoing process rather than as isolated stages in a pipeline.

Why Software as a Service?

To motivate students, it's helpful to use a platform that lets them create compelling apps. As of 2011, there were approximately 4.2 billion mobile phones deployed worldwide, or enough for 3 out of every 5 people on the planet; combined with the explosive growth of SaaS, we believe the future of software is "client + cloud" applications that are split between a tablet or smart phone and a cluster of servers to do heavy computation and persist data.

Therefore, both mobile applications for smart phones and tablets and Software as a Service (SaaS) for cloud computing are compelling targets for teaching students. As you can teach the principles with either target, given the time constraints of a single college course, we choose in favor of the platform with the most productive tools. Our experience is that it is no contest: the programming and testing frameworks for SaaS and cloud computing are dramatically more productive than those for mobile apps, and the client part of many SaaS apps can be adapted to mobile devices using the HTML/CSS/JavaScript skills learned in creating SaaS.

In addition, beyond the commercial promise of SaaS and the "hireability" of students who know how to create it, SaaS projects can be deployed inexpensively using public cloud computing, which means students' course artifacts are on display for the whole world to see. The exposure and "look what I made" factor of public deployment are hard to match.

Why Ruby and Rails? Why Not Java, C++, Python, or Scala?

We want students to understand that in the real world, programmers are rewarded not for the number of lines of code written or for how quickly they can "bash out" a feature, but for functionality delivered with high assurance of stability and while keeping the codebase beautiful and maintainable for continued growth. To many students, especially "hotshot" coders who come into a software engineering course with nontrivial programming experience, the methodologies and techniques we use to do this—design patterns, refactoring, test-first development, behavior-driven design—seem a strange and a dubious use of time.

We have found that students are more likely to gradually embrace these practices if given the best possible tools to support the practices. The Rails community has created by far the most seamless, elegant, and comprehensive tool set to support Agile and XP, and the idea of constantly refining and inventing tools that support testing as well as helping produce beautiful application code is a distinguishing characteristic of the Ruby developer ecosystem. While learning Ruby and Rails will be new to most students, juniors and seniors seem to learn it without difficulty, and far superior tools outweigh the learning costs.

A common counterargument in academia is "Our curriculum already teaches language X, so upper-division courses should leverage that knowledge." We believe

this approach optimizes for the wrong thing. First, software professionals are routinely expected to learn new languages by applying concepts from languages they already know, so “learning how to learn” new languages is a good skill to cultivate in class. Second, a language that makes it difficult to write and test beautiful and concise code is a poor vehicle for teaching those techniques, so the only investment being “leveraged” is syntactic knowledge, a hurdle surmounted with relative ease. Thus, even if our students never use Ruby again, they will have learned how to *reduce to practice* such important ideas as metaprogramming, higher-order programming, functional programming, and use of closures in the service of higher productivity and more maintainable code. We believe these skills will transfer to new languages, framework, and programming systems. Our survey of alumni of the course that led to this book (see Chapter [13](#)) suggests that our belief is well founded.

Why So Many Quotes?

We like quotes and we think they make the book more fun to read, but quotes are also efficient mechanisms to pass along wisdom to let novices learn from their elders, and 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.

Exercises, Self-Study, Suggested Projects, and MOOCs

One challenge with textbooks is what to do about exercises in the Internet age. Exercises traditionally serve multiple roles in textbooks:

- Used by teachers for homework assignments or even exams to get students to read and study material and to see if they understand it.
- Used by individual readers to see if they understand what they read. Note that teachers want the answers to be hidden so that students do their own work while individuals want the answers to be available so that they can check their understanding.
- Offer open-ended assignments that lead to interesting discussions or challenging programming tasks for people who want to go into the material in more depth.

Given the world wide web and search engines, answers are no longer hidden: faculty or teaching assistants routinely post answers to homework assignments on the web that search engines easily find. There are even entrepreneurs who sell answers to exercises of popular books either by pirating the instructor’s manual or by hiring students to write solutions.

From an author’s perspective, traditional exercises are now problematic:

- Writing good exercises is difficult, in part because you are asking questions rather than explaining, and in part because you want to ask questions whose answer is not simply a regurgitation of earlier text.
- Since students are being graded in part on whether they understand the exercise, the writing has to be precise and unambiguous, even to readers for whom English is a second language. Clarity demands increase the difficulty of writing good exercises, and exercises are more likely to have Errata than the rest of the book.
- To give faculty choices of what to ask and to increase chances that the answers are not available, you need to write a lot of exercises. Indeed, another textbook by one of your authors has 166 pages of exercises, which represent 20% of the book! Hence, exercises can add significantly to the cost and effort of a textbook.
- Moreover, despite these challenges, the “half-life” of exercises is so short that they are only useful for a few months after a new edition comes out.

As we state in the history section below, one reason we wrote this book is to have the chance to rethink what is different about an electronic textbook versus a print textbook. The section also states that one twist is that we are also offering a Massive Open Online Course (MOOC) open to everyone, which covers the same material. In fact, the video segments of the MOOC map nearly one-to-one with sections of the book.

Given this brave new world of electronic textbooks and electronic classrooms, we decided to try to a new approach to issues that exercises in print textbook try to solve:

- To help ensure that readers understand the material, each section ends with one or two short “Self-Check” questions, which come with answers.
- Each chapter ends with several open-ended assignments—“Suggested Projects”—for readers who want to go into more depth. Answers are not provided for projects.
- We have created an online question bank available to instructors for homework assignments and exams. Being online makes it much easier to make corrections to improve clarity. Instructors can make both withdrawals from and deposits to the question bank, and the questions have both answers and ratings of their usefulness. Since there is already a large set of questions and the number of questions will expand routinely versus being reset every few years on each new edition of a textbook, we believe it will be much harder for students to find online answers to their homework assignments.
- The MOOC has assignments and exams that correspond to book sections. To learn more, see www.saas-class.org. If you just want to learn the material and test if you understand on your own, you can sign up at any time and try the homeworks and the exams. Assignments and exams will be automatically graded, just like those of other MOOC students. (If you want to earn a

"Statement of Accomplishment," you need to sign up for the MOOC during restricted time periods, complete the course in a timely fashion, and get passing scores on the assignments and exams. However, you can watch videos and/or turn in assignments and/or take exams at any time if you don't care about the statement.)

Updating Electronic Books

One delightful feature of electronic textbooks from an author's perspective is that we can update all electronic copies of an edition when readers find mistakes in the book. Our plan is to collect the Errata together and release updates a few times a year.

Alas, ebook suppliers are reluctant to notify all ebook owners of updates, but there is no problem in ebook owners asking the supplier for updates. You can learn about new versions of an edition either by going to the web site beta.saasbook.info or by following the book on Twitter: follow @saasbook.

If you have a print book, you can follow the traditional model of checking the [Errata web page](#).

Instructor Support

Contact the authors if you are interested in the resources available to instructors who use this book.

Paths Through This Book

While the chapters are presented in what we think is a logical order, they are designed to be useful in many different combinations. In fact, to accommodate our goals for the Berkeley class, we have tried a few different orders that don't match the book. Figure 1 shows the schedule of topics and chapters/sections for the Spring 2012 and Fall 2012 versions of the class. (Note that in the latter portion of the class the students are primarily working on their projects, which is why we there are no reading assignments or lectures after the 10th week even though the Berkeley semester lasts 14 weeks.)

Lecture	Spring 2012	Fall 2012
1	Intro, Agile vs Waterfall: 1.1-1.6	Intro, Agile vs Waterfall: 1.1-1.6
2	Agile: 1.7-1.9, Architecture: 2.1-2.4	Agile: 1.7-1.9, Pair Prog.: 9.3, Intro I
3	Intro Ruby: 3.1-3.6	Intro Ruby: 3.3-3.8
4	Architecture: 2.5-2.12	Architecture: 2.1-2.6
5	Intro Ruby: 3.7-3.8, Rails: 4.1-4.4	Architecture: 2.7-2.10, Rails: 4.1-4.3
6	Rails: 4.5-4.10, BDD: 5.1-5.3	BDD: 5.1, 5.2, 5.5; Rails: 4.4-4.6
7	BDD: 5.4-5.9	Rails: 4.7-4.9; Cucumber: 5.3, 5.4, 5
8	TDD: 6.1-6.3	BDD: 5.7-5.9, TDD: 6.1-6.4
9	TDD: 6.4-6.7	TDD: 6.5-6.11
10	TDD: 6.8-6.11, Advanced SaaS: 7.1-7.3	Advanced SaaS: 7.1-7.5
11	Advanced SaaS: 7.4-7.5	Teams: 9.1-9.7 (less 9.3)
12	Teams: 9.1-9.7	Legacy: 8.1-8.6
13	Legacy: 8.1-8.6	Legacy & refactoring: 8.7-8.12
14	Legacy & refactoring: 8.7-8.12	Design patterns: Chapter 10
15	Design patterns: Chapter 10	Design patterns: Chapter 10
16	Design patterns: Chapter 10	JavaScript: Chapter 11
17	JavaScript: Chapter 11	Operations: Chapter 12
18	Operations: Chapter 12	Operations: Chapter 12
19	Operations: Chapter 12	End of Course: Chapter 13
20	End of Course: Chapter 13	

Figure 1: Lecture schedule with book sections for Berkeley course for Spring and Fall 2012. Each lecture is 80 minutes and there are two per week. We also moved exams in Fall to outside of class, so the it took one fewer lecture than Spring. The chapters and section numbers refer to the Beta edition, even though the Spring 2012 course used the Alpha edition, which numbered chapters differently.

The schedule changes for the Fall were based on student feedback. In particular, they suggested learning about Pair Programming earlier (section [9.3](#)), which is particularly helpful when learning Ruby. They also wanted to start engaging with non-technical customers earlier so that they could have more time on the project, which meant moving User Stories (section [5.1](#)), LoFi User Interfaces and Storyboarding (section [5.5](#)) a little earlier.

History of this Book

The material in this book started as a byproduct of a [Berkeley research project](#) that was developing technology to make it easy to build the next great Internet service. We decided that young people were more likely to come up with such a service, so we started teaching Berkeley undergraduates about Software as a Service using Agile techniques in 2007. Each year the course improved in scope, ambition, and popularity, embracing the rapid improvements in the Rails tools along the way. Between 2007 and 2012, our enrollments followed Moore's Law: 35, 50, 75, 115, and 170.

A colleague suggested that this would be excellent material for the software

engineering course that has long been taught at Berkeley, so one of us (Fox) taught that course with this new content. The results were so impressive that the other of us (Patterson) suggested that writing a textbook would let other universities benefit from this powerful curriculum.

These ideas crystallized with the emerging viability of electronic textbooks and the possibility of avoiding the costs and delays of a traditional publisher. In March 2011 we made a pact to write the book together. We were equally excited by making the material more widely available and about rethinking what an electronic textbook should be, since up until then they were essentially just the PDFs of print books.

The first step was to read related textbooks and professional books, and there is no shortage of books on each topic. Figures [2](#) and [3](#) show just 24 of the 50+ books we consulted, representing more than 10,000 pages! The sheer mass of these books can intimidate beginners. Therefore, a second reason to write the book is simply to offer a coherent introduction and overview of all these important topics within a single slim volume.

We also talked to others about the content. We attended conferences such as SIGCSE (Computer Science Education), the Conference on Software Engineering Education and Training, and the Federated Computing Research Conference both to talk with colleagues and to send them a survey to get their feedback. Also, since industry often complains about weaknesses in software education, we spoke to representatives from many leading software companies including Amazon, eBay, Facebook, Google, and Microsoft. We were struck by the unanimity of the number one request from each company: that students learn how to enhance sparsely-documented legacy code. In priority order, other requests were making testing a first-class citizen, working with non-technical customers, and working in teams. We believe the social skills needed to work effectively with nontechnical customers and work well in teams surely are helpful for the developers' whole careers; the question is how to fit them into one book. Similarly, no one questions the value of emphasizing testing; the question is how to get students to embrace it.

Given the perspective of educators and industrial colleagues, we proposed an outline that we thought addressed all these concerns, and started writing in June 2011. Given Fox's much greater expertise on the subject, he is writing roughly two-thirds of the chapters and Patterson the rest. Both of us collaborated on the organization and were first reviewers for each other's chapters. Fox also created the LaTeX pipeline that let us produce the many formats of the book for the various electronic and print targets. Most of this pipeline is now available at github.com/armandofox/latex2ebook, in case other authors may want to follow Fox's lead.

Our plan was to offer an Alpha edition of the textbook for 115 Berkeley students for the Spring semester 2012. Writing slowed considerably while teaching the course, but restarted in earnest based on student feedback in June 2012. The next step was to offer the Beta version of the textbook in the Fall semester 2012, ideally at other colleges and universities in addition to Berkeley. After that final round of feedback, the plan was to revise and publish the first edition for real in March 2013.

(Please check the [Errata web page](#) to suggest improvements to this edition and see corrections for errors found so far.)

These plans were altered in October 2011 when we were recruited to offer the first part of the Spring 2012 course online (for free) via [saas-class.org](#). As many as 25,000 students watched videos for this Massive Online Open Course (MOOC) that started in February, with 10,000 trying the first programming assignment, and 3500 hardy souls staying with the MOOC to the finish line. Given the popularity of the MOOC, we offered it again to thousands of students in May and July. Hence, this book received a much bigger Alpha test than we could ever have envisioned! These MOOCs covered the first half of the book. In addition to Beta testing in the classroom, we plan to cover the full curriculum as part of a two-part MOOC in the Fall, so the test will cover all chapters of the Beta edition.

We apologize in advance for the problems you find in this edition, and look forward to your feedback on how to improve this material.

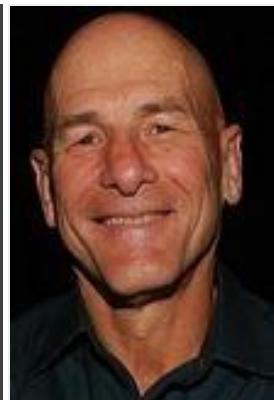
Armando Fox and David Patterson

August, 2012

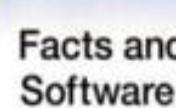
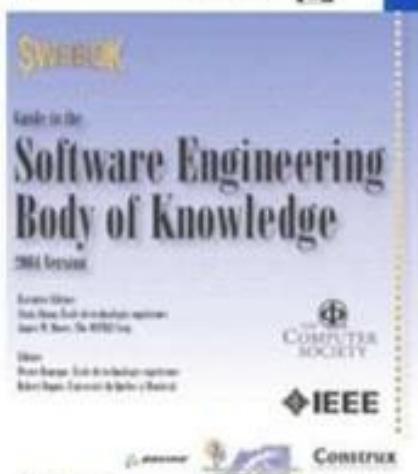
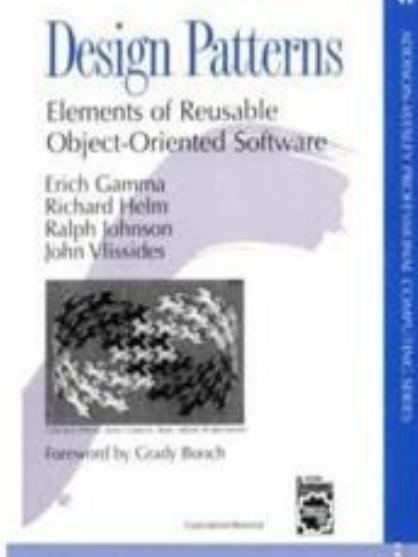
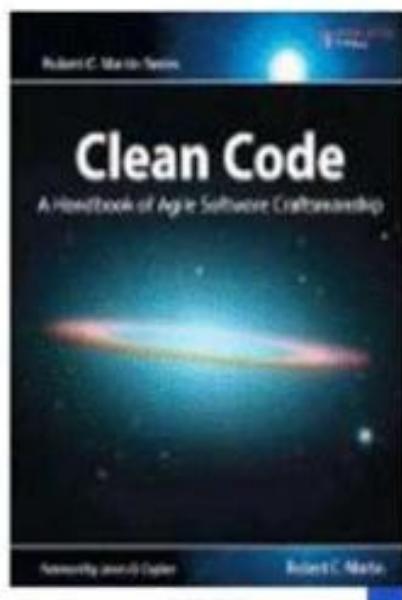
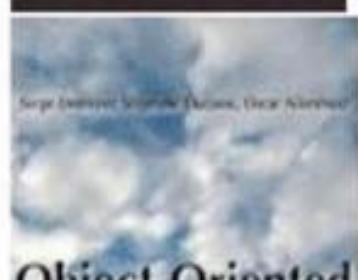
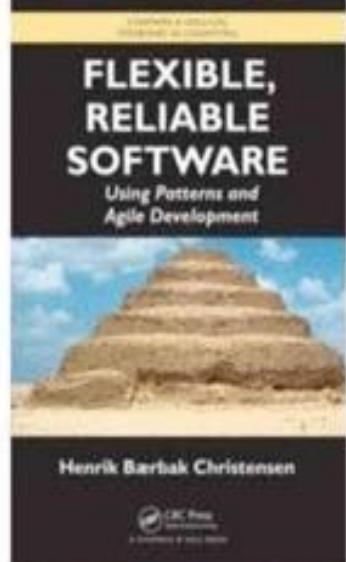
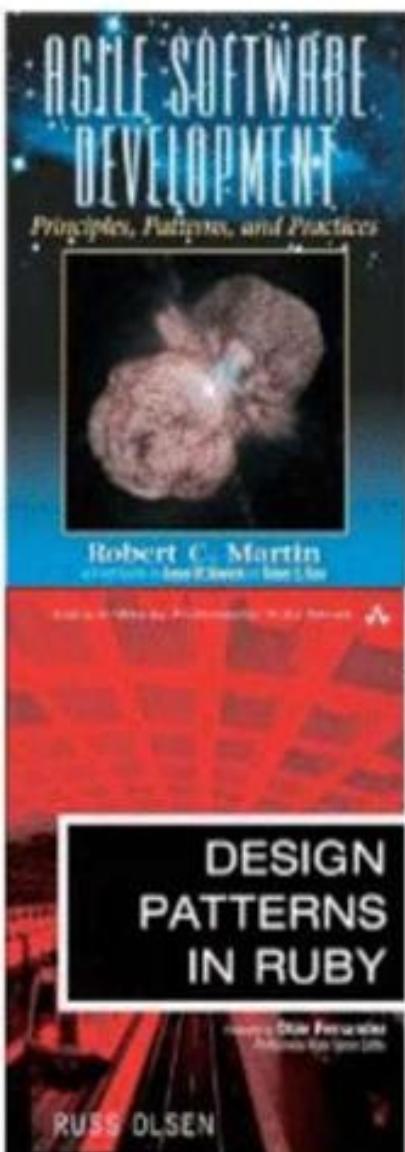
Berkeley, California



Armando Fox



David Patterson

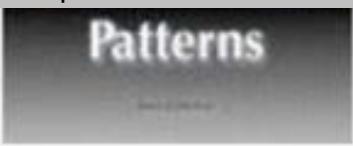


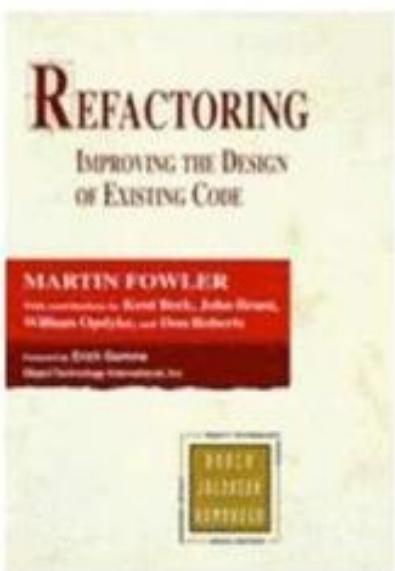
Robert
Forward





Figure 2: These 12 books contain more than 5000 pages. Your authors read more than 30 books to prepare this text. Most of these books are listed in the To Learn More sections at the end of the appropriate chapters.

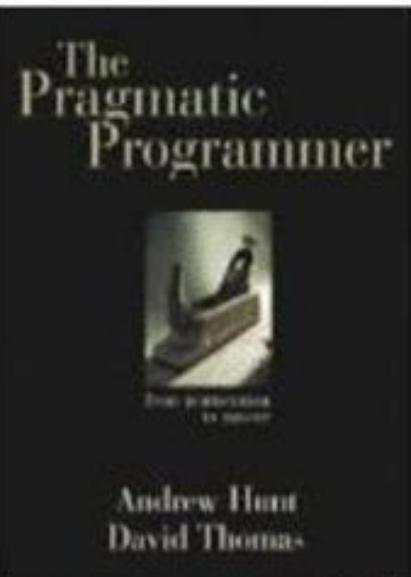




**SOFTWARE
MAINTENANCE
MANAGEMENT** A Study of
the Maintenance of
Computer Application Software
in 467 Data Processing
Organizations

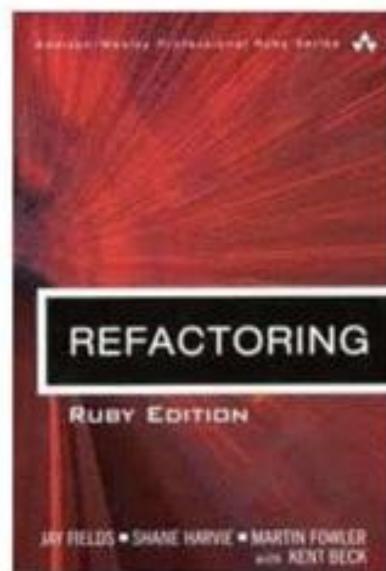
Bennet P. Liem
Institute School of Management, MSA
E. Burton Bassett
Institute School of Management, MSA

A
KODAK-AT&T PUBLISHING COMPANY
Peter Lang Publishers • 2000 • 400 pages
ISBN 1-56833-100-1 • \$29.95, \$19.95, \$12.95



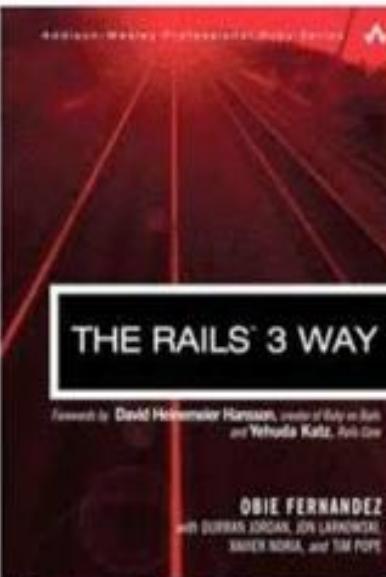
Andrew Hunt
David Thomas

Companion
Addison Wesley Professional Ruby Series



Software
Requirements
& Specifications

A Practical Guide for System and Analysis



Version Control with

Git



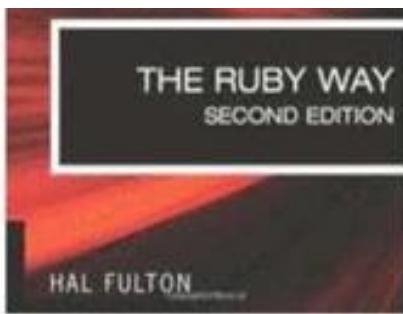


Figure 3: Another 12 books your authors read also contain more than 5000 pages. Most of these books are listed in the To Learn More sections at the end of the appropriate chapters.

Disclaimer

Throughout the book we use specific Web sites, tools, products, or trade names to ground examples in reality. Unless specifically noted, the authors have no formal connection to any of these sites, tools, or products, and the examples are for informational purposes only and not meant as commercial endorsements. Any trademarked names mentioned are the property of their respective owners and mentioned here for informational purposes only. Where possible, we focus on free and/or open-source software so that students can get hands-on ability related to the examples without incurring additional out-of-pocket costs.

The authors' opinions are their own and not necessarily those of their employer.

Acknowledgments

We thank our industrial colleagues who gave us feedback on our ideas about the course and the book outline, especially Tracy Bialik, Google; Brian Cunnie, Pivotal Labs; Brad Green, Google; Edward Hieatt, Pivotal Labs; Jason Huggins, SauceLabs; Matthew Kocher, Pivotal Labs; Raffi Krikorian, Twitter; Jim Larus, Microsoft Research; Jacob Maine, Pivotal Labs; Ken Mayer, Pivotal Labs; Rob Mee, Pivotal Labs; Tony Ng, eBay; Russ Rufer, Google; Arjun Satyapal, Google; Peter Van Hardenberg, Heroku; and Peter Vosshall, Amazon Web Services.

We thank our academic colleagues for their feedback on our approach and ideas, especially Marti Hearst, UC Berkeley; Paul Hilfinger, UC Berkeley; Daniel Jackson, MIT; Timothy Lethbridge, University of Ottawa; and Mary Shaw, Carnegie-Mellon University.

Part of the “bookware” is the collection of excellent third-party sites supporting SaaS development. For their help in connecting us with the right products and services that could be offered free of charge to students in the class, and valuable discussion on how to use them in an educational setting, we thank John Dunham, SauceLabs; Maggie Johnson, Google; Dana Le, Salesforce; James Lindenbaum, Heroku; Kami Lott and Chris Wanstrath, GitHub; Rob Mee, Pivotal Labs; Ann Merrihew, Kurt Messersmith, Marvin Theimer, Jinesh Varia, and Matt Wood, Amazon

Web Services; and Juan Vargas and Jennifer Perret, Microsoft.

We thank our graduate student instructors Kristal Curtis and Shoaib Kamil for helping us reinvent the on-campus class that led to this effort, and graduate student instructors Michael Driscoll and Richard Xia for helping us make scalable automatic grading a reality for the thousands of students that enrolled in the online course. Last but far from least, we thank our dedicated undergraduate lab staff over four iterations of the class: Alex Bain, Aaron Beitch, Allen Chen, James Eady, David Eliahu, Max Feldman, Amber Feng, Karl He, Arthur Klepchukov, Jonathan Ko, Brandon Liu, Robert Marks, Jimmy Nguyen, Sunil Pedapudi, Omer Spillinger, Hubert Wong, Tim Yung, and Richard Zhao.

We'd also like to thank Andrew Patterson, Grace Patterson, and Owyn Patterson for their help in marketing the book, as well as to their managers Heather Patterson, Michael Patterson, David Patterson, and Zackary Patterson.

Finally, we thank the hundreds of UC Berkeley students (115 took CS 169 in Spring 2012 and 170 are taking it in Fall 2012) and the thousands of MOOC students (3500 completed the February 2012 course and 1000 finished the May 2012 course) for being the guinea pigs!

About the Authors

Armando Fox is an Adjunct Associate Professor at UC Berkeley and a co-founder of the Berkeley AMP Lab. 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. He was named one of the "Scientific American 50" in 2003 and is the recipient of an NSF CAREER award and the Gilbreth Lectureship of the National Academy of Engineering. In previous lives he helped design the Intel Pentium Pro microprocessor and founded a successful startup to commercialize his UC Berkeley dissertation research on mobile computing. He received his other degrees in electrical engineering and computer science from MIT and the University of Illinois and is an ACM Distinguished Member. He is also a classically-trained musician and performer, an avid musical theater fan and freelance Music Director, and bilingual/bicultural (Cuban-American) New Yorker transplanted to San Francisco.

David Patterson is the Pardee Professor of Computer Science at UC Berkeley and is currently Director of the Parallel Computing Lab. In the past, he served as Chair of Berkeley's CS Division, Chair of the CRA, and President of the ACM. His best-known research projects are Reduced Instruction Set Computers (RISC), Redundant Arrays of Inexpensive Disks (RAID), and Network 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 as well as being named a Fellow of the Computer History Museum, ACM, IEEE, and both AAAS organizations. His teaching awards include the Distinguished Teaching Award (UC Berkeley), the Karlstrom Outstanding Educator Award (ACM), the Mulligan Education Medal (IEEE), and the

Undergraduate Teaching Award (IEEE). He received all his degrees from UCLA, which gave 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, annual charity bike rides and sprint triathlons, as well as the occasional weight-lifting contest.



Sir Maurice Wilkes (1913–2010) received the 1967 Turing Award for designing and building EDSAC in 1949, one of the first stored program computers. The [Turing Award](#) is the highest award in computing, which the Association for Computing Machinery (ACM) has bestowed annually since 1966. Named after computing pioneer Alan Turing, it is known informally as the “Nobel Prize of Computer Science.”

(This book uses sidebars to include what your authors think are interesting asides or short biographies of computing pioneers that supplement the primary text. We hope readers will enjoy them.)

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*

- 1.1 [Introduction](#)
- 1.2 [Product Lifetimes: Independent Products vs. Continuous Improvement](#)
- 1.3 [Development Processes: Waterfall vs. Spiral vs. Agile](#)
- 1.4 [Assurance: Testing and Formal Methods](#)
- 1.5 [Productivity: Conciseness, Synthesis, Reuse, and Tools](#)
- 1.6 [Software as a Service](#)
- 1.7 [Service Oriented Architecture](#)
- 1.8 [Cloud Computing](#)
- 1.9 [Fallacies and Pitfalls](#)
- 1.10 [Guided Tour of the Book](#)
- 1.11 [How NOT to Read this Book](#)

[1.12 Concluding Remarks: Engineering Software is More Than Programming](#)

[1.13 To Learn More](#)

[1.14 Suggested Projects](#)

To help understand the nature of engineering software, we contrast it with hardware engineering with regards to product lifetimes, development processes, productivity, and assurance. The similarities and differences led to popular processes for software development: Waterfall, Spiral, and Agile. We show the synergy between Software as a Service (SaaS), Cloud Computing, and Agile software development. We conclude with a tour of the remainder the book.

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

Gerald Weinberg, Weinberg's Second Law

Since the UNIVAC I in 1951, which was the first commercial computer, we have made tremendous advances in hardware design. Thanks in large part to Moore's Law, today's computers have 100 billion times the UNIVAC I's price-performance. ([Patterson and Hennessy 2008](#))

Ariane 5 flight 501. On June 4, 1996, an overflow occurred 37 seconds after liftoff in a guidance system, with [spectacular consequences](#), when a floating point number was converted to a shorter integer. This exception could not occur on the slower Ariane 4 rocket, so reusing successful components without thorough system testing was expensive: satellites worth \$370M were lost.

Over these six decades there have been so many software disasters that they are clichés, captured by short phrases that software engineers have memorized since they've heard the stories so many times: [Ariane 5 rocket explosion](#), [Therac-25](#) lethal radiation overdose, [Mars Climate Orbiter](#) disintegration, FBI [Virtual Case File](#) project abandonment, and so on. However, it is hard to recall a single comparable computer hardware disaster.

Why hasn't software engineering been as successful as hardware engineering? The reasons have to do with differences in nature of the two media and the subsequent cultures that have grown up around them. Let us start with the one major difference—product lifetimes—and the resulting development processes for hardware and software, before discussing two big ideas that are similar for software and hardware: assurance and productivity.

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.

The Oldest Living Program [might be MOCAS](#) ("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.

Hence, software can achieve a high-tech version of immortality, potentially getting

better over time while generations of computer hardware decay into obsolescence. This fundamental difference in the two media has led to different customer expectations, business models, and engineering development processes. Since a hardware product is not going to get any better over time, customers expect it to be basically bug-free when it arrives. If it's not working properly, the assumption is there had to have been a flaw in manufacturing of this particular item, and so it should be returned and replaced. If too many customers return products, the company loses money.

Customers are considerably more forgiving for software, assuming if there are problems when software is installed, they just need to get the latest version of it, presumably with their bugs fixed. Moreover, they 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!

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 of inelegance of its design or use of antiquated technology. To 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.



Surprisingly, despite the widely accepted importance of enhancing existing software, this topic is traditionally ignored in college courses and textbooks. We feature such software in this book for two 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, as that increases software's chances of a long life. 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.

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.

While software development processes initially followed hardware development processes, the potential to evolve over time has led to new development processes for software.



Grace Murray Hopper (1906–1992) was one of the first programmers, developed the first compiler, and was referred to as “Amazing Grace.” She became a rear admiral in the US Navy, and in 1997 a warship was named for her: the USS Hopper.

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

Grace Murray Hopper

The hardware community developed a rigid development process that starts, sensibly enough, with a thorough, detailed, written specification of what the product should do. What follows are a series of refinements that descend through levels of abstraction until there is description of the design at a low enough level that it can be manufactured. A great deal of time and effort is then expended to test that the low-level design matches the high level specification until it passes acceptance tests. Indeed, given the cost of mistakes, as much effort can be spent on verification as was spent on design.

The software version of this hardware development process is also a sequence of phases:

- Requirements analysis and specification
- Architectural design
- Implementation and Integration
- Verification
- 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 can also prevent unnecessary work downstream. This approach also expects extensive documentation with each phase, to make sure that important information is not lost if a person leaves the project and so that new people can get up to speed quickly when they join the project. Because it flows from the top down to completion, it 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.

Windows 95 was heralded by a [US\\$300 million outdoor party](#) 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.

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.

The Waterfall model can work well with well-defined tasks like NASA space flights, but it runs into trouble when customers are unclear on 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 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 1.1 illustrates this model of development across the four phases, which gives this lifecycle its name: the **Spiral model**. The phases are

Determine objectives and constraints of this iteration

Evaluate alternatives and identify and resolve risks

Develop and verify the prototype for this iteration

Plan the next iteration

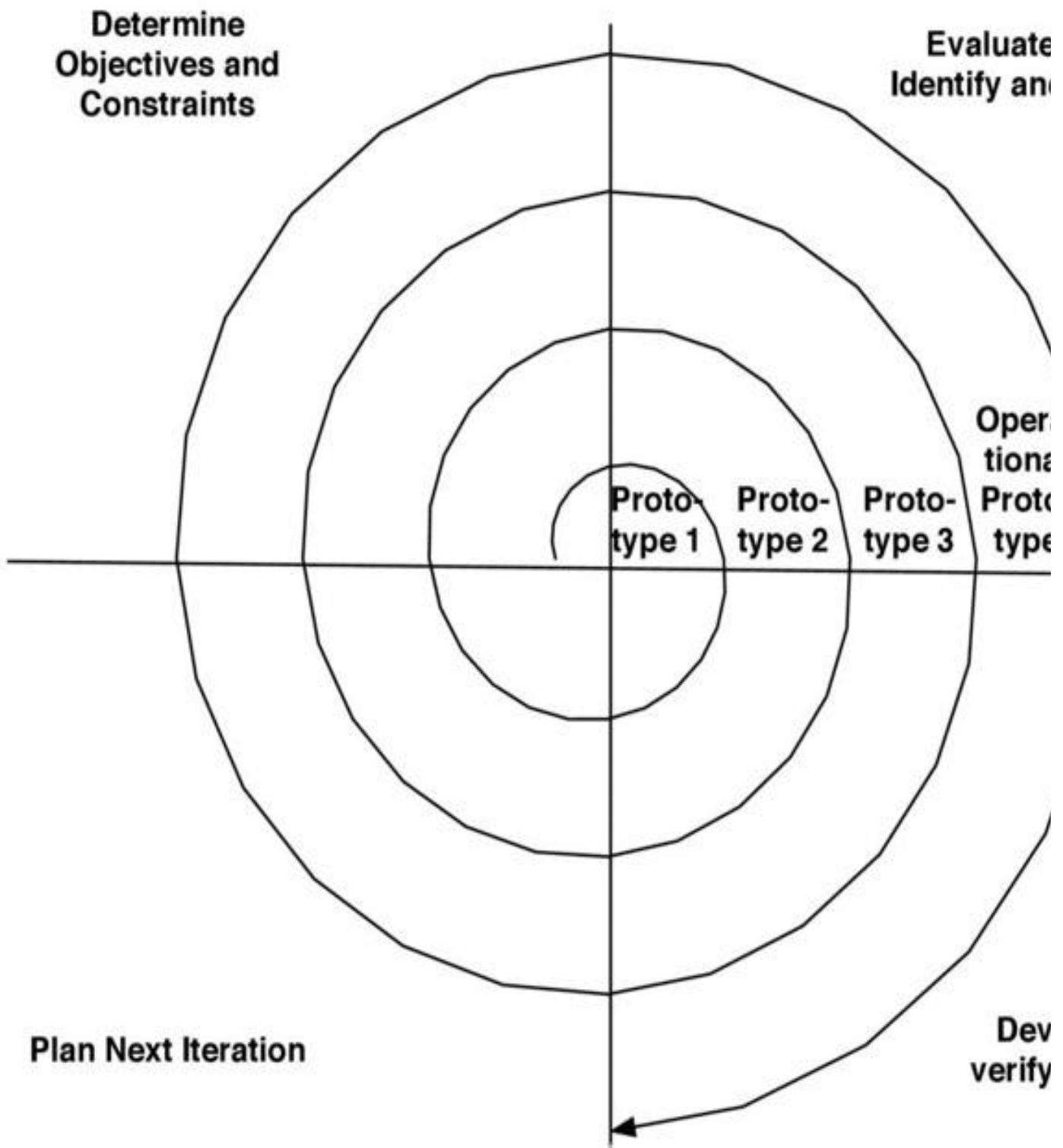


Figure 1.1: The Spiral lifecycle combines Waterfall with prototyping. It starts at the center, with each iteration around the spiral going through the four phases and resulting in a revised prototype until the product is ready for release.

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. Note that 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!

We will refer to the software processes like Waterfall and Spiral that involve a lot of planning and long, major phase changes with the term **Big Design Up Front**, abbreviated **BDUF**.

While a variety of software development lifecycles were developed over the years to better match the nature of software versus hardware beyond Waterfall and Spiral, perhaps the Reformation moment for software engineering was the Agile Manifesto in February 2001. A group of software developers met to develop a lighter-touch software lifecycle. Here is what the **Agile Alliance** nailed to the door of the “Church of Big Design Up Front”:

“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.”

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**.

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 result, with the customer offering feedback on each iteration. Agile emphasizes **Test-Driven Development (TDD)** to reduce mistakes, **user stories** to reach agreement and validate customer requirements, and **velocity** to measure 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 two weeks, so they are not even special events as in the Waterfall or Spiral models. The assumption is one of basically continuous improvement over its lifetime.

Summary: The Waterfall software development process or **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. In contrast, the Agile lifecycle works with customers to continuously add features to working prototypes until the customer is satisfied.

Self-Check 1.3.1. 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 to lead to creation of requirements as user stories, as we shall see in Chapter 5.

Self-Check 1.3.2. 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.

Having highlighted the difference in product lifetimes and the impact on development processes, we now show how hardware and software take similar approaches to making sure the artifacts meet their specifications.

And the users exclaimed with a laugh and a taunt:

"It's just what we asked for, but not what we want." *Anonymous*

We start this topic with definitions of two terms that are commonly interchanged but have subtle distinctions:

- **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?)

Generally, hardware prototypes aim at verification—ensuring that hardware meets the spec—while software prototypes more typically help with validation, since customers could not change hardware even if they wanted to, but they can demand and receive changes to software.

The two main approaches to verification and validation are **testing** and **formal analysis**. The motivation for both approaches is that the earlier developers find mistakes, the cheaper it is to repair them. Given the substantial costs of repair, both testing and formal analysis involve large communities of researchers and practitioners with many books on the topics. We highlight just some of the main issues here, starting with testing.

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!

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 and 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.

Another perspective, less common today, 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.

Given the multiple approaches and many levels of testing, from unit tests to acceptance tests, testing needs an indication of the fraction of the possible paths that have been tested. If we represent the paths as a graph, the term [**test coverage**](#) means what fraction of the paths is covered by the test. (Section [6.9](#) goes into more depth on test coverage.) Note that even 100% test coverage is no guarantee of design reliability, however, since it says nothing about the *quality* of the tests that were run.

Given the long development time of hardware and the long lifetime of software, another concern is whether later changes in design will cause failures in tests that it previously passed. To prevent such a backwards step, software engineers use [**regression testing**](#) to automatically rerun old tests to ensure that the current version still works at least as well as it used to. A related term is [**continuous integration**](#) or [**CI**](#) testing, which means the entire program is tested every time new code is checked in, versus having a separate test phase after completing development. Section [6.9](#) discusses testing your tests by tweaking source code ([**mutation testing**](#)) and testing your interfaces by throwing random data at your application and see what breaks ([**fuzz testing**](#)).

Given the quick tour of issues in testing, we can see how assurance is performed in our 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 Agile comes from test-driven development, in that the tests are written *before* the code when coding from scratch. When enhancing with existing code, TDD 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.

While testing is widely relied upon, quoting another Turing Award winner:



Edsger W. Dijkstra (1930–2002) received the 1972 Turing Award for fundamental contributions to developing programming languages.

Program testing can be used to show the presence of bugs, but never to show their absence! *Edsger W. Dijkstra*

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 using either [**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 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 and et al 2010](#))

NASA spent \$35M per year to maintain 420,000 lines of code for the space shuttle, or about \$80 per line of code per year.

Hence, formal methods are *not* good matches to high-function software that changes frequently, as is the case for Cloud Computing applications of this book. Consequently, we will not cover formal methods further.

Summary: Testing and formal methods reduce 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.
- Attacking the infeasibility of exhaustive testing, we divide 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.

- **Black-box** vs. **white-box** or **glass-box testing** refers to whether tests rely on the external specifications vs. the implementation of a module.
 - By mapping designs to graphs and recording which nodes and arcs are traversed, **test coverage** indicates what has and has not been tested.
 - **Regression testing** reapply old tests to reduce the chance of new revisions breaking designs that have worked in the past.
 - **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.
-

Self-Check 1.4.1. While all of the following help with verification, which form of tests 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.

Given this review of hardware and software development processes and approaches to assurance, we are ready to see how they make developers more productive.

Moore's Law means that hardware resources double every 18 months. Thus, computers would take increasingly long to design unless hardware engineers improved *their* productivity. In turn, these faster computers with much larger memories could run much larger programs. Similarly, to build bigger applications that could take advantage of the more powerful computers, software engineers needed to improve their productivity.

Both hardware and software engineers developed four fundamental mechanisms to improve their productivity:

Clarity via conciseness

Synthesis

Reuse

Automation and Tools

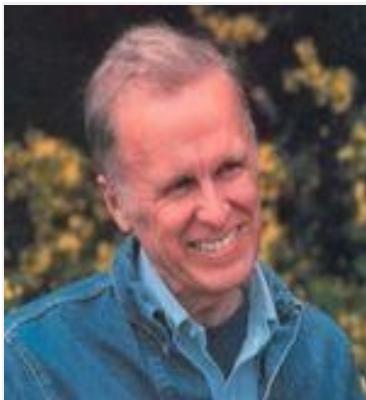
One of the driving assumptions of improving productivity of programmers is that if programs are easier to understand, then they will have fewer bugs as well as to be easier to evolve. A closely related corollary is that if the program is smaller, it's generally easier to understand. We capture this notion with our motto of "clarity via conciseness".

Programming languages do this 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, 7)  
a.should be >= 7
```

Unquestionably, the second version (which happens to be legal Ruby) is shorter and easier to read and understand, and will likely have fewer bugs and be easier to maintain. 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 (see Chapter 3).

The second way to improve clarity is to raise the level of abstraction. For hardware engineers, it meant going from designing transistors to gates to adders and so on up the hierarchy. With the design of whole systems on a single chip today, building blocks are whole processors or caches.



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 HLL.

Raising the level of abstraction for software engineers 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 that they formerly performed themselves to the compiler and runtime system. 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 [metaprogramming](#), which allows programs to modify their own structure and behavior at runtime. To highlight examples that improve productivity via conciseness, we will use this



“Concise” icon.

The second productivity mechanism is synthesis; that is, the implementation is generated 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, 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 third productivity mechanism is to reuse portions from past designs rather than write everything from scratch. 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. Hardware engineers also had the equivalent of procedures and design libraries.

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 ***object oriented programming***, 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 ([Alexander et al. 1977](#)) work in civil architecture, arose to address this need. Language support for reuse of design patterns includes ***dynamic typing***, which facilitates composition of abstractions, and ***mix-ins***, which offers ways to collect functionality from multiple methods without some of the pathologies of multiple inheritance found in some object oriented programming. 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 cutting 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.

A core value of computer engineers is finding ways to replace tedious manual tasks with automation to save time, improve accuracy, or both. Examples for lay people include word processors to simplify typing and editing, spreadsheets to make accounting easier, and email to make exchanging messages much lower overhead



than letters. (We highlight automation examples with the robot icon.) Not surprisingly, we also automate tedious tasks to help ourselves, giving us our fourth productivity enhancer. Electronic Computer Aided Design (ECAD) tools kept pace with the hardware designers' ascension in abstraction. New ECAD tools were also invented as new problems presented themselves with the march of Moore's Law, such as concerns about energy and power. Obvious CAD tools for software development are 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 that automate tedious tasks.

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.*

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. For example, Chapter 5 shows how [**Cucumber**](#) automates turning user stories into integration tests, Chapter 6 introduces [**RSpec**](#) that automates the unit testing process, and Chapter 9 demonstrates how [**Pivotal Tracker**](#) automatically measures [**Velocity**](#), which is a measure of the rate of adding features to an application. (We highlight tool



examples with the hammer icon.)

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. 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.

Returning to the development part of the lifecycle, productivity is measured in the people-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 a month—so much more work is done between releases that the customer sees, and hence the chances are greater for Waterfall and Spiral that more work will ultimately be rejected by the customer.

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. Recent advances include [reflection](#) that allows programs to observe themselves and [metaprogramming](#) that allows programs to modify their own structure and behavior at runtime.
 - 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) CAD tools to automate tedious tasks.
-

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?

④ 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, it is the weakest of the four for explaining the benefits of compilers.

Having reviewed how to improve software productivity, we now introduce a new way to ship software.

[***Software as a Service \(SaaS\)***](#) delivers software and data as a service over the Internet, usually via a thin program such as a browser that runs on local client devices instead of binary code that must be installed and runs wholly on that device. Examples that many of us use everyday include searching, social networking, and watching videos. The advantages for the customer and for the software developer are widely touted, including:

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.

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 data by losing the whole device, such as a phone or tablet.

When a group of users want to collectively interact with the same data, SaaS is a natural vehicle.

When data is large and/or updated frequently, it may make more sense to centralize data and offer remote access via SaaS.

Only a single copy of the server software runs in a uniform, tightly-controlled hardware and operating system environment selected by the developer, which avoids the compatibility hassles of distributing binaries that must run on wide-ranging computers and operating systems. In addition, developers

can test new versions of the application on a small fraction of the real customers temporarily without disturbing most customers. (If the SaaS client runs in a browser, there still are compatibility challenges, which we describe in Chapter 2.)

Since only developers have a copy of the software, they can upgrade the software and underlying hardware frequently as long as they don't violate the user-facing application program interfaces (API). Moreover, they don't need to annoy users with the seemingly endless requests for permission to upgrade their applications.

Multiplying the advantages to the customer and the developer together explains why SaaS is rapidly growing and why traditional software products are increasingly being transformed to offer SaaS versions. An example of the latter is Microsoft Office 365, which allows you to use the popular Word, Excel, and PowerPoint productivity programs as a remote service by paying for use instead of software that must be purchased in advance and installed on your local computer. Another example is TurboTax Online, which offers the same deal for another shrink-wrap standard-bearer.

SaaS: Innovate or Die? Lest you think the perceived need to improve a successful service is just software engineering paranoia, the most popular search engine used to be Alta Vista and the most popular social networking site used to be MySpace.

Note that the last item in the list above—frequent upgrades due to only one copy of the software—perfectly aligns itself with the Agile software lifecycle. 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. Hence, Amazon, eBay, Facebook, Google, and other SaaS providers all rely on the Agile lifecycle, and traditional software companies like Microsoft are increasing using Agile in their product development.

SaaS Programming Framework Programming Language

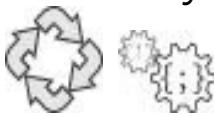
Active Server Pages (ASP.NET) Common Language Runtime (CLR)

Django	Python
Enterprise Java Beans (EJB)	Java
JavaServer Pages (JSP)	Java
Rails	Ruby
Sinatra	Ruby
Spring	Java
Zend	PHP

Figure 1.2: Examples of SaaS programming frameworks and the programming languages they are written in.

Unsurprisingly, given the popularity of SaaS, Figure 1.2 shows that there are many programming frameworks that claim to help. In this book we use Ruby on Rails (“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.

Ruby is typical of modern scripting languages in including automatic memory management and dynamic typing. 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. Useful additional features that help productivity via reuse include [mix-ins](#), which collects related behaviors and makes it easy to add them to many different classes, and [metaprogramming](#), which allows Ruby programs to synthesize code at runtime. Reuse is also enhanced with Ruby’s support for [closures](#) via [blocks](#) and [yield](#). Chapter 3 is a short description of Ruby for those who already know Java, and



Chapter 4 introduces Rails.

In addition to our view of Rails being technically superior for Agile and SaaS, Ruby and Rails are widely used. For example, Ruby routinely appears among top 10 most popular programming languages. Probably the best-known SaaS using Rails is Twitter, which began as a Rails app in 2006 and grew from 20,000 tweets per day in 2007 to 200,000,000 in 2011, during which time other frameworks replaced various parts of it.



If you are not already familiar with Ruby or Rails, this gives you a chance to practice an important software engineering skill mentioned above: 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 they routinely improve productivity by inventing new tools to automate tasks that were formerly done manually.

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 it, and so there are many frameworks to support them. This book uses Ruby on Rails.

Self-Check 1.6.1. Which of the following examples of Google SaaS apps is the *best* match to each of the six arguments given above: Search, Maps, News, Gmail,

Calendar, YouTube, and Documents.

While you can argue the mappings, below is our answer. (Note that we cheated and put some apps in multiple categories)

No user installation: Documents

Can't lose data: Gmail, Calendar.

Users cooperating: Documents.

Large/changing datasets: Search, Maps, News, and YouTube.

Software centralized in single environment: Search.

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 CLR and ASP.NET instead of Ruby and Rails.

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

SOA confusion At the time of this writing, Wikipedia has independent entries for [Service-oriented Architecture](#), [Service Oriented Architecture Fundamentals](#), and [Service-Oriented Architecture Types](#). The Wikipedia authors can't even agree on the capitalization and hyphenation!

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*

SaaS is actually a special case of a more general software architecture where all components are designed to be services: a [Service Oriented Architecture \(SOA\)](#). Alas, SOA is one of those terms that is so ill defined, over used, and over hyped that some think it is just an empty marketing phrase, like **modular**. SOA actually means that components of an application act as interoperable services, and can be used independently and recombined in other applications. The contrasting implementation is considered a "software silo," which rarely has APIs to internal components. If you mis-estimate what the customer really wants, the cost is much lower with SOA than with "siloed" software to recover from that mistake and try something else or to produce a similar-but-not-identical variant to please a subset of users.

For example, Amazon started in 1995 with siloed software for its online retailing site. [According to former Amazonian Steve Yegge](#), in 2002 the CEO and founder of Amazon mandated a change to what we would today call SOA. He broadcast an email to all employees along these lines:

All teams will henceforth expose their data and functionality through service interfaces.

Teams must communicate with each other through these interfaces.

There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols—doesn't matter. [Amazon CEO Jeff] Bezos doesn't care.

All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions. Anyone who doesn't do this will be fired.

Thank you; have a nice day!

A similar software revolution happened at Facebook in 2007—three years after the company went online—when [**Facebook Platform**](#) was launched. Relying on SOA, Facebook Platform allowed third party developers to create applications that interact with core features of Facebook such as what people like, who their friends are, who is tagged in their photos, and so on. For example, the New York Times was one of the early Facebook Platform developers. Facebook users reading the New York Times online on May 24, 2007 suddenly noticed that they could see which articles their friends were reading and which articles their friends liked. As a contrasting example of a social networking site using a software silo, Google+ had no APIs when it was launched on June 28, 2011 and had just one heavyweight API three months later: following the complete stream of everything a Google+ user sees.

To make these notions more concrete, suppose we wanted to create a bookstore service as first a silo and then as a SOA. Both will contain the same three subsystems: reviews, user profiles, and buying.

Figure 1.3 shows the silo version. The silo means subsystems internally can share access to data directly in different subsystems. For example, the reviews subsystem can get user profile info out of the users subsystem. However, all subsystems are inside a single external API (“the bookstore”).

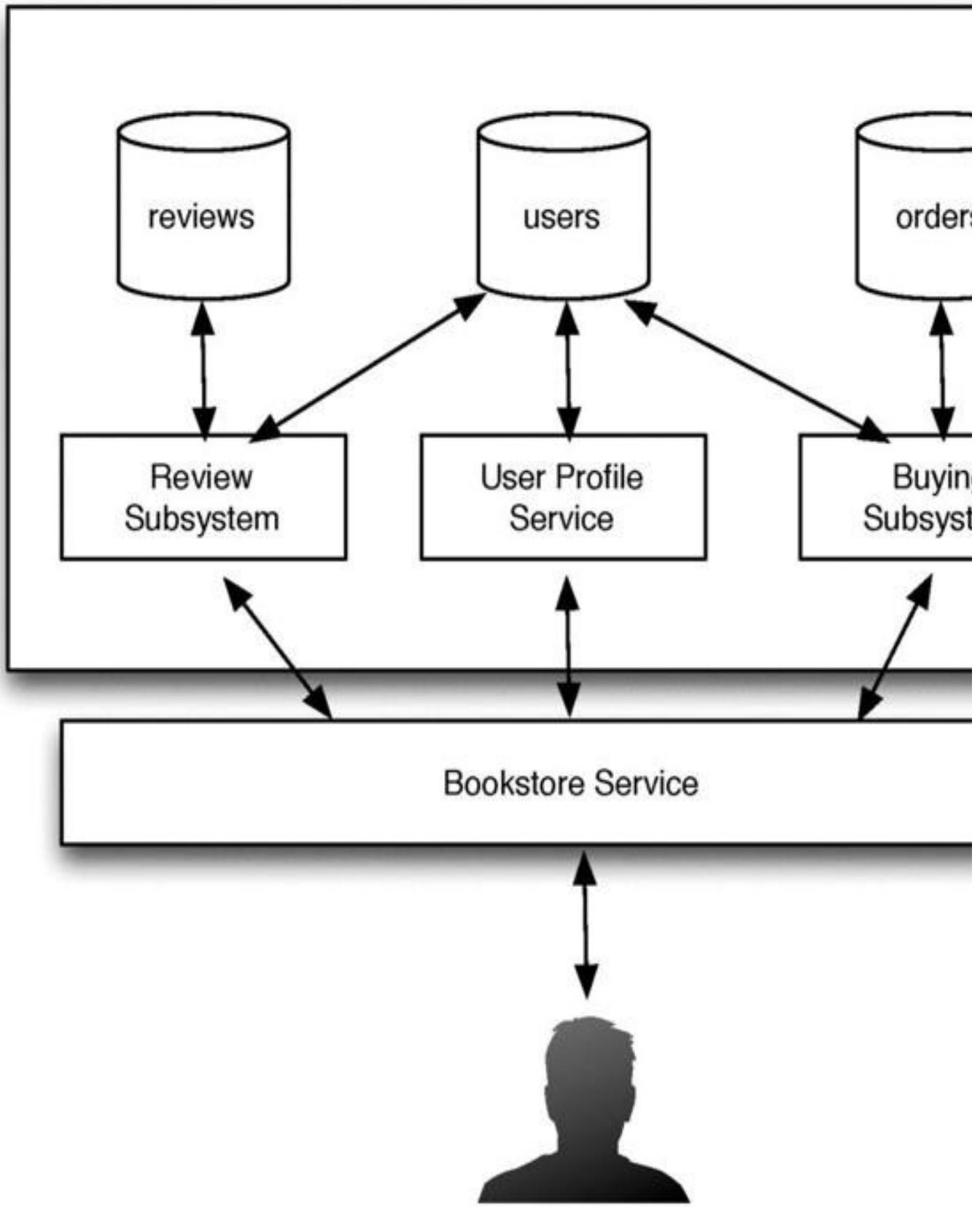


Figure 1.3: Silo version of a fictitious bookstore service, with all subsystems behind a single API.

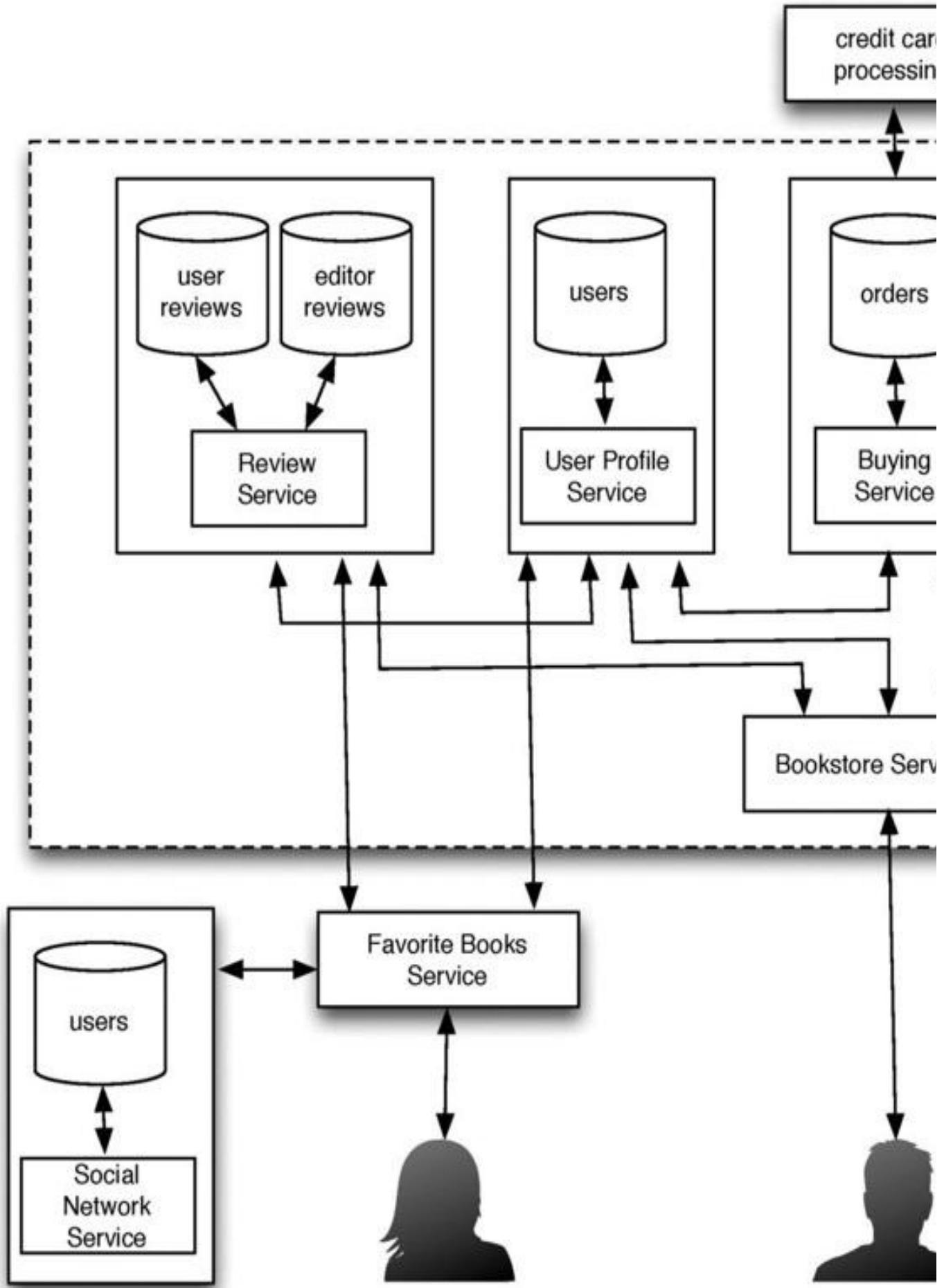


Figure 1.4: SOA version of a fictitious bookstore service, where all three subsystems are independent and available via APIs.

Figure 1.4 shows the SOA version of the bookstore service, where all subsystems are separate and independent. Even though all are inside the “boundary” of the bookstore’s datacenter, which is shown as a dotted rectangle, the subsystems interact with each other as if they were in separate datacenters. For example, if the reviews subsystem wants information about a user, it can’t just reach directly into the users database. Instead, it has to ask the users **service**, via whatever API is provided for that purpose. A similar restriction is true for buying.

The “bookstore app” is then just one particular composition of these services. Consequently, others can recombine the services with others to create new apps. For example, a “my favorite books” app might combine the users service and reviews service with a social network, so you can see what your social-network friends think about the books you have reviewed (see Figure 1.4).

The critical distinction of SOA is that no service can name or access another service’s data; it can only make requests for data through an external API. If the data it wants is not available through that API, then too bad. Note that SOA does not match the traditional layered model of software, in which each higher layer is built directly from the primitives of the immediately lower layer as in siloed software. SOA implies vertical slices through many layers, and these slices are connected together to form a service. While SOA usually means a bit more work compared to building a siloed service, the payback is tremendous reusability. Another upside of SOA is that the explicit APIs make testing easier.

There are two widely accepted downsides to SOA. Each invocation of a service involves the higher cost of wading through the deeper software stack of a network interface, so there is a performance hit to SOA. While a siloed system is very likely to be completely down on a failure, software engineers using SOA must deal with the sticky case of partial failures, so SOA makes dependability planning a bit more challenging.

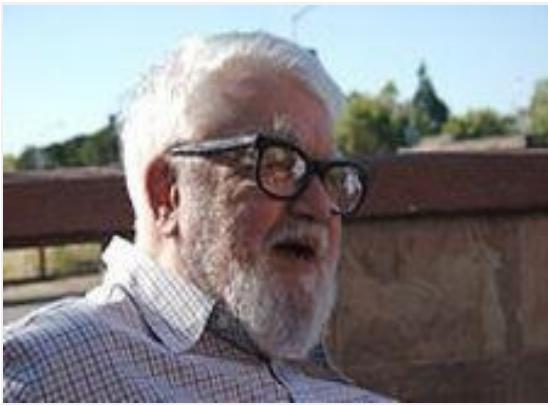
Summary: Although the term was nearly lost in a sea of confusion, **Service Oriented Architecture (SOA)** just means an approach to software development where all the subsystems are only available as external services, which means others can recombine them in different ways. Following the tools and guidelines in this book ensures that your SaaS apps will be a good fit to SOA.

Self-Check 1.7.1. 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 APIs visible is so that programmers can stand on the shoulders of others.

Given the case for SaaS and the understanding that it relies on a Service Oriented Architecture, we are ready to see the underlying hardware that makes SaaS possible.



John McCarthy (1927–2011) received the Turing

Award in 1971 and is the inventor of Lisp. As a pioneer of timesharing large computers, as early as 1961 he envisioned an “ecosystem” foreshadowing today’s Software as a Service in which large computers provide continuous service to large numbers of users with a utility-pricing-like model. Clusters of commodity hardware and the spread of fast networking have helped make this vision a reality.

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:

Communication, to allow any customer to interact with the service.

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.

Dependability, in that both the service and the communication vehicle must be continuously available: every day, 24 hours a day (“24×7”).

The gold standard set by the US public phone system is 99.999% availability (“five nines”), or about 5 minutes of downtime per year.

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.

- 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. The goal is to imitate with low overhead, and one popular use is to simplify software distribution within a cluster.
- ([Barroso and Hoelzle 2009](#)) show that the cost of the equivalent amount of processors, memory, and storage is much less for clusters than “big iron,” perhaps by a factor of 20.
- 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. 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.

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***, as 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*** or **[utility computing](#)**, which offers computing, storage, and communication at pennies per hour (see ([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 App Engine, and Microsoft Azure. The public cloud means today that 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.

Rapid growth of FarmVille The prior record for number of users of a social networking game was 5 million. FarmVille had 1 million players within 4 days after it was announced, 10 million after 2 months, and 28 million daily players and 75 million monthly players after 9 months. 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.

Today, we call this long held dream of computing as a utility **[Cloud Computing](#)**. We believe that Cloud Computing and SaaS are transforming the computer industry, with the full impact of this revolution taking the rest of this decade to determine. Indeed, this revolution is one reason we decided to write this book, as we believe engineering SaaS for Cloud Computing is radically different from engineering shrink-wrap software for PCs and servers.

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 1.8.1. True or False: Internal datacenters could get the same cost savings as Warehouse Scale Computers 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 internal datacenters.

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

We include this section near the end of chapters to explain the ideas once again from another perspective and to give 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: 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 project 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.*

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

Agile is a nice match to some types of software, particularly SaaS, which is why we use it in this book. However, Agile is *not* best for all software. Agile is inappropriate 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.

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

Pitfall: Ignoring the cost of software design.

Since there is no cost to manufacture 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 costs 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.

I hear and I forget. I see and I remember. I do and I understand. *Confucius*

With this introduction behind us, we can now explain what follows and what paths you might want to take. To do and understand, as Confucius advises, begin by reading Appendix [A](#). It explains how to obtain and use the “bookware”, which is our name for the software associated with the book.

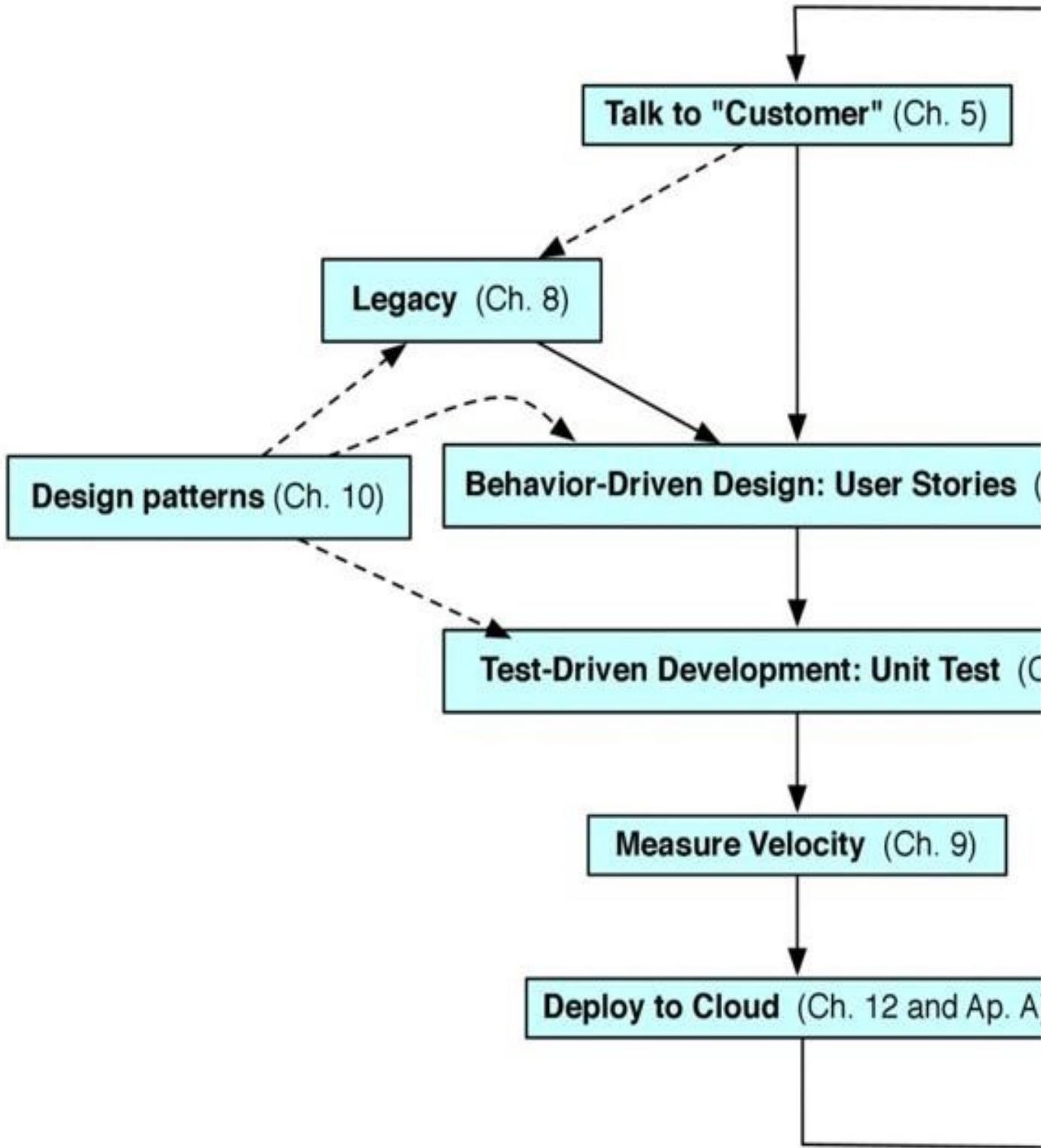


Figure 1.5: An iteration of the Agile software lifecycle and its relationship to the chapters in this book. The dashed arrows indicate a more tangential relationship between the steps of an iteration, while the solid arrows indicate the typical flow. Note that the Agile process applies equally well to existing legacy applications and new applications, although the customer may play a smaller role with legacy apps.

Chapter 2 explains the architecture of a SaaS application, using an altitude analogy of going from the 100,000-foot view to the 500-foot view. During the descent you'll

learn the definition of many acronyms that you may have already heard—APIs, CSS, IP, REST, TCP, URLs, URIs, and XML—as well as some widely used buzzwords: cookies, markup languages, port numbers, and three-tier architectures. It also demonstrates the importance of design patterns, particularly Model-View-Controller that is at the heart of Rails.

Rather than just tell you how to build long lasting software and watch you forget, we believe you must do to understand. It is much easier to try good guidelines if the tools encourage it, and we believe today the best SaaS tools are found in the Rails framework, which is written in Ruby. Thus, Chapter 3 introduces Ruby. The Ruby introduction is short because it assumes you already know another object-oriented programming language well, in this case Java. As mentioned above, we believe successful software engineers will need to routinely learn new languages and tools over their careers, so learning Ruby and Rails is good practice. Chapter 4 next introduces the basics of Rails. Readers already familiar with Ruby and Rails should skip these chapters.

Given this background, the next five chapters can illustrate important software engineering principles using Rails tools. Figure 1.5 shows one iteration of the Agile lifecycle, which we use as a framework on which to hang the next chapters of the



book.

Chapter 5 discusses how to talk to the customer. **[Behavior-Driven Design \(BDD\)](#)** advocates writing tests that customers without a programming background can read, called **[user stories](#)**, and Chapter 5 shows how to write user stories so that they can be turned into integration tests as well as acceptance tests. It introduces the **[Cucumber](#)** tool to help automate this task. This testing tool can be used with any language and framework, not just Rails. As SaaS apps are often user facing, the chapter also covers how to prototype a useful user interface using “Lo-Fi” prototyping.

Chapter 6 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 **[Autotest](#)** tool for automating test running, and the **[SimpleCov](#)** tool to measure test coverage.

Given the several chapters above to gain experience with Ruby and Rails basics, we return to Rails to explain more advanced features in Chapter 7. While more challenging to learn and understand, your application can be DRYer and more concise if you use concepts like partials, validations, lifecycle callbacks, filters, associations, and foreign keys. The chapter also explains how dynamic program synthesis (metaprogramming) in Rails improves productivity by automating the creation of JavaScript code for widely used AJAX scenarios, which are useful for

enriching your SaaS app’s user experience.



Chapter 8 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 [9](#) gives advice on how to organize and work as part of an effective team versus doing it all by yourself. It explains the term **Velocity** and how to use it to measure progress in the rate that you deliver features, and introduces the SaaS-based tool [**Pivotal Tracker**](#) to simplify such measurements. It also describes how the version control system [**Git**](#) and the corresponding services [**GitHub**](#) and [**ProjectLocker**](#) 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 [10](#) 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 [10](#) is about software architecture whereas prior chapters are about the Agile development process. We placed this chapter after the Agile process chapters as 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 [1.5](#) suggests, knowing design patterns will be useful when writing or refactoring code as part of the BDD/TDD process.

Building on familiarity with Ruby, Rails, and RSpec by this point in the book, Chapter [11](#) introduces the programming language JavaScript, its productive framework jQuery, and the testing tool Jasmine.

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

Chapter [13](#) summarizes the book, presents a survey from Berkeley alumni now in industry on the usefulness of the ideas in this book, and projects what is next.

Don't skip the screencasts. The temptation is to skip sidebars, elaborations, and screencasts to just skim the text until you find what you want to answer your question.

While elaborations are typically for experienced readers who want to know more about what is going on behind the curtain, and sidebars are just short asides that we think you'll enjoy, screencasts are *critical* to learning this material. While we wouldn't say you could skip the text and just watch the screencasts, we would say that they are some of the most important parts of the book. They allow us to express a lot of concepts, show how they interact, and demonstrate you can do the same tasks yourself. What would take many pages and be difficult to describe can

come alive in a two to five minute video. Screencasts allow us to follow Confucius' advice: "I see and I remember."

In our view, screencasts are the most important advantage of ebooks over print books, and they contain important information not found elsewhere in the book. So please watch them!

Learn by doing. The second point is to have your computer open with the Ruby interpreter ready so that you can try the examples in the screencasts and the text. We even make it easy to copy-and-paste the code using the service [Pastebin](#), (If you're reading the ebook, the link accompanying each code example will take you to that code example on Pastebin.) This practice follows Confucius' advice of "I do and I understand." Specific opportunities to learn by doing are highlighted by a bicycle icon.



There are topics that you will need to study to learn, especially in our buzzword-intensive ecosystem of Agile + Ruby + Rails + SaaS + Cloud Computing. Indeed, Figure 13.2 in Chapter 13 lists nearly 120 new terms introduced in just the first three chapters. To help you learn them, each term is linked to the appropriate [Wikipedia](#) article the first time it appears. We also use icons to remind you of the common themes throughout the book, which Figure 1.6 summarizes as a single handy place to look them up.



Beautiful Code



Legacy Code



Convention over Configuration



Don't Repeat Yourself (DRY)



Clarity via Conciseness



Learn By Doing



Productivity via Automation



Productivity via Code Generation



Productivity via Reuse



Productivity via Tools



Fallacy



Pitfall

Figure 1.6: Summary of Icons used in the book.

Depending on your background, we suspect you'll need to read some chapters more than once before you get the hang of it. To help you focus on the key points, each section includes **self-check questions** with answers and a **summary** of the key concepts in that section, and each chapter ends with **Fallacies and Pitfalls** explaining common misconceptions or problems that are easy to experience if

you're not vigilant. **Exercises** at the end of each chapter are more open-ended than the self-check questions; some have solutions posted at the book's website, saasbook.info. **Sidebars** highlight what we think are interesting asides about the material. In particular, to give readers a perspective about who came up with these big ideas that they are learning and that information technology relies upon, we use sidebars to introduce 20 Turing Award winners. (As there is no Nobel Prize in IT, our highest honor is known as the "Nobel Prize of Computing".)

We deliberately chose to keep the book concise, since different readers will want additional detail in different areas. Links are provided to the Ruby and Rails online documentation for built-in classes or methods, to Wikipedia for definitions of important concepts you may be unfamiliar with, and to the Web in general for further reading related to the material. If you're using the Interactive Edition or the Kindle edition, the links should be live if you're connected to the Internet; in the print version, the link URIs appear at the end of each chapter.

Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves. *Alan Kay, ACM Queue, 2005*

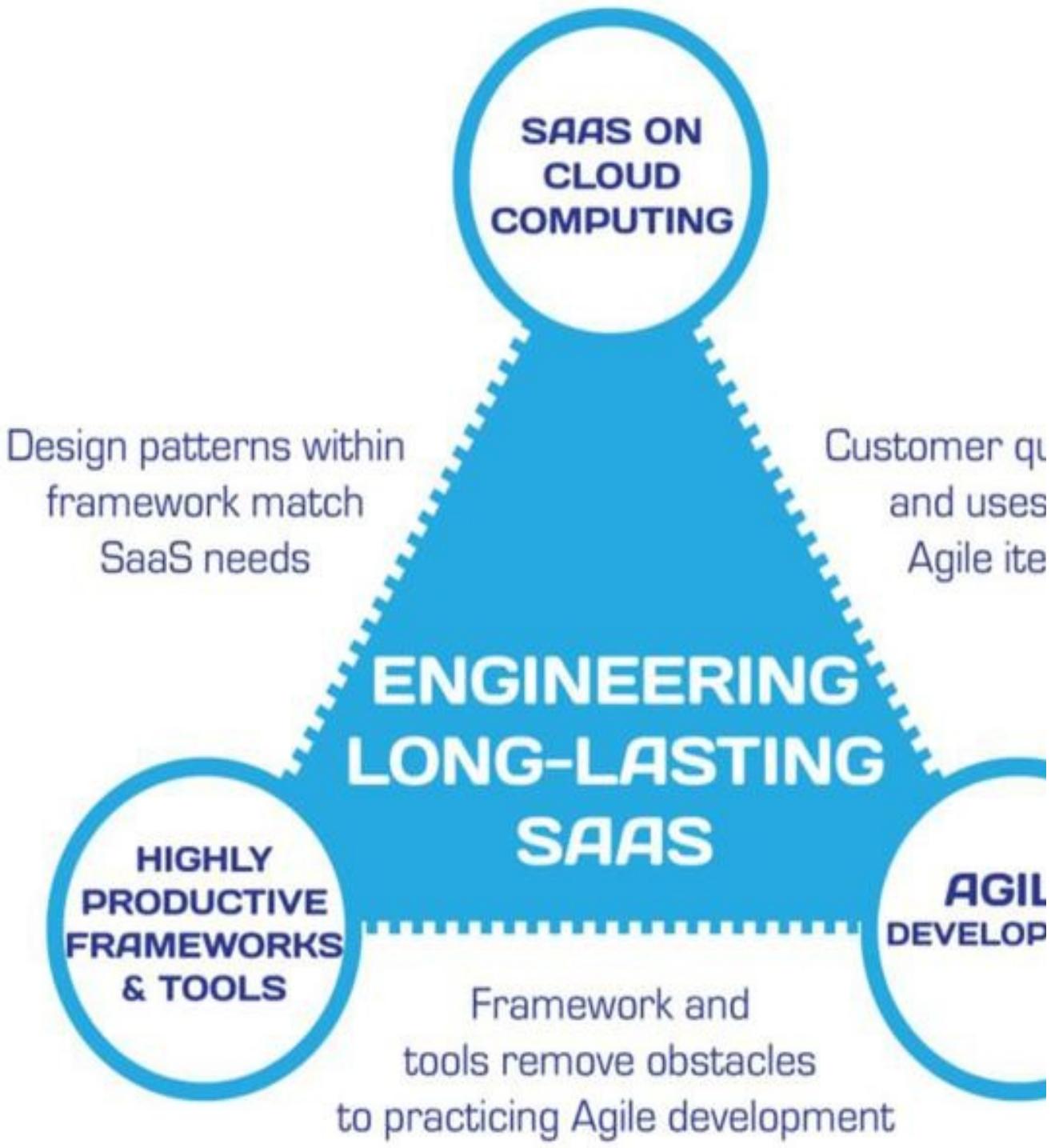


Figure 1.7: The Virtuous Triangle of Engineering long-lasting SaaS is formed from the three software engineering crown jewels of (1) SaaS on Cloud Computing, (2) Highly Productive Framework and Tools, and (3) Agile Development.

Engineering software shares the assurance and productivity goals and techniques of engineering hardware. However, the long-lasting and evolvable nature of successful software versus the relatively short life of successful computer hardware has led to different development processes. Inspired by hardware development, the Waterfall and Spiral lifecycles can be best for well-defined and slowly changing hardware and software products. The Agile lifecycle is a much better match to the rapidly

changing nature of Software as a Service (SaaS) delivered via Cloud Computing. Following such a software development process increases the chances that your software will be dependable, pleasing to the customer, long lasting, and considered



beautiful code by your peers.

Figure 1.7 shows the synergistic relationship between SaaS on Cloud Computing, Highly Productive Frameworks and Tools, and Agile Development, which are the three foundations of this book. SaaS on Cloud Computing is synergistic with Highly Productive Frameworks that expose design patterns such as Model–View–Controller that help SaaS (see Chapter 2). 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 Chapter 5). Highly Productive Frameworks and Tools designed to support Agile development remove obstacles to practicing the methodology (see Chapters 6 and 9). We believe these three “crown jewels” form a “virtuous triangle”



that leads to the engineering of beautiful, long-lasting Software as a Service. 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!

We believe if you learn the contents of this book and use the “bookware” that comes with it, you can build your own (simplified) version of a popular software service like FarmVille or Twitter. Indeed, we're amazed that today you can learn this valuable technology in such a short time; the main reason we wrote this book is to help more people become aware of and take advantage of this extraordinary opportunity. 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 us becoming some of its passionate fans!

1.1.3 To Learn More, 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/pdf/10.2200/S00193ED1V01Y200905CAC006>.

B. W. Boehm. A spiral model of software development and enhancement. In *ACM SIGSOFT Software Engineering Notes*, 1986.

M. Fowler. The New Methodology. *martinfowler.com*, 2005. URL <http://www.martinfowler.com/articles/newMethodology.html>.

G. Klein and et al. seL4: Formal verification of an OS kernel. *Communications of the ACM (CACM)*, 53(6):107–115, June 2010.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*. Morgan Kaufmann, 2008. ISBN 0123744938.

Project 1.1. (Discussion) In your opinion, how would you rank the software disasters in the first section from most terrible to the least? How did you rank them?

Project 1.2. (Discussion) The closest hardware failure to the software disasters mentioned in the first section is probably the [Intel Floating Point Divide bug](#). Where would you put this hardware problem in the ranked list of software examples from the exercise above?

Project 1.3. (Discussion) Measured in lines of code, what is the largest program in the world? For purposes of this exercise, assume it can be a suite of software that is shipped as a single product.

Project 1.4. (Discussion) Which programming language has the most active programmers?

Project 1.5. (Discussion) In which programming language is the most number of lines of code written annually? Which has the most lines of active code cumulatively?

Project 1.6. (Discussion) Make a list of, in your opinion, the Top 10 most important applications. Which would best be developed and maintained using a Waterfall lifecycle versus a Spiral lifecycle versus an Agile lifecycle? List your reasons for each choice.

Project 1.7. (Discussion) Given the list of Top 10 applications from the exercise above, how important are each of the four productivity techniques listed above?

Project 1.8. (Discussion) Given the list of Top 10 applications from the exercise above, what aspects might be difficult to test and need to rely on formal methods? Would some testing techniques be more important for some applications than others? State why.

Project 1.9. (Discussion) What are the top 5 reasons that SaaS and Cloud Computing will grow in popularity and the Top 5 obstacles to its growth?



Dennis Ritchie (left, 1941–2011) and Ken Thompson (right, 1943–) shared 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

- 2.1 [100,000 Feet: Client-Server Architecture](#)
- 2.2 [50,000 Feet: Communication—HTTP and URIs](#)
- 2.3 [10,000 feet: Representation—HTML and CSS](#)
- 2.4 [5,000 Feet: 3-Tier Architecture & Horizontal Scaling](#)
- 2.5 [1,000 Feet: Model-View-Controller Architecture](#)
- 2.6 [500 Feet: Active Record for Models](#)
- 2.7 [500 feet: Routes, Controllers, and REST](#)
- 2.8 [500 feet: Template Views](#)
- 2.9 [Fallacies and Pitfalls](#)
- 2.10 [Concluding Remarks: Patterns, Architecture, and Long-Lived APIs](#)
- 2.11 [To Learn More](#)
- 2.12 [Suggested Projects](#)

Whether creating a new system or preparing to modify an existing one, understanding its architecture at multiple levels is essential. Happily, good software leverages patterns at many levels—proven solutions to similar architectural problems, adapted to the needs of a specific problem. Judicious use of patterns helps simplify design, reveal intent, and compose software components into larger systems. We'll examine the patterns present at various logical layers of SaaS apps, discuss why each pattern was chosen, and where appropriate, note the opportunity cost of not choosing the alternative. Patterns aren't perfect for every problem, but the ability to separate the things that change from those that stay the same is a powerful tool for organizing and implementing large systems.



Since the best way to learn about software is by doing, let's jump in right away. If you haven't done so already, turn to Appendix A and get this book's "bookware" running on your own computer or in the cloud. Once it is ready, Screencast 2.1.1 shows how to deploy and login to your Virtual Machine and try an interaction with the simple educational app Rotten Potatoes, which aspires to be a simplified version of the popular movie-rating Web site [RottenTomatoes](#).

Screencast 2.1.1: [Getting Started](#)

Once logged in to your VM, the screencast shows how to open a Terminal window, cd (change to) the directory Documents/rottenpotatoes, and start the Rotten Potatoes app by typing rails server. We then opened the Firefox web browser and entered http://localhost:3000/movies into the address bar and pressed Return, taking us to the Rotten Potatoes home page.

What's going on? You've just seen the simplest view of a Web app: it is an example of the [client-server architecture](#). Firefox is an example of a client: a program whose specialty is asking a server for information and (usually) allowing the user to interact with that information. WEBrick, which you activated by typing rails server, is an example of a server: a program whose specialty is waiting for clients to make a *request* and then providing a *reply*. WEBrick waits to be contacted by a Web browser such as Firefox and routes the browser's requests to the Rotten Potatoes app. Figure 2.1 summarizes how a SaaS application works, from 100,000 feet.

1. A Web client (Firefox) requests the Rotten Potatoes home page from a Web server
 2. WEBrick obtains content from the Rotten Potatoes app and sends this content back to the client
 3. Firefox displays the content and closes the HTTP connection.
-

Figure 2.1: 100,000-foot view of a SaaS client-server system.

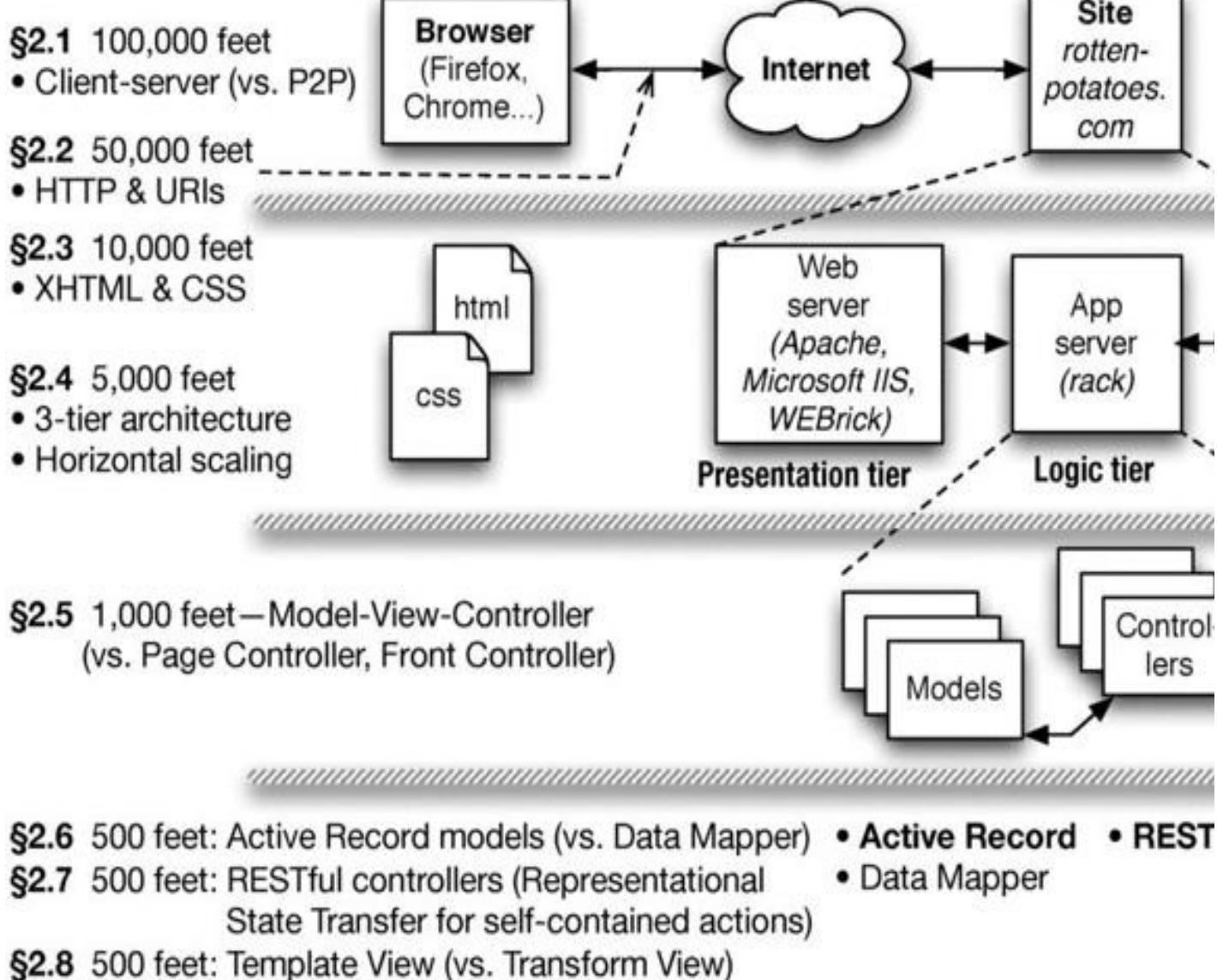


Figure 2.2: Using altitude as an analogy, this figure illustrates important structures in SaaS at various levels of detail and serves as an overall roadmap of the discussion in this chapter. Each level is discussed in the sections shown.

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. Firefox and other browsers (Chrome, Safari, Internet Explorer) are clients used by millions of people (let's call them *production clients*). WEBrick, on the other hand, is not a production server, but a “mini-server” with just enough functionality to let one user at a time (you, the developer) interact with your Web app. A real Web site would use a production server such as the [Apache web server](#) or the [Microsoft Internet Information Server](#), either of which can be deployed on hundreds of computers efficiently serving many copies of the same site to millions of users. Before the Web’s open standards were proposed in 1990, users would install separate and mutually-incompatible proprietary clients for each Internet service

they used: Eudora (the ancestor of Thunderbird) for reading email, AOL or CompuServe for accessing proprietary content portals (a role filled today by portals like MSN and Yahoo!), and so on. Today, the Web browser has largely supplanted proprietary clients and is justifiably called the “universal client.” Nonetheless, the proprietary clients and servers still constitute examples of client-server architecture, with clients specialized for asking questions on behalf of users and servers specialized for answering questions from many clients. 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. Because of the Web’s ubiquity, we will use the term SaaS to mean “client-server systems built to operate using the open standards of the World Wide Web.” In the past, the client-server architecture implied that the server was a much more complex program than the client. Today, with powerful laptops and Web browsers that support animation and 3D effects, a better characterization might be that clients and servers are comparably complex but have been specialized for their very different roles. In this book we will concentrate on server-centric applications; although we cover some JavaScript client programming in Chapter [11](#), its context is in support of a server-centric application rather than for building complex in-browser applications such as [Google Docs](#).

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.

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.
- 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 2.1.1. What is the primary difference between a client and a server 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.

Self-Check 2.1.2. What element(s) in Figure 2.2 refer to a SaaS client and what element(s) refer to a SaaS server?

- Ⓑ The **Browser** box in the upper-left corner refers to a client. The **html** and **css** document icons refer to content delivered to the client. All other elements are part of the server.



Vinton E. "Vint" Cerf (left, 1943–) and Bob Kahn (right,

1938–) shared the 2004 Turing Award for their pioneering work on networking architecture and protocols, including TCP/IP.

A **network protocol** is a set of communication rules on which agents participating in a network agree. In this case, the agents are Web clients (like Firefox) and Web servers (like WEBrick or Apache). Browsers and Web servers communicate using the **HyperText Transfer Protocol**, or **HTTP**. Like many Internet application protocols, HTTP relies on **TCP/IP**, the venerable **Transmission Control Protocol/Internet Protocol**, which allows a pair of agents to communicate ordered sequences of bytes. Essentially, TCP/IP allows the communication of arbitrary character strings between a pair of network agents.

In a TCP/IP network, each computer has 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 based on TCP/IP, is automatically invoked to map easy-to-remember **hostnames** like `www.eecs.berkeley.edu` to IP addresses. Browsers automatically contact a DNS server to look up the site name you type in the address bar, such as `www.eecs.berkeley.edu`, and get the actual IP address, in this case 128.32.244.172. A convention used by TCP/IP-compatible computers is that if a program running on a computer refers to the name `localhost`, it is referring to the very computer it's running on. That is why typing `localhost` into Firefox's address bar at the beginning of Section 2.1 caused Firefox to communicate with the WEBrick process running on the same computer as Firefox itself.

ELABORATION: Networking: multi-homing, IPv6 and HTTPS.

We have simplified some aspects of TCP/IP: technically each **network interface device** has an IP address, and some **multi-homed** computers may have multiple network interfaces. Also, for various reasons including the exhaustion of the address space of IP numbers, the current version of IP (version 4) is slowly being phased out in favor of version 6 (IPv6), which uses a different format for addresses. However, since most computers have only one network interface active at a time and SaaS app writers rarely deal directly with IP addresses, these simplifications don't materially alter our explanations. We also defer discussion of the Secure HTTP protocol (HTTPS) until Chapter 12. HTTPS uses **public-key cryptography** to encrypt (encode) communication between an HTTP client and server, so that an eavesdropper sees only gibberish. From a programmer's point of view, HTTPS behaves like HTTP, but only works if the Web server has been configured to support HTTPS access to certain pages. "Mini" servers like WEBrick typically don't support it.

What about the :3000 we appended to localhost in the example? Multiple agents on a network can be running at the same IP address. Indeed, in the example above, both the client and server were running on your own computer. Therefore, TCP/IP uses **port numbers** from 1 to 65535 to distinguish different network agents at the same IP address. All protocols based on TCP/IP, including HTTP, must specify both the host and port number when opening a connection. When you directed Firefox to go to localhost :3000/movies, you were indicating that on the computer called localhost (that is, "this computer"), a server program was monitoring port 3000 waiting for browsers to contact it. If we didn't specify the port number (3000) explicitly, it would default to 80 for http or 443 for https (secure) connections.

The IANA. The [Internet Assigned Numbers Authority](#) assigns official default port numbers for various protocols and manages the top-level or "root" zone of DNS.

To summarize, communication in HTTP is initiated when one agent *opens a connection* to another agent by specifying a hostname and port number; an HTTP server process must be *listening for connections* on that host and port number.

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.

The string `http://localhost:3000/movies` that you typed into Firefox's address bar is a **URI**, or **Uniform Resource Identifier**. A URI begins with the name of the communication *scheme* by which the information may be retrieved, followed by a hostname, optional port number, and a *resource* on that host that the user wants to retrieve. A resource generally means "anything that can be delivered to the browser": an image, the list of all movies in HTML format, and a form submission that creates a new movie are all examples of resources. Each SaaS application has its own rules for interpreting the resource name, though we will soon see one proposal called REST that strives for simplicity and consistency in resource naming across different SaaS apps.

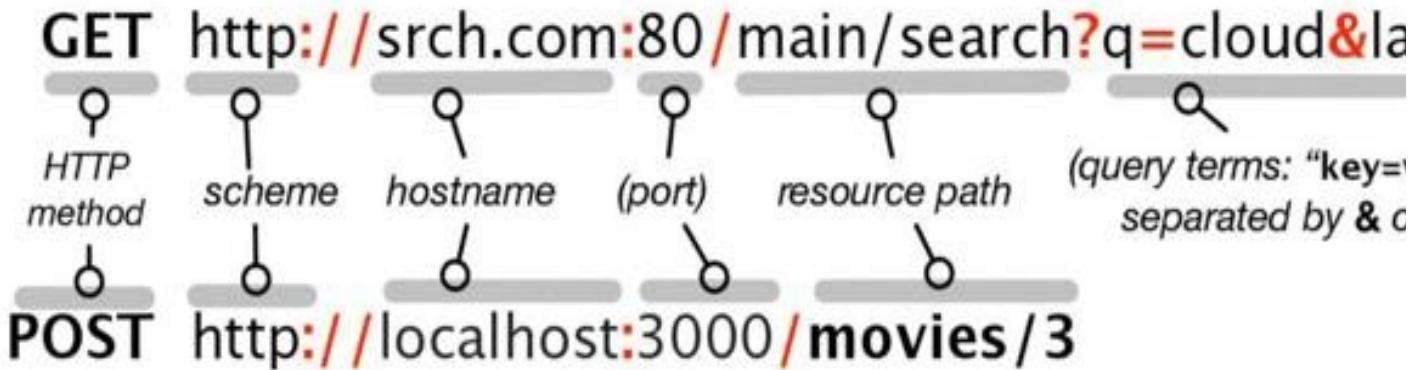


Figure 2.3: An HTTP request consists of an [HTTP method](#) plus a [URI](#). A *full URI* begins with a scheme such as `http` or `https` and includes the above components. Optional components are in parentheses. A *partial URI* omits any or all of the leftmost components, in which case those components are filled in or resolved relative to a *base URI* determined by the specific application. Best practice is to use full URLs.

HTTP is a [stateless protocol](#) because every HTTP request is independent of and unrelated to all previous requests. A web app that keeps track of “where you are” (Have you logged in yet? What step of the checkout process are you on?) must have its own mechanisms for doing so, since nothing about an HTTP request remembers this information. HTTP [cookies](#) associate a particular user’s browser with information held at the server corresponding to that user’s [session](#), but it is the browser’s responsibility, not HTTP’s or the SaaS app’s, to make sure the right cookies are included with each HTTP request. Stateless protocols therefore simplify server design at the expense of application design, but happily, successful frameworks such as Rails shield you from much of this complexity.

Screencast 2.2.1: [Cookies](#)

Cookies are used to establish that two independent requests actually originated from the same user’s browser, and can therefore be thought of as part of a session. On the first visit to a site, the server includes a long string (up to 4 KBytes) with the `Set-Cookie`: HTTP response header. It is the browser’s responsibility to include this string with the `Cookie`: HTTP request header on subsequent requests to that site. The cookie string, which is intended to be opaque to the user, contains enough information for the server to associate the request with the same user session. (*Note: The version of RottenPotatoes used in this Alpha edition screencast is slightly different from the version used in the other examples, but it doesn’t materially affect the explanation in the screencast.*)

We can now express what’s happening when you load the Rotten Potatoes home page in slightly more precise terms, as Figure 2.4 shows.

1. A Web client (Firefox) requests the Rotten Potatoes home page from a Web server
 - a) Firefox constructs an HTTP request using the URI **`http://localhost:3000`** to contact the server (WEBrick) listening on port 3000 on the same computer as Firefox itself.
 - b) WEBrick, listening on port 3000, receives the HTTP request for the resource '/n' (all movies in Rotten Potatoes).
2. WEBrick obtains content from the Rotten Potatoes app and sends this content back to Firefox.
3. Firefox displays the content and closes the HTTP connection.

Figure 2.4: At 50,000 feet, we can expand Step 1 from Figure 2.1.

To drill down further, we'll next look at how the content itself is represented.

Summary

- Web browsers and servers communicate using the ***HyperText Transfer Protocol***. HTTP relies on ***TCP/IP*** (Transmission Control Protocol/Internet Protocol) to reliably exchange ordered sequences 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.
- To run a SaaS app locally, you activate an HTTP server listening on a port on localhost. WEBrick, Rails' lightweight server, uses port 3000.
- A ***Uniform Resource Identifier*** (URI) names a resource available on the Internet. The interpretation of the resource name varies from application to application.
- 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.

ELABORATION: Client Pull vs. Server Push.

The Web is primarily a *client pull* client-server architecture because the client initiates all interactions—HTTP servers can only wait for clients to contact them. This is because HTTP was designed as a **request-reply protocol**: only clients can initiate anything. Evolving standards, including WebSockets and HTML5, have some support for allowing the server to *push* updated content to the client. In contrast, true *server push* architectures, such as text messaging on cell phones, allow the server to initiate a connection to the client to “wake it up” when new information is available; but these cannot use HTTP. An early criticism of the Web’s architecture was that a pure request-reply protocol would rule out such ***push-based*** applications, but in practice, the high efficiency of specialized server software supports creating Web pages that frequently *poll* (check in with) the server to receive updates, giving the user the illusion of a push-based application even without the features proposed in WebSockets and HTML5.

Self-Check 2.2.1. What happens if we visit the URI `http://google.com:3000` and why?

 The connection will eventually “time out” unable to contact a server, because Google (like almost all Web sites) listens on TCP port 80 (the default) rather than 3000.

Self-Check 2.2.2. What happens if we try to access Rotten Potatoes at (say) `http://localhost:3300` (instead of :3000) and why?

 You get a “connection refused” since nothing is listening on port 3300. If the Web browser is the universal client, **HTML**, the HyperText Markup Language, is the universal language. A **markup language** combines text with markup (annotations about the text) in a way that makes it easy to syntactically distinguish the two. Watch Screencast [2.3.1](#) for some highlights of HTML 5, the current version of the language, then continue reading.

Screencast 2.3.1: [HTML Introduction](#)

HTML consists of a hierarchy of nested elements, each of which consists of an opening tag such as `<p>`, a content part (in some cases), and a closing tag such as `</p>`. Most opening tags can also have attributes, as in ``. Some tags that don’t have a content part are self-closing, such as `<br clear="both"/>` for a line break that clears both left and right margins.

The use of angle brackets for tags comes from **SGML** (Standard Generalized Markup Language), a

codified standardization of IBM's General Markup Language, developed in the 1960s for encoding computer-readable project documents.

There is an unfortunate and confusing mess of terminology surrounding the [lineage of HTML](#). HTML 5 includes features of both its predecessors (HTML versions 1 through 4) and XHTML (eXtended HyperText Markup Language), which is a subset of [XML](#), an eXtensible Markup Language that can be used both to represent data and to describe other markup languages. Indeed, XML is a common data representation for exchanging information *between* two services in a Service-Oriented Architecture, as we'll see in Chapter [6](#) when we extend Rotten Potatoes to retrieve movie information from a separate movie database service. The differences among the variants of XHTML and HTML are difficult to keep straight, and not all browsers support all versions. Unless otherwise noted, from now on when we say HTML we mean HTML 5, and we will try to avoid using features that aren't widely supported. Of particular interest are the HTML tag attributes `id` and `class`, because they figure heavily into connecting the HTML structure of a page with its visual appearance. The following screencast illustrates the use of Firefox's Web Developer toolbar to quickly identify the ID's and Classes of HTML elements on a page.

Screencast 2.3.2: [Inspecting the ID and Class attributes](#)

CSS uses [selector notations](#) such as `div#name` to indicate a `div` element whose `id` is *name* and `div.name` to indicate a `div` element with class *name*. Only one element in an HTML document can have a given `id`, whereas many elements (even of different tag types) can share the same `class`. All three aspects of an element—its tag type, its `id` (if it has one), and its `class` attributes (if it has any)—can be used to identify an element as a candidate for visual formatting.

For an extreme example of how much can be done with CSS, visit the [CSS Zen Garden](#).

As the next screencast shows, the [CSS \(Cascading Style Sheets\)](#) standard allows us to associate visual "styling" instructions with HTML elements by using the elements' classes and IDs. The screencast covers only a few basic CSS constructs, which are summarized in Figure [2.5](#). The Resources section at the end of the chapter lists sites and books that describe CSS in great detail, including how to use CSS for aligning content on a page, something designers used to do manually with HTML tables.

Screencast 2.3.3: [Introduction to CSS](#)

There are four basic mechanisms by which a selector in a CSS file can match an HTML element: by tag name, by class, by ID, and by hierarchy. If multiple selectors match a given element, the rules for which properties to apply are complex, so most

designers try to avoid such ambiguities by keeping their CSS simple. A useful way to see the “bones” of a page is to select CSS>Disable Styles>All Styles from the Firefox Web Developer toolbar. This will display the page with all CSS formatting turned off, showing the extent to which CSS can be used to separate visual appearance from logical structure.

Selector	What is selected
h1	Any h1 element
div#message	The div whose ID is message
.red	Any element with class red
div.red, h1	The div with class red, or any h1
div#message h1	An h1 element that's a child of (inside of) div#message
a.lnk	a element with class lnk
a.lnk:hover	a element with class lnk, when hovered over
Attribute	Example values
font-family	"Times, serif"
font-weight	bold
font-size	14pt, 125%, 12px
font-style	italic
color	black
margin	4px
background-color	red, #c2eed6 (RGB values)
border	1px solid blue
text-align	right
text-decoration	underline
vertical-align	middle
padding	1cm

Figure 2.5: A few CSS constructs, including those explained in Screencast [2.3.3](#). The top table shows some CSS *selectors*, which identify the elements to be styled; the bottom table shows a few of the many attributes, whose names are usually self-explanatory, and example values they can be assigned. Not all attributes are valid on all elements.

Using this new information, Figure [2.6](#) expands steps 2 and 3 from the previous section's summary of how SaaS works.

1. A Web client (Firefox) requests the Rotten Potatoes home page from a Web server
 - a) Firefox constructs an HTTP request using the URI ***http://localhost:3000*** to contact the server (WEBrick) listening on port 3000 on the same computer as Firefox itself.
 - b) WEBrick, listening on port 3000, receives the HTTP request for the resource '/n' (all movies in Rotten Potatoes).
2. WEBrick obtains content from the Rotten Potatoes app and sends this content back
 - a) WEBrick returns content encoded in HTML, again using HTTP. The HTML may also contain other kinds of media such as images to embed in the displayed page. The HTML may contain a reference to a CSS stylesheet containing formatting information describing the visual attributes of the page (font sizes, colors, layout, and so on).
3. Firefox displays the content and closes the HTTP connection.
 - a) Firefox fetches any referenced assets (CSS, images, and so on) by repeating the steps as needed but providing the URLs of the desired assets as referenced in the HTML.
 - b) Firefox displays the page according to the CSS formatting directives and includes assets such as embedded images.

Figure 2.6: SaaS from 10,000 feet. Compared to Figure 2.4, step 2 has been expanded to describe the content returned by the Web server, and step 3 has been expanded to describe the role of CSS in how the Web browser renders the content.

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.
- A **selector** is an expression that identifies one or more HTML elements in a document by using a combination of the element name (such as body), element id (an element attribute that must be unique on a page), and element class (an attribute that need not be unique on a page).
- **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. A special link element inside the head element of an HTML document associates a stylesheet with that document.
- The Firefox Web Developer toolbar is invaluable in peeking under the hood to examine both the structure of a page and its stylesheets.

Self-Check 2.3.1. True or false: every HTML element must have an ID.

 False—the ID is optional, though must be unique if provided.

Pastebin is the service we use to make it easy to copy-and-paste the code. (If you’re reading the ebook, the link accompanying each code example will take you to that code example on Pastebin; you need to type in the URI if you’re reading the print book.)

Self-Check 2.3.2. Given the following HTML markup:

<http://pastebin.com/bJJcbUap>

```
1   <p class="x" id="i">I hate <span>Mondays</span></p>
2   <p>but <span class="y">Tuesdays</span> are OK.</p>
```

Write down a CSS selector that will select *only* the word *Mondays* for styling.

 Three possibilities, from most specific to least specific, are: `#i span`, `p.x span`, and `.x span`. Other selectors are possible but redundant or over-constrained; for example, `p#i span` and `p#i.x span` are redundant with respect to this HTML snippet since at most one element can have the ID `i`.

Self-Check 2.3.3. In Self-Check 2.3.2, why are `span` and `p span` *not* valid answers?

 Both of those selector also match *Tuesdays*, which is a `span` inside a `p`.

Self-Check 2.3.4. What is the most common way to associate a CSS stylesheet with an HTML or HTML document? (HINT: refer to the earlier screencast example.)

 Within the HEAD element of the HTML or HTML document, include a LINK element with at least the following three attributes: REL="STYLESHEET", TYPE="text/css", and HREF="*uri*", where *uri* is the full or partial URI of the stylesheet. That is, the stylesheet must be accessible as a resource named by a URI. So far we've seen how the client communicates with the server and how the information they exchange is represented, but we haven't said anything about the server itself. Moving back to the server side of Figure 2.2 and zooming in for additional detail on the second level, Web apps are structured as three logical *tiers*. The ***presentation tier*** usually consists of an ***HTTP server*** (or simply "***Web server***"), which accepts requests from the outside world (i.e., users) and usually serves static assets. We've been using WEBrick to fulfill that role.

Because application servers sit between the Web server (presentation tier) and your actual app code, they are sometimes referred to as ***middleware***.

The web server forwards requests for dynamic content to the ***logic tier***, where the actual application runs that generates dynamic content. The application is typically supported by an ***application server*** whose job is to hide the low-level mechanics of HTTP from the app writer. For example, an app server can route incoming HTTP requests directly to appropriate pieces of code in your app, saving you from having to listen for and parse incoming HTTP requests. Modern application servers support

one or more **[Web application frameworks](#)** that simplify creation of a particular class of Web applications in a particular language. We will be using the Rails framework and the Rack application server, which comes with Rails. WEBrick can “speak” to Rack directly; other Web servers such as Apache require additional software modules to do so. If you were writing in PHP, Python, or Java, you would use an application server that handles code written in those languages. For example, Google AppEngine, which runs Python and Java applications, has proprietary middleware that bridges your app’s Python or Java code to the Google-operated infrastructure that faces the outside world.

Finally, since HTTP is stateless, application data that must remain stored across HTTP requests, such as session data and 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.

LAMP. Early SaaS sites were created using the Perl and 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.

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. In fact, Rotten Potatoes has been doing just this: its presentation tier is just WEBrick, and its persistence tier is a simple open-source database called SQLite, which stores its information directly in files on your local computer. In production, it’s more common for each tier to span one or more physical computers. As Figure [2.7](#) shows, in a typical site, incoming HTTP requests are directed to one of several Web servers, which in turn select 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 layer shared-nothing is much more complicated. Figure [2.7](#) shows the **[master-slave](#)** approach, used when the database is read much more frequently than it is written: any slave can perform reads, only the master can perform writes, and the master updates the slaves with the results of writes as quickly as possible. However, in the end, this technique only postpones the scaling problem rather than solving it. As one of [Heroku’s](#) 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](#)

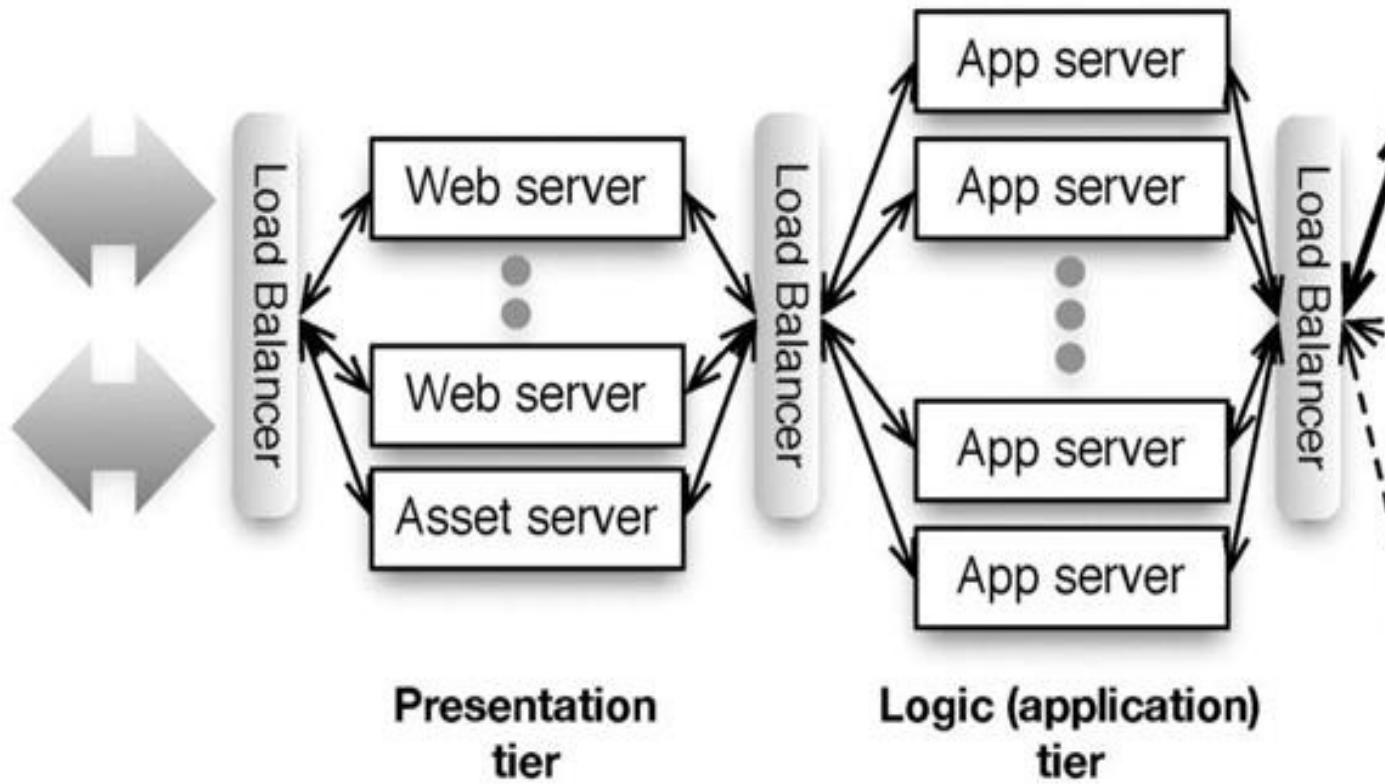


Figure 2.7: The 3-tier ***shared-nothing*** architecture, so called because entities within a tier generally do not communicate with each other, allows adding computers to each tier independently to match demand. ***Load balancers***, which distribute workload evenly, can be either hardware appliances 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.

We can now add one more level of detail to our explanation; step 2a is new in Figure 2.8.

1. A Web client (Firefox) requests the Rotten Potatoes home page from a Web server
 - a) Firefox constructs an HTTP request using the URI **`http://localhost:3000`** to contact the server (WEBrick) listening on port 3000 on the same computer as Firefox itself.
 - b) WEBrick, listening on port 3000, receives the HTTP request for the resource '/n' (all movies in Rotten Potatoes).
2. WEBrick obtains content from the Rotten Potatoes app and sends this content back
 - a) Via the Rack middleware (written in Ruby), WEBrick calls Rotten Potatoes code in the tier. This code generates the page content using movie information stored in the persistence tier, implemented by a SQLite database using local files.
 - b) WEBrick returns content encoded in HTML, again using HTTP. The HTML may contain references to other kinds of media such as images to embed in the displayed page. The HTML may also contain a reference to a CSS stylesheet containing formatting information describing the visual attributes of the page (font sizes, colors, layout, and so on).
3. Firefox displays the content and closes the HTTP connection.
 - a) Firefox fetches any referenced assets (CSS, images, and so on) by repeating the previous steps as needed but providing the URLs of the desired assets as referenced in the HTML.
 - b) Firefox displays the page according to the CSS formatting directives and includes any assets such as embedded images.

Figure 2.8: SaaS from 5,000 feet. Compared to Figure 2.6, step 2a has been inserted, describing the actions of the SaaS server in terms of the three-tier architecture.

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 2.4.1. 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, but it is the shared-nothing architecture that makes it straightforward to "absorb" the new computers into a running app and "release" them when no longer needed.

Self-Check 2.4.2. In the ____ tier of three-tier SaaS apps, scaling is much more complicated than just adding computers.

 Persistence tier

So far we've said nothing about the structure of the app code in Rotten Potatoes. In fact, just as we used the client-server architectural pattern to characterize the "100,000-foot view" of SaaS, we can use an architectural pattern called ***Model-View-Controller*** (usually shortened to MVC) to characterize the "1,000-foot view."

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. In our simplified Rotten Potatoes app, there is only a Movie model, but we'll be adding others later. Because models deal with the application's data, they contain the code that communicates with the storage tier.

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 Rotten Potatoes 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 (e.g. by clicking something on a Web page), 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 Rotten Potatoes 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 other choices are possible, such as those in Figure 2.9 excerpted from [Martin Fowler's Catalog of Patterns of Enterprise Application Architecture](#). Apps consisting of mostly static content with only a small amount of dynamically-generated content, such as a weather site, might choose the *Template View* pattern. The *Page Controller* pattern works well for an application that is easily structured as a small number of distinct pages, effectively giving each page its own simple controller that only knows how to generate that page. For an application that takes a user through a sequence of pages (such as signing up for a mailing list) but has few models, the *Front Controller* pattern might suffice, in which a single controller handles all incoming requests rather than separate controllers handling requests for each model.

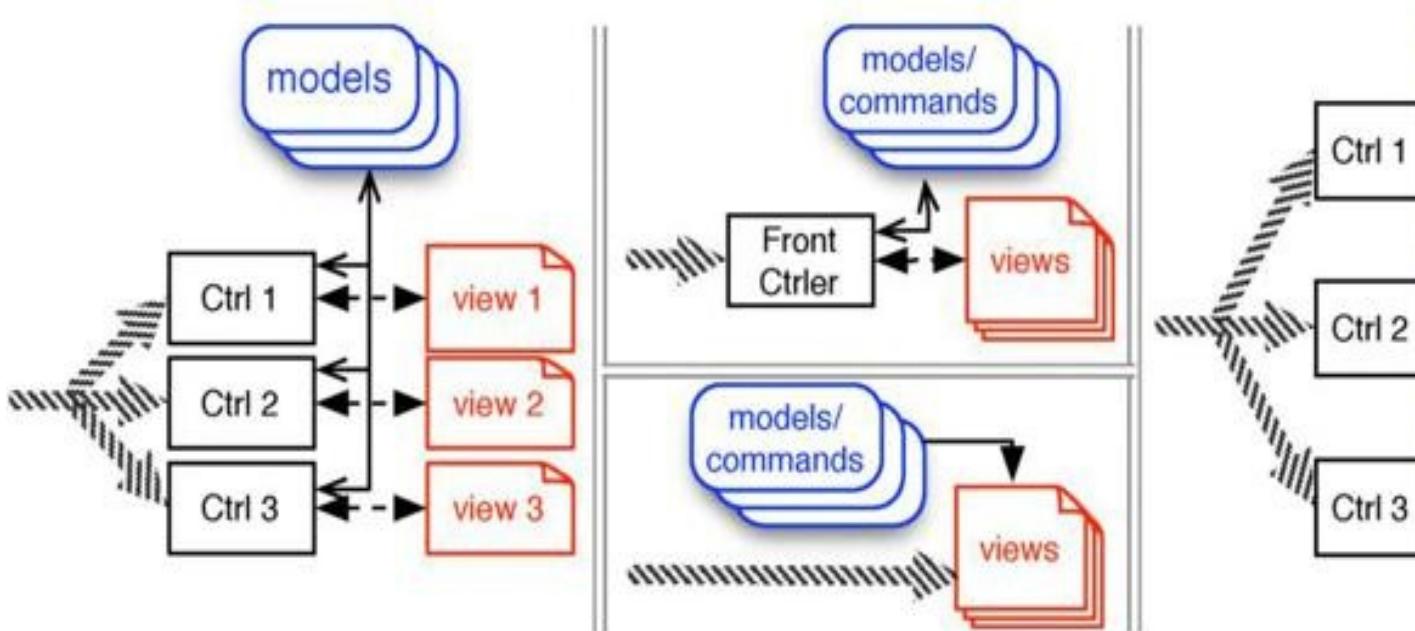


Figure 2.9: Comparing Web app architectural patterns. Models are rounded rectangles, controllers are rectangles, and views are document icons. Page Controller (left), used by Sinatra, has a controller for each logical page of the app. Front Controller (top center), used by Java 2 Enterprise Edition (J2EE) servlets, has a single controller that relies on methods in a variety of models to generate one of a

collection of views. Template View (bottom center), used by PHP, emphasizes building the app around the views, with logic in the models generating dynamic content in place of part of the views; the controller is implicit in the framework. Model-View-Controller (right), used by Rails and Java Spring, associates a controller and a set of views with each model type.

Figure 2.10 summarizes our latest understanding of the structure of a SaaS app.

1. A Web client (Firefox) requests the Rotten Potatoes home page from a Web server.
 - a) Firefox constructs an HTTP request using the URI ***http://localhost:3000*** to contact the server (WEBrick) listening on port 3000 on the same computer as Firefox itself.
 - b) WEBrick, listening on port 3000, receives the HTTP request for the resource '/n' (all movies in Rotten Potatoes).
2. WEBrick obtains content from the Rotten Potatoes app and sends this content back to Firefox.
 - a) Via the Rack middleware (written in Ruby), WEBrick calls Rotten Potatoes code.
This code generates the page content using movie information stored in the database implemented by a SQLite database using local files.
 - i) Rack routes the request to the ***index*** action of the Movies controller; the result of this route is the list of all movies.
 - ii) The Ruby function implementing the ***index*** action in the Movies controller asks the model for a list of movies and associated attributes.
 - iii) If successful, the controller identifies a View that contains the HTML markup for the list of movies, and passes it the movie information so that an HTML page can be generated. If it fails, the controller identifies a View that displays an error message.
 - iv) Rack passes the constructed view to WEBrick, which sends it back to Firefox as an HTTP reply.
 - b) WEBrick returns content encoded in HTML, again using HTTP. The HTML may contain references to other kinds of media such as images to embed in the displayed page. The HTML may also contain a reference to a CSS stylesheet containing formatting information describing the visual attributes of the page (font sizes, colors, layout, and so on).
3. Firefox displays the content and closes the HTTP connection.
 - a) Firefox fetches any referenced assets (CSS, images, and so on) by repeating the steps as needed but providing the URLs of the desired assets as referenced in the HTML.
 - b) Firefox displays the page according to the CSS formatting directives and includes the assets such as embedded images.

Figure 2.10: Step 2a has been expanded to show the role of the MVC architecture in fulfilling a SaaS app request.

Summary

- The ***Model-View-Controller*** or MVC design pattern 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, 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.
- MVC is appropriate for interactive SaaS apps with a variety of model types, where it makes sense to situate controllers and views along with each type of model. Other architectural patterns may be more appropriate for smaller apps with fewer models or a smaller repertoire of operations.

Self-Check 2.5.1. Which tier(s) in the three-tier architecture are involved in handling each of the following: (a) models, (b) controllers, (c) views?

(a) models: logic and persistence tiers; (b) controllers: logic and presentation tiers; (c) views: logic and presentation tiers.

How do the models, views, and controllers actually do their jobs? Again, we can go far by describing them in terms of patterns.

Every nontrivial application needs to store and manipulate persistent data. Whether using a database, a plain file, or other persistent storage location, we need a way to convert between the data structures or objects manipulated by the application code and the way that data is stored. In the version of Rotten Potatoes used in this chapter, the only persistent data is information about movies. Each movie's *attributes* include its title, release date, MPAA rating, and short "blurb" summarizing the movie. A naive approach might be to store the movie information in a plain text file, with one line of the file corresponding to one movie and attributes separated by commas:

<http://pastebin.com/sjD9Gu4M>

1 Gone with the Wind, G, 1939-12-15, An American classic ...

2 Casablanca, PG, 1942-11-26, Casablanca is a classic and...

To retrieve movie information, we would read each line of the file and splitting it into *fields* at the commas. Of course we will run into a problem with the movie Food, Inc. whose title contains a comma:

<http://pastebin.com/mz4wzzPm>

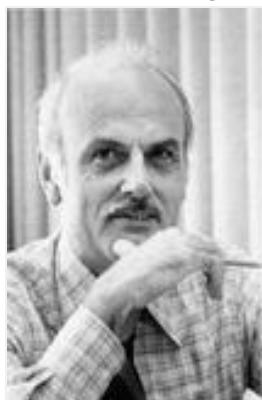
```
1     Food, Inc.,PG,2008-09-07,The current method of raw...
```

We might try to fix this by surrounding each field with quote marks:

<http://pastebin.com/UwYasCHE>

```
1
"Food, Inc.", "PG", "2008-09-07", "The current method of raw..."
```

...which will be fine until we try to enter the movie Waiting for "Superman". As this example shows, devising even a simple storage format involves tricky pitfalls, and would require writing code to convert an in-memory object to our storage representation (called **marshalling** or **serializing** the object) and vice versa (*unmarshalling* or *deserializing*).



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

Fortunately, the need to persist objects is so common that several design patterns have evolved to fulfill it. A subset of these patterns makes use of **structured storage**—storage systems that allow you to simply specify the desired structure of stored objects rather than writing explicit code to create that structure, and in some cases, to specify relationships connecting objects of different types. *Relational database management systems* (RDBMSs) evolved in the early 1970s as elegant structured storage systems whose design was based on a formalism for representing structure and relationships. We will discuss RDBMSs in more detail

later, but in brief, 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 Rotten Potatoes includes columns for title, rating, release_date, and description, and the rows of the table look like Figure 2.11.

	id	title	rating	release_date	description
1	Gone with the Wind	G	1939-12-15	An American classic ...	
2	Casablanca	PG	1942-11-26	Casablanca is a...	

Figure 2.11: A possible RDBMS table for storing movie information. The id column gives each row's **primary key** or permanent and unique identifier. 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.

Since it is the responsibility of the Models to manage the application's data, some correspondence must be established between the operations on a model object in memory (for example, an object representing a movie) and how it is represented and manipulated in the storage tier. The in-memory object is usually represented by a class that, among other things, provides a way to represent the object's attributes, such as the title and rating in the case of a movie. The choice made by the Rails framework is to use the [Active Record architectural pattern](#). In this pattern, a single instance of a model class (in our case, the entry for a single movie) corresponds to a single row in a specific table of an RDBMS. The model object has built-in behaviors that directly operate on the database representation of the object:

- 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).

This collection of four commands is often abbreviated [CRUD](#). Later we will add the ability for moviegoers to review their favorite movies, so there will be a one-to-many relationship or *association* between a moviegoer and her reviews; Active Record exploits existing mechanisms in the RDBMS based on foreign keys (which we'll learn about later) to make it easy to implement these associations on the in-memory objects.

Summary

- One important job of the Model in an MVC SaaS app is to persist data, which requires converting between the in-memory representation of an object and its representation in permanent storage.
- Various design patterns have evolved to meet this requirement, many of them making use of structured storage such as Relational Database Management Systems (RDBMSs) to simplify not only the storage of model data but the maintenance of relationships among models.
- The four basic operations supported by RDBMSs are Create, Read, Update, Delete (abbreviated CRUD).
- In the ActiveRecord design pattern, every model knows how to do the CRUD operations for its type of object. The Rails ActiveRecord library provides rich functionality for SaaS apps to use this pattern.

Self-Check 2.6.1. Which of the following are examples of structured storage: (a) an Excel spreadsheet, (b) a plain text file containing the text of an email message, (c) a text file consisting of names, with exactly one name per line.

(a) and (c) are structured, since an app reading those files can make assumptions about how to interpret the content based on structure alone. (b) is unstructured. Active Record gives each model the knowledge of how to create, read, update, and delete instances of itself in the database (CRUD). Recall from Section 2.5 that in the MVC pattern, controller actions mediate the user's Web browser interactions that cause CRUD requests, and in Rails, each controller action is handled by a particular Ruby method in a controller file. Therefore, each incoming HTTP request must be mapped to the appropriate controller and method. In Rails, this mapping is called a **route**.

In an unfortunate collision of terminology, an HTTP request is characterized by both its URI and the **HTTP method** used for the request, as we saw in Figure 2.3. The HTTP standard defines various methods including GET, POST, PUT, and DELETE. When discussing routes and controllers, to avoid confusion we will use *method* to mean the HTTP method portion of a request and *controller action* or simply *action* to mean the Ruby code (method) that handles a particular kind of request.

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

A route, then, associates a URI plus an HTTP method with a particular controller file and action. The HTTP standard defines a number of methods including GET, POST, PUT, and DELETE, which Rails uses in defining routes. Although Web browsers only implement GET when you follow a link and POST when you click the Submit button on a fill-in form, Rails works around this inconvenience and allows using all four methods in our routes. The route mappings are generated by code in the file

`config/routes.rb`, which we'll learn about in Chapter 3. In the Terminal window, make sure you are in the `rottenpotatoes` directory and type the command `rake routes` to show what routes are defined for Rotten Potatoes. Figure 2.12 explains the output. In a displayed URI such as `/movies/:id`, the tokens beginning with `:` act like wildcards, with `:id` representing the `id` attribute (primary key) of a model instance. For example, the route GET `/movies/8` will match line 5 of Figure 2.12 with `:id` having the value 8; therefore it is a request for an HTML page that shows details for the movie whose ID in the `Movies` table is 8, if such a movie exists.

`rake` runs maintenance tasks defined in Rotten Potatoes' `Rakefile`. `rake --help` shows other options.

Similarly, the route GET `/movies` matches line 1, requesting an HTML page that lists all the movies (the `Index` action), and the route POST `/movies` matches line 2 and creates a new movie entry in the database. (The POST `/movies` route doesn't specify an `id` because the new movie won't have an ID until after it's created in the database.) Lines 1 and 2 illustrate that routes with the same URI but different HTTP methods can map to different controller actions.

<http://pastebin.com/E7AWbUg0>

```
1 I   GET /movies           {:action=>"index", :controller=>"movies"}
2 C   POST /movies          {:action=>"create", :controller=>"movies"}
3     GET /movies/new       {:action=>"new", :controller=>"movies"}
4     GET /movies/:id/edit  {:action=>"edit", :controller=>"movies"}
5 R     GET /movies/:id     {:action=>"show", :controller=>"movies"}
6 U     PUT /movies/:id    {:action=>"update", :controller=>"movies"}
7 D DELETE /movies/:id    {:action=>"destroy", :controller=>"movies"}
```

Figure 2.12: The routes recognized by Rotten Potatoes, with the optional (`.:format`) tokens removed for clarity (see the Elaboration at the end of this section), showing which CRUD action each route represents. The additional `I` action is for `Index`, which lists members of collection. The rightmost column shows which Rails controller and which action (method) in that controller are called when a request

matches the given URI and HTTP method.

Returning to Figure 2.12, observe that the create route (line 2) and the new route (line 3) both appear to be involved in handling the creation of a new movie. Why are two routes needed for this action? The reason is that in a Web app, two interactions are required to create a new movie, as Screencast 2.7.1 shows.

Screencast 2.7.1: [Create and Update each require two interactions](#)

Creating a new movie requires two interactions with Rotten Potatoes, because before the user can submit information about the movie he must be presented with a form in which to enter that information. The empty form is therefore the resource named by the route in line 3 of Figure 2.12, and the submission of the filled-in form is the resource named by the route in line 2. Similarly, updating an existing movie requires one resource consisting of an editable form showing the existing movie info (line 4) and a second resource consisting of the submission of the edited form (line 6).

To summarize, routes are part of the middleware that connect URIs to specific controller methods in Rails. When a URI matching a defined route is received, the controller action indicated in the rightmost column is called to handle that request. In addition, wildcards such as :id that were matched in the URI are made available to the controller action as variables.

The routes in Figure 2.12 have an important property that is easy to overlook: they allow any CRUD request to be expressed by a URI that is self-contained, including all the information needed to satisfy that request. In particular, the result of a request, or the information needed to satisfy the request, does *not* depend on requests that have come before—a good fit to the statelessness of HTTP.

The philosophy that externally-visible Application Programming Interfaces (APIs) should expose only self-contained operations is called ***REpresentational State Transfer*** or ***REST***. Services and APIs that follow this principle are said to be RESTful, so we can say that Figure 2.12 depicts RESTful routes and their corresponding RESTful URIs. Although simple to explain, REST is an unexpectedly powerful organizing principle for SaaS applications, because it makes the designer think carefully about exactly what conditions or assumptions each request depends on in order to be self-contained. A RESTful interface simplifies participating in a Service-Oriented Architecture because if every request is self-contained, interactions between services don't need to establish or rely on the concept of an ongoing session, as many SaaS apps do when interacting with human users.

<http://pastebin.com/u/saasbook>

	Non-RESTful Site	RESTful Site
1		
2		
3 Login to site: e	POST /login/dave	POST /login/dav
4 Welcome page: welcome	GET /welcome	GET /user/301/ welcome
5 Add item ID 427 to cart: add/427	POST /add/427	POST /user/301/ add/427
6 View cart: cart	GET /cart	GET /user/301/ cart
7 Pay: checkout	POST /checkout	POST /user/301/ checkout

Figure 2.13: Non-RESTful and RESTful ways of mapping site functionality to HTTP methods and URIs. 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.

As a concrete illustration of the REST principle, Figure 2.13 shows both a RESTful and non-RESTful way that an 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 request after the login (line 3) relies on implicit information: line 4 assumes the site “remembers” who the currently-logged-in user is to show him his welcome page, and line 7 assumes 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

- An HTTP request consists of both a URI and an HTTP request method (GET, POST, PUT, or DELETE).
- A Rails route maps a request (URI+method) to a specific controller action that handles that request. In keeping with MVC terminology, the term “controller

“action” may refer either to the logical action itself or to the Ruby code in a controller that implements that action.

- If a URI is self-contained and includes all the information needed to satisfy the request, it’s said to be RESTful (for REpresentational State Transfer). Rails generates RESTful routes by default, making it easy to integrate your user-facing Rails SaaS app into a Service-Oriented Architecture.
-

ELABORATION: PUT and DELETE.

HTTP’s PUT and DELETE methods, used in actions such as Update (line 6 of Figure 2.12), are not supported by Web browsers for historical reasons. To compensate, Rails includes a special mechanism that “decorates” submissions that should use PUT and DELETE in a special way and lets the Web browser submit the request using POST instead. A complementary mechanism “un-decorates” the specially-decorated request and 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 the PUT and DELETE methods are actually supported. Both mechanisms are part of the Rails code that implements route mappings.

ELABORATION: Optional :format in routes

The raw output of `rake routes` includes a token (`. :format`) in most routes, which we omitted for clarity in Figure 2.12. If present, the format specifier allows a route to request resources in an output format other than the default of HTML—for example, GET `/movies.xml` would request the list of all movies as an XML document rather than an HTML page. Although in this simple application we haven’t included the code to generate formats other than HTML, this mechanism allows a properly-designed existing application to be easily integrated into a Service-Oriented Architecture—changing just a few lines of code allows all existing controller actions to become part of an external RESTful API.

Self-Check 2.7.1. True or false: If an app has a RESTful API, it must be performing CRUD operations.

 False. The REST principle can be applied to any kind of operation.

Self-Check 2.7.2. True or false: Supporting RESTful operations simplifies integrating a SaaS app with other services in a Service-Oriented Architecture.

 True

We conclude our brief tour with a look at Rails views. Because Web applications primarily deliver HTML pages, most frameworks provide a way to create a page of static markup (HTML or otherwise) interspersed with variables or very brief snippets of code. At runtime, the variable values or results of code execution are substituted or **interpolated** into the page. This architecture is known as Template View, and it is used by Rails’ views subsystem as well as being the basis of frameworks such as PHP.

We prefer Haml’s conciseness to Rails’ built-in `erb` templating system, so Haml is preinstalled with the bookware.

We will use a templating system called Haml (for HTML Abstraction Markup Language, pronounced “HAM-ell”) to streamline the creation of HTML template

views. We will learn more details and create our own views in Chapter 3, but in the interest of visiting all the “moving parts” of a Rails app, open app/views/movies/index.html.haml in the Rotten Potatoes directory. This is the view used by the Index controller action on movies; by convention over configuration, the suffixes .html.haml indicate that the view should be processed using Haml to create index.html, and the location and name of the file identify it as the view for the index action in the movies controller. Screencast 2.8.1 presents

the basics of Haml, summarized in Figure 2.14.



Screencast 2.8.1: [Interpolation into views using Haml](#)

In a Haml template, lines beginning with % expand into the corresponding HTML opening tag, with no closing tag needed since Haml uses indentation to determine structure. Ruby-like hashes following a tag become HTML attributes. Lines – beginning with a dash are executed as Ruby code with the result discarded, and lines =beginning with an equals sign are executed as Ruby code with the result interpolated into the HTML output.

Haml	HTML
%br{:clear => 'left'}	<br clear="left"/>
%p.foo Hello	<p class="foo">Hello</p>
%p#foo Hello	<p id="foo">Hello</p>
.foo	<div class="foo">. . . </div>
#foo.bar	<div id="foo" class="bar">. . . </div>

Figure 2.14: Some commonly used Haml constructs and the resulting HTML. A Haml tag beginning with % must either contain the tag and all its content on a single line, as in lines 1–3 of the table, or must appear by itself on the line as in lines 4–5, in which case all of the tag’s content must be indented by 2 spaces on subsequent lines. Notice that Haml specifies class and id attributes using a notation deliberately similar to CSS selectors.

According to MVC, views should contain as little code as possible. Although Haml technically permits arbitrarily complex Ruby code in a template, its syntax for including a multi-line piece of code is deliberately awkward, to discourage programmers from doing so. Indeed, the only “computation” in the Index view of Rotten Potatoes is limited to iterating over a collection (provided by the Model via the Controller) and generating an HTML table row to display each element. In contrast, applications written using the PHP framework often mingle large

amounts of code into the view templates, and while it's possible for a disciplined PHP programmer to separate the views from the code, the PHP framework itself provides no particular support for doing this, nor does it reward the effort. MVC advocates argue that distinguishing the controller from the view makes it easier to think first about structuring an app as a set of RESTful actions, and later about rendering the results of these actions in a separate View step. Section [1.7](#) made the case for Service-Oriented Architecture; it should now be clear how the separation of models, views and controllers, and adherence to a RESTful controller style, naturally leads to an application whose actions are easy to "externalize" as standalone API actions.

ELABORATION: Alternatives to Template View

Because all Web apps must ultimately deliver HTML to a browser, building the output (view) around a static HTML "template" has always made sense for Web apps, hence the popularity of the Template View pattern for rendering views. That is, the input to the view-rendering stage includes both the HTML template and a set of Ruby variables that Haml will use to "fill in" dynamic content. An alternative is the Transform View pattern ([Fowler 2002](#)), in which the input to the view stage is *only* the set of objects. The view code then includes all the logic for converting the objects to the desired view representation. This pattern makes more sense if many different representations are possible, since the view layer is no longer "built around" any particular representation. An example of Transform View in Rails is a set of Rails methods that accept ActiveRecord resources and generate pure-XML representations of the resources—they do not instantiate any "template" to do so, but rather create the XML starting with just the ActiveRecord objects. These methods are used to quickly convert an HTML-serving Rails app into one that can be part of a Service-Oriented Architecture.

Self-Check 2.8.1. What is the role of indentation in the Index view for Movies described in Screencast [2.8.1](#)?

 When one HTML element encloses other elements, indentation tells Haml the structure of the nesting so that it can generate closing tags such as `</tr>` in the proper places.

Self-Check 2.8.2. In the Index view for Movies, why does the Haml markup in line 11 begin with `-`, while the markup in lines 13–16 begins with `=`?

 In line 10 we just need the code to execute, to start the for-loop. In lines 13–16 we want to substitute the result of executing the code into the view.

Fallacy: Rails doesn't scale (or Django, or PHP, or other frameworks).

With the shared-nothing 3-tier architecture depicted in Figure [2.7](#), 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 Web server and app 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 master-slave replication 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.

An API that isn’t comprehensible isn’t usable. *James Gosling*

To understand the architecture of a software system is to understand its organizing principles. We did this by identifying patterns at many different levels: client-server, three-tier architecture, model-view-controller, Active Record, REST.

Patterns are a powerful way to manage complexity in large software systems. 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](#)), which described what are now called the 23 GOF Design Patterns focusing on class-level structures and behaviors. 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.

Indeed, 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 3-tier architecture 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 2.9. 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 Chapters 3 and 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.

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.

In fact, Rails itself was originally extracted from a standalone app written by the consulting group 37signals.

Lastly, it is worth remembering that a key factor in the Web’s success has been the adoption of well-defined protocols and formats whose design allows 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. Although dealing with multiple protocol and language versions puts an additional burden on browsers, it has led to a remarkable result: A Web page created in 2011, using a markup language based on 1960s technology, can be retrieved using network protocols developed in 1969 and displayed by a browser created in 1992. Separating the things that change from those that stay the same is part of the path to creating long-lived software.

Tim Berners-Lee, a computer scientist at [CERN](#), led the development of HTTP and HTML in 1990. Both are now stewarded by the nonprofit vendor-neutral [World Wide Web Consortium \(W3C\)](#).

2.11 To Learn More

- [W3Schools](#) is a free (advertising-supported) site with tutorials on almost all Web-related technologies.
- [Nicole Sullivan](#), a self-described “CSS ninja,” has a great blog with indispensable CSS/HTML advice for more sophisticated sites.
- [The World Wide Web Consortium \(W3C\)](#) stewards the official documents describing the Web’s open standards, including HTTP, HTML, and CSS.
- [The XML/XHTML Validator](#) is one of many you can use to ensure the pages delivered by your SaaS app are standards-compliant.
- [The Object-Oriented Design web site](#) has numerous useful resources for developers using OO languages, including a nice catalog of the GoF design patterns with graphical descriptions of each pattern, some of which we will

discuss in detail in Chapter 10.

M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. ISBN 0321127420. URL <http://martinfowler.com/eaaCatalog/>.

E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN 0201633612.

Project 2.1. If the DNS service stopped working, would you still be able to surf the Web? Explain why or why not.

Project 2.2. Suppose HTTP cookies didn't exist. Could you devise another way to track a user across page views? (HINT: it involves modifying the URI and was a widely-used method before cookies were invented.)

Project 2.3. Find a Web page for which the [W3C's online XHTML validator](#) finds at least one error. Sadly, this should be easy. Read through the validation error messages and try to understand what each one means.

Project 2.4. What port numbers are implied by each of the following URIs and why:

`https://paypal.com`

`http://mysite.com:8000/index`

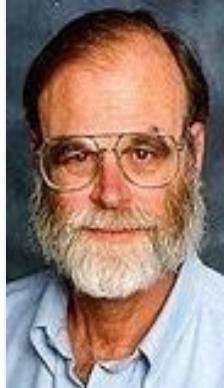
`ssh://root@cs.berkeley.edu/tmp/file` (HINT: recall that the IANA establishes default port numbers for various network services.)

Project 2.5.

As described on [Google's Search API documentation](#), you can do a Google search for a term by constructing a URI that includes the search query as a parameter named `q`, for example, `http://www.google.com/search?q=saas` to search for the term "saas". However, as Figure 2.3 showed, some characters are not allowed in URIs because they are "special," including spaces, '?', and '&'. Given this restriction, construct a legal URI that search Google for the terms "M&M" and "100%?".

`String#ord` returns the string's first `codepoint` (numeric value corresponding to a character in a character set). If the string is encoded in `ASCII`, `ord` returns the first character's ASCII code. So `"%" . ord` shows the ASCII code for %, and `"%" . ord.to_s(16)` shows its hexadecimal equivalent.

Project 2.6. Why do Rails routes map to controller actions but not model actions or views?



Jim Gray (1944–Lost at sea 2007) was a friendly giant in computer science. He was the first PhD in Computer Science from UC Berkeley, and he mentored hundreds of PhD students and faculty around the world. He received the 1998 Turing Award for contributions to database and transaction processing research and technical leadership in system implementation.

Well, the <omitted> paper is in good company (and for the same reason).

The B-tree paper was rejected at first.

The Transaction paper was rejected at first.

The data cube paper was rejected at first.

The five-minute rule paper was rejected at first.

But linear extensions of previous work get accepted.

So, resubmit! PLEASE!!! *Jim Gray, Email to Jim Larus about a rejected paper, 2000*

3.1 [Overview and Three Pillars of Ruby](#)

3.2 [Everything is an Object](#)

3.3 [Every Operation is a Method Call](#)

3.4 [Classes, Methods, and Inheritance](#)

3.5 [All Programming is Metaprogramming](#)

3.6 [Blocks: Iterators, Functional Idioms, and Closures](#)

3.7 [Mix-ins and Duck Typing](#)

3.8 [Make Your Own Iterators Using Yield](#)

3.9 [Fallacies and Pitfalls](#)

3.10 [Concluding Remarks: Idiomatic Language Use](#)

3.11 [To Learn More](#)

3.12 [Suggested Projects](#)

This quick introduction will get you up to speed on idiomatic Ruby, a highly productive scripting language. We focus on the unique productivity-enhancing features of Ruby that may be unfamiliar to Java programmers, and we omit many details that are well covered by existing materials. As with all languages, becoming truly comfortable with Ruby's powerful features will require going beyond the material in this introduction to the materials listed in Section 3.11.



Programming can only be learned by doing, so we've placed this icon in the margin in places where we strongly encourage you to try the examples yourself. Since Ruby is interpreted, there's no compile step—you get instant gratification when trying the examples, and exploration and experimentation are easy. Each example has a link to [Pastebin](#), where you can copy the code for that example with a single click and paste it into a Ruby interpreter or editor window. (If you're reading the ebook, these links are live.) We also encourage you to check the official documentation for much more detail on many topics we introduce in Section 3.11. Ruby is a minimalist language: while its libraries are rich, there are few mechanisms *in the language itself*. Three principles underlying these mechanisms will help you read and understand idiomatic code:

Everything is an object. In Java, some primitive types like integers must be “boxed” to get them to behave like objects.

Every operation is a method call on some object and returns a value. In Java, operator overloading is different from method overriding, and it's possible to have `void` functions that return no value.

All programming is metaprogramming: classes and methods can be added or changed at any time, even while a program is running. In Java, all classes must be declared at compile time, and base classes can't be modified by your app even then.

Variables	<code>local_variable, @@class_variable, @instance_variable</code>
Constants	<code>ClassName, CONSTANT, \$GLOBAL, \$global</code>
Booleans	<code>false, nil</code> are false; <code>true</code> and <i>everything else</i> (zero, empty string, etc.) is true.
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
Expressions in <i>double-quoted</i> strings	<code>@foo = 3 ; "Answer is #{@foo}"; %Q{Answer is #{@foo+1}}</code>
Regular expression matching	<code>"hello" =~ /lo/ or "hello".match(Regexp.new 'lo')</code>
Arrays	<code>a = [1, :two, 'three'] ; a[1] == :two</code>
Hashes	<code>h = {:@a =>1, 'b' =>"two"} ; h['b'] == "two" ; h.has_key?(:a) == true</code>
Instance method	<code>def method(arg, arg)...end</code> <i>(use <code>*args</code> for variable number of arguments)</i>

Class (static) method	<code>def ClassName.method(arg, arg)...end</code>	
Special method names	<code>def self.method(arg, arg)...end</code>	
<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 3.6)	Exceptions
<code>if cond (or unless cond)</code>	<code>while cond (or until cond)</code>	<code>begin</code>
<i>statements</i>	<i>statements</i>	<i>statements</i>
<code>[elsif cond</code>	<code>end</code>	<code>rescue AnError => e</code>
<i>statements]</i>	<code>1.upto(10) do </code>	<i>e is an exception of class</i>
	<code>i ...end</code>	<code>AnError;</code>
<code>[else</code>	<code>10.times do...end</code>	<i>multiple rescue clauses OK</i>
<i>statements]</i>	<code>collection.each do </code>	<code>[ensure</code>
	<code>elt ...end</code>	<i>this code is always executed]</i>
<code>end</code>		<code>end</code>

Figure 3.1: Basic Ruby elements and control structures, with optional items in [square brackets].

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

Figure 3.2: Review of Ruby's regular expressions.

Each of these three principles will be covered in its own section. #1 and #2 are straightforward. #3 gives Ruby much of its productivity-enhancing power, but must be qualified with the admonition that with great power comes great responsibility. Using Ruby's metaprogramming features tastefully will make your code elegant and

DRY, but abusing them will make your code brittle and impenetrable. Ruby's basic syntax should be unsurprising if you're familiar with other modern scripting languages. Figure 3.1 shows the syntax of basic Ruby elements. Statements are separated by newlines (most commonly) or semicolons (rarely). Indentation is insignificant. While Ruby is concise enough that a single line of code rarely exceeds

one screen line, breaking up a single statement with a newline is allowed if it doesn't cause a parsing ambiguity. An editor with good syntax highlighting can be very helpful if you're not sure whether your line break is legal.

A [symbol](#), such as `:octocat`, is an immutable string whose value is itself; it is typically used in Ruby for enumerations, like an `enum` type in C or Java, though it has other purposes as well. However, a symbol is not the same as a string—it is its own primitive type, and string operations cannot be performed on it, though it can easily be converted to a string by calling `to_s`. For example, `:octocat.to_s` gives `"octocat"`, and `"octocat".to_sym` gives `:octocat`.

[Regular expressions](#) or [regexps](#) (often `regex` and `regexes` in order to be pronounceable) are part of every programmer's toolbox. A regular expression allows matching a string against a pattern containing possible "wildcards." Ruby's regular expression support resembles that of other modern programming languages: regexes appear between slashes and may be followed by one or more letters modifying their behavior, for example, `/regex/i` to indicate that the regex should ignore case when matching. As Figure 3.2 shows, special constructs embedded in the regex can match multiple types of characters and can specify the number of times a match must occur and whether the match must be "anchored" to the beginning or end of the string. For example, here is a regex that matches a time of day, such as "8:25pm", on a line by itself:

<http://pastebin.com/85Xs5A9V>

```
1   time_regex = /^\\d\\d?:\\d\\d\\s*[ap]m$/i
```

This regexp matches a digit at the beginning of a string (`^\\d`), optionally followed by another digit (`\\d?`), followed by a colon, exactly two digits, zero or more whitespace characters (`\\s*`), either `a` or `p`, then `m` at the end of the string (`m$`) and ignoring case (the `i` after the closing slash). Another way to match one or two digits would be `[0-9][0-9]?` and another way to match *exactly* two digits would be `[0-9][0-9]`.

Ruby allows the use of parentheses in regexes to *capture* the matched string or substrings. For example, here is the same regexp with three capture groups:

<http://pastebin.com/ib56Dv0y>

```
1   x = "8:25 PM"
2   x =~ /(\d\d?)(\d\d)\s*([ap])m$/i
```

The second line attempts to match the string `x` against the regex. If the match succeeds, the `=~` operator will return the position in the string (with 0 being the first character) at which the match succeeded, the global variable `$1` will have the value `"8"`, `$2` will be `"25"`, and `$3` will be `"P"`. The last-match variables are reset the next time you do another regex match. If the match fails, `=~` will return `nil`. Note that `nil` and `false` are not actually equal to each other, but both evaluate to “false” when used in a conditional expression (in fact, they are the *only* two values in Ruby that do so). Idiomatically, methods that are truly Boolean (that is, the only possible return values are “true” or “false”) return `false`, whereas methods that return an object when successful return `nil` when they fail.

Lastly, note that `=~` works on both strings and `Regexp` objects, so both of the following are legal and equivalent, and you should choose whichever is easiest to understand in the context of your code.

<http://pastebin.com/DMuMijw0>

```
1 "Catch 22" =~ /\w+\s+\d+/
2 /\w+\s+\d+/ =~ "Catch 22"
```

Summary

- A distinguishing 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. Symbols aren’t the same as strings, but they can easily be converted back and forth with the methods `to_s` and `to_sym`.
- Ruby statements are separated by newlines, or less commonly, by semicolons.
- Ruby’s regular expression facilities are comparable to those of other modern languages, including support for capture groups using parentheses and for match modifiers such as a trailing `i` for “ignore case when matching.”

Self-Check 3.1.1. 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 3.1.2. What is captured by `$1` when the string `25 to 1` is matched against each of the following regexps:

(a) `/(\d+)/$`
(b) `/^ \d+([^\d]+) /`

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

Self-Check 3.1.3. When is it correct to write

`Fixnum num=3`

to initialize the variable `num`: (a) on its first use; (b) on any use, as long as it’s the same class `Fixnum` each time; (c) never

-  Never; variable declarations aren’t used in Ruby.

Ruby supports the usual basic types—fixed-point integers (class `Fixnum`), floating-point numbers (class `Float`), strings (class `String`), linear arrays (class `Array`), and associative arrays or hashmaps (class `Hash`). But in contrast to Java, Ruby is

dynamically typed: the type of a variable is generally not inferable until runtime.

That is, while objects have types, the variables that reference them do not. So `s = 5` can follow `s = "foo"` in the same block of code. Because variables do not have types, an array or hash can consist of elements of all different types, as Figure 3.1 suggests. We speak only of “an array” rather than “an array of Ints” and of “a hash” rather than “a hash with keys of type Foo and values of type Bar.”

Ruby’s object model descends from Smalltalk, whose design was inspired by ideas in Simula. Everything in Ruby, even a plain integer, is an object that is an instance of some class. All operations, without exception, are performed by calling a method on an object, and every such call (indeed, every Ruby statement) returns a value. The notation `obj.meth()` calls method `meth` on the object `obj`, which is said to be the ***receiver*** and is hopefully able to *respond to* `meth`. As we will see shortly, parentheses around method arguments are often optional. For example:

<http://pastebin.com/CqArx48i>

```
1      5.class      # => Fixnum
```

(We strongly recommend you start up a Ruby interpreter by typing `irb` in a Terminal window in your VM so you can try these examples as you go.) The above call *sends* the method call `class` to the object `5`. The `class` method happens to return the class that an object belongs to, in this case `Fixnum`. (We will use the notation `# =>` in code examples to indicate what the interpreter should print as the result of evaluating a given expression.)

Even a class in Ruby is itself an object—it’s an instance of `Class`, which is a class whose instances are classes (a ***metaclass***).

Every object is an instance of some class. Classes can inherit from superclasses as they do in Java, and all classes ultimately inherit from `BasicObject`, sometimes called the ***root class***. Ruby does not support multiple inheritance, so every class has exactly one superclass, except for `BasicObject`, 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. If no class along the way, including the root class, is able to handle the method, an *undefined method* exception is raised.

Try `5.class.superclass` to find out what `Fixnum`'s superclass is; this illustrates **method chaining**, a very common Ruby idiom. Method chaining associates to the left, so this example could be written `(5.class).superclass`, meaning: “Call the `class` method on the receiver `5`, and whatever the result of that is, call the `superclass` method on that receiver.”

Object-orientation (OO) and class inheritance are *distinct properties*. Because popular languages such as Java combine both, many people conflate them. Ruby also happens to have both, but the two features do not necessarily interact in all the same ways they would in Java. In particular, compared to Java, reuse through inheritance is much less important, but the implications of object-orientation are much more important. For example, Ruby supports 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. Given that an object responds to a method, how can you tell if the method is defined in the object’s class or an ancestor class? `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`. Determining which class’s methods will handle a method call is called **looking up a method** on a receiver, and is analogous to virtual method dispatch in Java.

Summary

- 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.
 - Ruby has comprehensive [reflection](#), allowing you to ask objects about themselves.
-

ELABORATION: Looking up a method

Previously we said that if method lookup fails in the receiver’s class, the call is passed up to the ancestor (superclass). The truth is a bit more subtle: mix-ins, which we’ll describe shortly, can handle a method call before punting up to the superclass.

Self-Check 3.2.1. 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.

To cement the concept that every operation is a method call, note that even basic math operations such as `+`, `*`, `==` (equality comparison) are syntactic sugar for method calls: the operators are actually method calls on their receivers. The same is true for array dereferences such as `x[0]` and array assignment such as `x[0] = "foo"`.

The table in Figure 3.3 shows the de-sugared versions of these expressions and of the regex syntax introduced in Section 3.1, as well as showing how Ruby's core method `send` can be used to send any method call to any object. Many Ruby methods including `send` accept either a symbol or a string argument, so the first example in the table could also be written `10.send('modulo', 3)`.

Sugared	De-sugared	Explicit <code>send</code>
<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> (<code>regex capture</code>)	<code>Regexp.last_match(n)</code>	<code>Regexp.send(:last_match, n)</code>

Figure 3.3: 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 a symbol (more idiomatic) for the method name.

A critical implication of “every operation is a method call” is that concepts such as type casting rarely apply in Ruby. In Java, if we write `f+i` where `f` is a float and `i` is an integer, the type casting rules state that `i` will be converted internally to a float so it can be added to `f`. If we wrote `i+s` where `s` is a `String`, a compile-time error would result.

In contrast, in Ruby `+` is just like any other method that can be defined differently by

each class, so its behavior depends entirely on the receiver's implementation of the method. Since `f+i` is syntactic sugar for `f.+({i})`, it's entirely up to the `+` method (presumably defined in `f`'s class or one of its ancestor classes) to decide how to handle different types of values for `i`. Thus, both `3+2` and `"foo"+"bar"` are legal Ruby expressions, evaluating to `5` and `"foobar"` respectively, but the first one is calling `+` as defined in `Numeric` (the ancestor class of `Fixnum`) whereas the second is calling `+` as defined in `String`. As above, you can verify that

`"foobar".method(:+)` and `5.method(:+)` refer to distinct methods. Although this might resemble operator overloading in other languages, it's more general: since only the method's name matters for dispatching, we'll see in Section 3.7 how this feature enables a powerful reuse mechanism called a mix-in.

In Ruby the notation `ClassName#method` is used to indicate the instance method `method` in `ClassName`, whereas `ClassName.method` indicates the class (static) method `method` in `ClassName`. Using this notation, we can say that the expression `3+2` results in calling `Fixnum#+` on the receiver `3`, whereas the expression `"foo"+"bar"` results in calling `String#+` on the receiver `"foo"`.

Similarly, in Java it's common to see explicit casts of a variable to a subclass, such as `Foo x = (Foo)y` where `y` is an instance of a subclass of `Foo`. In Ruby this is meaningless because variables don't have types, and it doesn't matter whether the responding method is in the receiver's class or one of its ancestors.

A method is defined with `def method_name(arg1, arg2)` and ends with `end`; all statements in between are the method definition. Every expression in Ruby has a value—for example, the value of an assignment is its right-hand side, so the value of `x=5` is `5`—and if a method doesn't include an explicit `return(blah)`, the value of the last expression in the method is returned. Hence the following trivial method returns `5`:

<http://pastebin.com/0UHrZHDJ>

```
1
2   def trivial_method      # no arguments; can also use trivial_met
3   hod()
4       x = 5
5   end
```

The variable `x` in the example is a local variable; its scope is limited to the block in which it's defined, in this case the method, and is undefined outside that method. In other words, Ruby uses [lexical scoping](#) for local variables. When we talk about classes in Ruby, we'll see how class and instance variables are alternatives to local variables.

An important Ruby idiom is [poetry mode](#): the ability to omit parentheses and curly

braces when the parsing is unambiguous. Most commonly, Ruby programmers may omit parentheses around arguments to a method call, and omit curly braces when the *last* argument to a method call is a hash. Hence the following two method calls are equivalent, given a method `link_to` (which we'll meet in Section 4.4) that takes one string argument and one hash argument:

<http://pastebin.com/C6Z9a3kW>

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

Poetry mode is exceedingly common among experienced Rubyists, is used pervasively in Rails, and provides a welcome elimination of clutter once you get used to it.

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.
- Ruby uses lexical scoping for local variables: a variable is defined in the scope in which it's first assigned and in all scopes enclosed inside that one, but reusing the same local variable name in an inner scope temporarily "shadows" the name from the enclosing scope.
- Poetry mode reduces clutter by allowing you to omit parentheses around method arguments and curly braces surrounding a hash, as long as the resulting code is syntactically unambiguous.

ELABORATION: Number of arguments

Although parentheses around method arguments are optional both in the method's definition and when calling the method, the *number* of arguments does matter, and an exception is raised if a method is called with the wrong number of arguments. The following code snippet shows two idioms you can use when you need more flexibility. The first is to make the last argument a hash and give it a default value of `{}` (the empty hash). The second is to use a splat (*), which collects any extra arguments into an array. As with so many Rubyisms, the right choice is whichever results in the most readable code.

<http://pastebin.com/7m7FtVN7>

```

1 # using 'keyword style' arguments
2 def mymethod(required_arg, args={})
3   do_fancy_stuff if args[:fancy]
4 end
5
6 mymethod "foo", :fancy => true # => args={:fancy => true}
7 mymethod "foo"                 # => args={}
8
9 # using * (splat) arguments
10 def mymethod(required_arg, *args)
11   # args is an array of extra args, maybe empty
12 end
13
14 mymethod "foo", "bar", :fancy => true # => args=["bar",
15 { :fancy=>true}]
16 mymethod "foo"                      # => args=[]

```



Self-Check 3.3.1. What is the explicit-`send` equivalent of each of the following expressions: `a<b`, `a==b`, `x[0]`, `x[0]='foo'`.

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

Self-Check 3.3.2. Suppose method `foo` takes two hash arguments. Explain why we can't use poetry mode to write

`foo :a => 1, :b => 2`

b Without curly braces, there's no way to tell whether this call is trying to pass a hash with two keys or two hashes of one key each. Therefore poetry mode can only be used when there's a single hash argument and it's the last argument.

The excerpt of a class definition for a `Movie` in Figure 3.4 illustrates some basic concepts of defining a class in Ruby. Let's step through it line by line.

<http://pastebin.com/01GAXXc1>

```

1 class Movie
2   def initialize(title, year)
3     @title = title
4     @year = year

```

```

5   end
6   def title
7     @title
8   end
9   def title=(new_title)
10    @title = new_title
11 end
12 def year ; @year ; end
13 def year=(new_year) ; @year = new_year ; end
14 # How to display movie info
15 @@include_year = false
16 def Movie.include_year=(new_value)
17   @@include_year = new_value
18 end
19 def full_title
20   if @@include_year
21     "#{self.title} (#{self.year})"
22   else
23     self.title
24   end
25 end
26 end
27
28 # Example use of the Movie class
29
30 beautiful = Movie.new('Life is Beautiful', '1997')
31
32 # What's the movie's name?
33 puts "I'm seeing #{beautiful.full_title}"
34
35 # And with the year
36 Movie.include_year = true
37 puts "I'm seeing #{beautiful.full_title}"
38
39 # Change the title
40 beautiful.title = 'La vita e bella'
41 puts "Ecco, ora si chiama '#{beautiful.title}'"

```

Figure 3.4: A simple class definition in Ruby. Lines 12 and 13 remind us that it's idiomatic to combine short statements on a single line using semicolons; most Rubyists take advantage of Ruby's conciseness

to introduce spaces around the semicolons for readability.

Line 1 opens the `Movie` class. As we'll see, unlike in Java, `class Movie` is *not* a declaration but actually a method call that creates an object representing a new class and assigns this object to the constant `Movie`. The subsequent method definitions will occur in the context of this newly-created class.

Line 2 defines the default constructor for the class (the one called when you say `Movie.new`), which *must* be named `initialize`. The inconsistency of naming a method `initialize` but calling it as `new` is an unfortunate idiosyncrasy you'll just have to get used to. (As in Java, you can define additional constructors with other names as well.) This constructor expects two arguments, and in **lines 3–4**, it sets the instance variables of the new `Movie` object to those values. The instance variables, such as `@title`, are associated with each instance of an object. The local variables `title` and `year` passed in as arguments are out of scope (undefined) outside the constructor, so if we care about those values we must capture them in instance variables.

Lines 6–8 define a getter method or accessor method for the `@title` instance variable. You might wonder why, if `beautiful` were an instance of `Movie`, we couldn't just write `beautiful.@title`. It's because in Ruby, `a.b` always means "Call method `b` on receiver `a`", and `@title` is not the name of any method in the `Movie` class. In fact, it is not a legal name for a method at all, since only instance variable names and class variable names may begin with `@`. In this case, the `title` getter is an instance method of the `Movie` class. That means that any object that is an instance of `Movie` (or of one of `Movie`'s subclasses, if there were any) could respond to this method.

Lines 9–11 define the instance method `title=`, which is distinct from the `title` instance method. Methods whose names end in `=` are setter or mutator methods, and just as with the getter, we need this method because we cannot write `beautiful.@title = 'La vita e bella'`. However, as line 40 shows, we *can* write `beautiful.title = 'La vita e bella'`. Beware! If you're used to Java or Python, it's very easy to think of this syntax as *assignment to an attribute*, but it is really just a method call like any other, and in fact could be written as `beautiful.send(:title=, 'La vita e bella')`. And since it is a method call, it has a return value: in the absence of an explicit `return` statement, the value returned by a method is just the value of the last expression evaluated in that method. Since in this case the last expression in the method is the assignment `@title=new_title` and the value of any assignment is its right-hand side, the method happens to return the value of `new_title` that was passed to it.

Unlike Java, which allows attributes as well as getters and setters, Ruby's data hiding or encapsulation is total: the *only* access to an instance variables or class variables from outside the class is via method calls. This restriction is one reason that Ruby is considered a more "pure" OO language than Java. But since poetry

mode allows us to omit parentheses and write `movie.title` instead of `movie.title()`, conciseness need not be sacrificed to achieve this stronger encapsulation.



Lines 12–13 define the getter and setter for `year`, showing that you can use semicolons as well as newlines to separate Ruby statements if you think it looks less cluttered. As we'll soon see, though, Ruby provides a much more concise way to define getters and setters using metaprogramming.

Line 14 is a comment, which in Ruby begins with `#` and extends to the end of the line.

Line 15 defines a [**class variable**](#), or what Java calls a **static variable**, that defines whether a movie's year of release is included when its name is printed. Analogously to the setter for `title`, we need one for `include_year=` (**lines 16–18**), but the presence of `Movie` in the name of the method (`Movie.include_year=`) tells us it's a class method. Notice we haven't defined a getter for the class variable; that means the value of this class variable cannot be inspected at all from outside the class.

Lines 19–25 define the instance method `full_title`, which uses the value of `@include_year` to decide how to display a movie's full title. Line 21 shows that the syntax `#{}</code>` can be used to interpolate (substitute) the value of an expression into a double-quoted string, as with `#{self.title}` and `#{self.year}`. More precisely, `#{}` evaluates the expression enclosed in the braces and calls `to_s` on the result, asking it to convert itself into a string that can be inserted into the enclosing string. The class `Object` (which is the ancestor of all classes except `BasicObject`) defines a default `to_s`, but most classes override it to produce a prettier representation of themselves.

Java

`class MyString extends String`

`class MyCollection extends Array`
`implements Enumerable`

Static variable: `static int anInt = 3`

Instance variable: `this.foo = 1`

Static method:

`public static int foo(...)`

Instance method: `public int foo(...)`

Ruby

`class MyString < String`

`class MyCollection < Array`
`include Enumerable`

Class variable: `@@an_int = 3`

Instance variable: `@foo = 1`

Class method:

`def self.foo ... end`

Instance method: `def foo ... end`

Figure 3.5: A summary of some features that translate directly between Ruby and Java.

Summary: Figure 3.5 compares basic OO constructs in Ruby and Java:

- `class Foo` opens a class (new or existing) in order to add or change methods in it. Unlike Java, it is not a declaration but actual code executed immediately, creating a new `Class` object and assigning it to the constant `Foo`.
 - `@x` specifies an instance variable and `@@x` specifies a class (static) variable. The namespaces are distinct, so `@x` and `@@x` are different variables.
 - A class's instance variables and class variables can be accessed only from within the class. Any access from the "outside world" requires a method call to either a getter or a setter.
 - A class method in class `Foo` can be defined using either `def Foo.some_method` or `def self.some_method`.
-

ELABORATION: Using `self` to define a class method.

As we'll soon see, the class method definition `def Movie.include_year` can actually appear *anywhere*, even outside the `Movie` class definition, since Ruby allows adding and modifying methods in classes after they've been defined. However, another way to define the class method `include_year` inside the class definition would be `def self.include_year=(...)`. This is because, as we mentioned above, `class Movie` in line 1 is not a declaration but actual code that is executed when this file is loaded, and inside the code block enclosed by `class Movie...end` (lines 2–25), the value of `self` is the new class object created by the `class` keyword. (In fact, `Movie` itself is just a plain old Ruby constant that refers to this class object, as you can verify by doing `c = Movie` and then `c.new('Inception', 2010)`.)

ELABORATION: Why `self.title` in `Movie#full_title`?

In lines 19–25, why do we call `self.title` and `self.year` rather than just referring directly to `@title` and `@year`, which would be perfectly legal inside an instance method? The reason is that in the future, we might want to change the way the getters work. For example, when we introduce Rails and ActiveRecord in Section 4.3, we'll see that getters for basic Rails models work by retrieving information from the database, rather than tracking the information using instance variables. Encapsulating instance and class variables using getters and setters hides the implementation of those attributes from the code that uses them, and there's no advantage to be gained by violating that encapsulation inside an instance method, even though it's legal to do so.

Self-Check 3.4.1. 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 3.4.2. Suppose we delete line 12 from Figure 3.4. What would be the result of executing `Movie.new('Inception', 2011).year`?

 Ruby would complain that the `year` method is undefined.

Since defining simple getters and setters for instance variables is so common, we can make the example more Ruby-like by replacing lines 6–11 with the single line

`attr_accessor :title` and lines 12–13 with `attr_accessor :year`. `attr_accessor` is not part of the Ruby language—it's a regular method call that defines the getters and setters on the fly. That is, `attr_accessor :foo` defines instance methods `foo` and `foo=` that get and set the value of instance variable `@foo`. The related method `attr_reader` defines only a getter but no setter, and vice versa for `attr_writer`.



This is an example of [metaprogramming](#)—creating code at runtime that defines new methods. In fact, in a sense *all* Ruby programming is metaprogramming, since even a class definition is not a declaration as it is in Java but actually code that is executed at runtime. Given that this is true, you might wonder whether you can *modify* a class at runtime. In fact you can, by adding or changing instance methods or class methods, even for Ruby's built-in classes. For example, Figure 3.6 shows a way to do time arithmetic that takes advantage of the `now` method in the `Time` class in the standard Ruby library, which returns the number of seconds since 1/1/1970.

<http://pastebin.com/RBejPpb>

```
1 # Note: Time.now returns current time as seconds since epoch
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
10 # => Mon Nov 07 10:18:10 -0800 2011
11 5.minutes.ago
12 # => Mon Nov 07 10:13:15 -0800 2011
13 5.minutes - 4.minutes
14 # => 60
15 3.hours.from_now
16 # => Mon Nov 07 13:18:15 -0800 2011
```

Figure 3.6: Doing simple time arithmetic by reopening the `Fixnum` class. Unix was invented in 1970, so its designers chose to represent time as the number of seconds since midnight (GMT) 1/1/1970, sometimes called the beginning of the *epoch*. For convenience, a Ruby `Time` object responds to

arithmetic operator methods by operating on this representation if possible, though internally Ruby can represent any time past or future.

In this example, we *reopened* the `Fixnum` class, a core class that we met earlier, and added six new instance methods to it. Since each of the new methods also returns a fixnum, they can be nicely “chained” to write expressions like `5.minutes.ago`. In fact, Rails includes a more complete version of this feature that does comprehensive time calculations.



Of course, we cannot write `1.minute.ago` since we only defined a method called `minutes`, not `minute`. We could define additional methods with singular names that duplicate the functionality of the methods we already have, but that's not very DRY. Instead, we can take advantage of Ruby's heavy-duty metaprogramming construct `method_missing`. If a method call cannot be found in the receiver's class or any of its ancestor classes, Ruby will then try to call `method_missing` on the receiver, passing it the name and arguments of the nonexistent method. The default implementation of `method_missing` just punts up to the superclass's implementation, but we can override it to implement “singular” versions of the time-calculation methods above:

<http://pastebin.com/mTkpBR1t>

```
1  class Fixnum
2    def method_missing(method_id, *args)
3      name = method_id.to_s
4      if name =~ /^(second|minute|hour)$/
5          self.send(name + 's')
6      else
7          super # pass the buck to superclass
8      end
9    end
10   end
```

We convert the method ID (which is passed as a symbol) into a string, and use a regular expression to see if the string matches any of the words `hour`, `minute`, `second`. If so, we pluralize the name, and send the pluralized method name to `self`, that is, to the object that received the original call. If it doesn't match, what should we do? You might think we should signal an error, but because Ruby has inheritance, we must allow for the possibility that one of our ancestor classes might be able to

handle the method call. Calling `super` with no arguments passes the original method call and its original arguments intact up the inheritance chain.



Try augmenting this example with a `days` method so that you can write `2.days.ago` and `1.day.ago`. Later in the chapter we'll see `method_missing` in action for constructing XML documents and for enhancing the power of the `find` method in the ActiveRecord part of the Rails framework.

Summary

- `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.
- When neither a receiver nor any of its ancestor classes can handle a method call, `method_missing` is called on the receiver. `method_missing` can inspect the name of the nonexistent method and its arguments, and can either take action to handle the call or pass the call up to the ancestor, as the default implementation of `method_missing` does.

ELABORATION: Pitfalls of dynamic language features

If your `Bar` class has actually been using an instance variable `@fox` but you accidentally write `attr_accessor :foo` (instead of `attr_accessor :fox`), you will get an error when you write `mybar.fox`. Since Ruby doesn't require you to declare instance variables, `attr_accessor` cannot check whether the named instance variable exists. Therefore, as with all dynamic language features, we must employ care in using it, and cannot "lean on the compiler" as we would in Java. As we will see in Chapter 6, test-driven development (TDD) helps avoid such errors. Furthermore, to the extent that your app is part of a larger Service-Oriented Architecture ecosystem, you *always* have to worry about runtime errors in other services that your app depends on, as we'll see in Chapters 7 and 12.

ELABORATION: Variable length argument lists

A call such as `1.minute` doesn't have any arguments—the only thing that matters is the receiver, `1`. So when the call is redispersed in line 5 of `method_missing`, we don't need to pass any of the arguments that were collected in `*args`. The asterisk is how Ruby deals with argument lists that have zero or more arguments: `*args` will be an array of any arguments passed to the original method, and if no arguments were passed it will be an empty array. It would be correct in any case for line 5 to read `self.send(name+'s', *args)` if we weren't sure what the length of the argument list was.

Self-Check 3.5.1. In the `method_missing` example above, why are `$` and `^` necessary in the regular expression match in line 4? (Hint: consider what happens if

you omit one of them and call `5.milliseconds` or `5.secondary`)

Without `^` to constrain the match to the beginning of the string, a call like `5.millisecond` would match, which will cause an error when `method_missing` tries to redispach the call as `5.milliseconds`. Without `$` to constrain the match to the end of the string, a call like `5.secondary` would match, which will cause an error when `method_missing` tries to redispach the call as `5.secondaries`.

Self-Check 3.5.2. Why should `method_missing` always call `super` if it can't handle the missing method call itself?

It's possible that one of your ancestor classes intends to handle the call, but you must explicitly "pass the method call up the chain" with `super` to give the ancestor classes a chance to do so.

Self-Check 3.5.3. In Figure 3.6, 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. 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. Here is a simple example assuming `movies` is an array of `Movie` objects as we defined in the previous examples:

<http://pastebin.com/K3QRKmzx>

```
1 movies.each do |m|
2   puts "#{m.title} is rated #{m.rating}"
3 end
```

The method `each` is an **iterator** available in all Ruby classes that are collection-like. `each` takes one argument—a block—and passes each element of the collection to the block in turn. As you can see, a block is bracketed by `do` and `end`; if the block takes arguments, the argument list is enclosed in `|` pipe symbols `|` after the `do`. The block in this example takes one argument: each time through the block, `m` is set to the next element of `movies`.

Unlike named methods, a block can also access any variable accessible to the scope in which the block appears. For example:

<http://pastebin.com/s1ULuMtX>

```
1 separator = '=>'
2 movies.each do |m|
3   puts "#{m.title} #{separator} #{m.rating}"
4 end
```

In the above code, the value of `separator` is visible inside the block, even though the variable was created and assigned *outside* the block. In contrast, the following would *not* work, because `separator` is not visible within `print_movies`, and therefore not visible to the `each` block:

<http://pastebin.com/TFb7QJtD>

```
1   separator = '=>'
2   print_movies(movies)
3   #
4   def print_movies(movie_list)
5     movie_list.each do |m|
6
7       puts "#{m.title} #{separator} #{m.rating}" # === FAILS!!
=====
7       end
8     end
```

In programming-language parlance, 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 later in this chapter) and model validations and controller filters (Chapter 7), 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.

The fact that blocks are closures should help explain the following apparent anomaly. If the *first* reference to a local variable is inside a block, that variable is “captured” by the block’s scope and is *undefined* after the block exits. So, for example, the following will *not* work, assuming line 2 is the *first* reference to `separator` within this scope:

<http://pastebin.com/k8ENWbi8>

```
1   movies.each do |m|
2     separator = '=>' # first assignment is inside a block!
3     puts "#{m.title} #{separator} #{m.rating}" # OK
```

```
4     end  
5     puts "Separator is #{separator}"      # === FAILS!! ===
```

In a lexically-scoped language such as Ruby, variables are visible to the scope within which they're created and to all scopes enclosed by that scope. Because in the above snippet `separator` is *created* within the block's scope, its visibility is limited to that scope.

In summary, `each` is just an instance method on a collection that takes a single argument (a block) and provides elements to that block one at a time. A related use of blocks is operations on collections, a common idiom Ruby borrows from [functional programming](#). For example, to double every element in a collection, we could write:

<http://pastebin.com/6iNYDax0>

```
1   new_collection = collection.map do |elt|  
2     2 * elt  
3   end
```

If the parsing is unambiguous, it is idiomatic to use curly braces to delineate a short (one-line) block rather than `do...end`:

<http://pastebin.com/WFHRA8Ap>

```
1   new_collection = collection.map { |elt| 2 * elt }
```

So, no for-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.

Ruby has a wide variety of such collection operators; Figure 3.7 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. For example, to return a list of all the words in some file (that is, tokens consisting entirely of word characters and separated by non-word characters) that begin with a vowel, are sorted, and without duplicates:

<http://pastebin.com/xEGKGTaR>

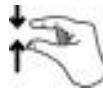
```

1  # downcase and split are defined in String class
2  words = IO.read("file").
3    split(/\W+).
4    select { |s| s =~ /^[aeiou]/i }.
5    map { |s| s.downcase }.
6    uniq.
7    sort

```

(Recall that Ruby allows breaking a single statement across lines for readability as long as it's not ambiguous where the statement ends. The periods at the end of each line make it clear that the statement continues, since a period must be followed by a method call.) In general, if you find yourself writing explicit loops in Ruby, you should reexamine your code to see if these collection idioms wouldn't

make it more concise and readable.



Method	#Args	Returns a <i>new collection containing...</i>
c.map	1	elements obtained by applying block to each element of c
c.select	1	Subset of c for which block evaluates to true
c.reject	1	Subset of c obtained by removing elements for which block evaluates to true
c.uniq		all elements of c with duplicates removed
c.reverse		elements of c in reverse order
c.compact		all non-nil elements of c
c.flatten		elements of c and any of its sub-arrays, recursively flattened to contain only non-array elements
c.partition	1	Two collections, the first containing elements of c for which the block evaluates to true, and the second containing those for which it evaluates to false
c.sort	2	Elements of c 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 *collection elements* to respond to `<=>`; see Section [3.7](#).

`c.sort`

If `sort` is called *without* a block, the elements are sorted according to how they respond to `<=>`.

`c.sort_by` 1

Applies the block to each element of `c` and sorts the result. For example, `movies.sort_by { |m| m.title }` sorts `Movie` objects according to how their titles respond to `<=>`.

`c.max, c.min`

Largest or smallest element in the collection

Figure 3.7: Some common Ruby methods on collections. For those that expect a block, we show 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. Some methods also have a *destructive* variant ending in `!`, for example `sort!`, that modify their argument in place (and also return the new value). Use destructive methods with extreme care.

Summary

- Ruby includes aspects of [functional programming](#) such as the ability to operate on entire collections with methods such as `map` and `sort`. It is highly idiomatic to use such methods to manipulate collections rather than iterating over them using for-loops.
- The `each` collection method returns one element of the collection at a time and passes it to a **block**. Blocks in Ruby can only occur as arguments to methods like `each` that expect a block.
- Blocks are closures: all variables visible to the block's code at the place where the block is defined will also be visible whenever the block executes.
- Most methods that appear to modify a collection, such as `reject`, actually return a new copy with the modifications made. Some have destructive versions whose name ends in `!`, as in `reject!`.

ELABORATION: Blocks and metaprogramming in XML Builder

An elegant example of combining blocks and metaprogramming is the [XML Builder](#) class. (As we mentioned briefly in Section [2.3](#), HTML is closely related to XML.) In the following example, the XML markup shown in lines 1–8 was generated by the Ruby code in lines 9–18. The method calls `name`, `phone`, `address`, and so on all use `method_missing` to turn each method call into an XML tag, and blocks are used to indicate tag nesting.

<http://pastebin.com/gVPp5XrK>

```
1  <person type="faculty">
2    <name>Barbara Liskov</name>
3    <contact>
4      <phone location="office">617-253-2008</phone>
5      <email>liskov@csail.mit.edu</email>
6    </contact>
7  </person>
8
9  # Code that generates the above markup:
10 require 'builder'
11 b = Builder::XmlMarkup.new(:indent => 2)
12 b.person :type => 'faculty' do
13   b.name "Barbara Liskov"
14   b.contact do
15     b.phone "617-253-2008", :location => 'office'
16     b.email "liskov@csail.mit.edu"
17   end
18 end
```

Self-Check 3.6.1. 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 [3.7](#), 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 `$`.



You may be surprised to learn that the collection methods summarized in Figure [3.7](#) (and several others not in the figure) aren't part of Ruby's `Array` class. In fact, they aren't even part of some superclass from which `Array` and other collection types inherit. Instead, they take advantage of an even more powerful

mechanism for reuse, called [***mix-ins***](#).

A mix-in is a collection of related behaviors that can be added to any class, although in some cases the class may have to fulfill a “contract” in order to use the mix-in. This may sound similar to an Interface in Java, but there are two differences. First, a mix-in is easier to reuse: the “contract,” if any, is specified in the mix-in’s documentation rather than being formally declared as a Java interface would be. Second, unlike a Java interface, which says nothing about *how* a class implements an interface, a mix-in is all about making it easy to reuse an implementation.

If you use the Emacs editor, you can think of Emacs minor modes (auto-fill, abbreviation support, and so on) as mix-ins that rely on contracts provided by the major mode and use Lisp’s dynamic typing to allow mixing them into any major mode.

A *module* is Ruby’s method for packaging together a group of methods as a mix-in. (Modules have other uses too, but mix-ins are the most important.) When a module is included into a class with `include ModuleName`, the instance methods, class methods, and variables in the module become available in the class.

The collection methods in Figure 3.7 are part of a module called **Enumerable** that is part of Ruby’s standard library; to mix **Enumerable** into your own class, just say `include Enumerable` inside the class definition.

Watch out! Because Ruby allows adding and defining methods at runtime, `include` cannot check whether the module’s contract is fulfilled by the class.

As its [documentation](#) states, **Enumerable** requires the class mixing it in to provide an `each` method, since **Enumerable**’s collection methods are implemented in terms of `each`. Unlike a Java interface, this simple contract is the *only* requirement for mixing in the module; it doesn’t matter what class you mix it into as long as that class defines the `each` instance method, and neither 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 **IO** 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. 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.” That is, 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`. Unlike Java’s **Interface**, no formal declaration is required for mix-ins; if we invented a new mixin that relied on (say) a class implementing the dereference operator `[]`, we could then mix it into any such class without otherwise modifying the classes themselves. 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.

Because **Enumerable** can deliver all the methods in Figure 3.7 (and some others) to any class that implements `each`, all Ruby classes that “quack like a collection” mix in

`Enumerable` for convenience. The methods `sort` (with no block), `max`, and `min` also require that the *elements* of the collection (not the collection itself) respond to the `<=>` method, which returns -1, 0, or 1 depending on whether its first argument is less than, equal to, or greater than its second argument. You can still mix in `Enumerable` even if the collection elements don't respond to `<=>`; you just can't use `sort`, `max`, or `min`. In contrast, in Java every collection class that implemented the `Enumerable` interface would have to ensure that its elements could be compared, whether that functionality was required or not.

| `<=>` is sometimes called the *spaceship operator* since some people think it looks like a flying saucer.

In Chapter 6 we will see how the combination of mix-ins, open classes, and `method_missing` allows you to write eminently readable unit tests using the RSpec tool.

Summary

- A mix-in is a set of related behaviors that can be added to any class that satisfies the mix-in's contract. For example, `Enumerable` is a set of behaviors on enumerable collections that requires the including class to define the `each` iterator.
- Ruby uses modules to group mix-ins. A module is mixed into a class by putting `include ModuleName` after the `class ClassName` statement.
- A class that implements some set of behaviors characteristic of another class, possibly by using mix-ins, is sometimes said to "quack like" the class it resembles. Ruby's scheme for allowing mix-ins without static type checking is therefore sometimes called duck typing.
- 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: Duck typing in the Time class

Ruby can represent times arbitrarily far in the past or future, can use timezones, and can handle non-Gregorian calendrical systems. Yet as we saw in Section 3.5, when a `Time` object receives a method call like `+` that expects an arithmetic argument, it attempts to return a representation of itself compatible with the Unix representation (seconds since the epoch). In other words, a `Time` object is not just a simple integer, but when necessary, it quacks like one.

Self-Check 3.7.1. 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 }`?

➊ 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.

Self-Check 3.7.2. Which statement is correct and why: (a) `include 'enumerable'` (b) `include Enumerable`

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

Althouh Ruby defines `each` for built-in collection classes like `Array`, you can define your own iterators using `each` as well. The idea of making iteration a first-class language feature first appeared in the CLU language invented by Barbara Liskov. Ruby's `yield` lets you define your own `each` methods that take a block, providing



an elegant way to allow collections to manage their own traversal.

<http://pastebin.com/9wJ3kBXY>

```
1 # return every n'th element in an enumerable
2 module Enumerable
3   def every_nth(count)
4     index = 0
5     self.each do |elt|
6       yield elt if index % count == 0
7       index += 1
8     end
9   end
10 end
11
12 list = (1..10).to_a # make an array from a range
13 list.every_nth(3) { |s| print "#{s}, " }
14 # => 1, 4, 7, 10,
15 list.every_nth(2) { |s| print "#{s}, " }
16 # => 1, 3, 5, 7, 9,
```

Figure 3.8: An example of using Ruby's `yield`, which is based on a construct introduced in the language CLU. Note that we define `every_nth` in the `Enumerable` module, which most collections mix in, as Section 3.7 describes.

Figure 3.8 shows how this unusual construct works. When a method containing `yield` is called, it starts executing until `yield` is reached; at that point, control is transferred to the block that was passed to the method. If `yield` had any arguments, those arguments become the arguments to the block.

A common use of `yield` is implementing iterators like `each` and `every_nth`. Unlike Java, in which you have to create an iterator by passing it a collection of some type and then repeatedly call `while (iter.hasNext())` and `iter.getNext()`, Ruby iterators allow turning the control structure “inside-out” and letting data structures manage *their own iteration*.

| Don't confuse this use of the term *yield* with the unrelated usage from operating systems, in which one thread or process is said to *yield* to another by giving up the CPU.

`yield` also enables reuse in situations where you need to “sandwich” some custom functionality inside of some common functionality. For example, consider an app that creates HTML pages and uses a standard HTML template for most pages that looks like this, where the only difference between different pages is captured by line 8:

<http://pastebin.com/iL25Hkxn>

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Report</title>
5      </head>
6      <body>
7          <div id="main">
8              ...user-generated content here...
9          </div>
10     </body>
11 </html>
```

In most languages, we could encapsulate the code that generates the boilerplate in lines 1–7 and 9–11 in methods called `make_header` and `make_footer`, and then require each method that wants to generate a page to do this:

<http://pastebin.com/uHbAXejk>

```
1  def one_page
2      page = ''
```

```
3     page << make_header()
4     page << "Hello"
5     page << make_footer()
6 end
7 def another_page
8     page = ''
9     page << make_header()
10    page << "World"
11    page << make_footer()
12 end
```

Since this code looks repetitive, we might instead wrap up both calls in a single method:

<http://pastebin.com/iXes5WSd>

```
1 def make_page(contents)
2     page = ''
3     page << make_header()
4     page << contents
5     page << make_footer()
6 end
7 #
8 def one_page
9     make_page("Hello")
10 end
11 def another_page
12     make_page("World")
13 end
```

But in Chapter 2 we learned that useful design patterns arise from the desire to *separate the things that change from those that stay the same*. **yield** provides a better way to encapsulate the common part—the boilerplate “around” the user content—in its own method:

<http://pastebin.com/ZwBsFVWm>

```
1 def make_page
```

```
2     page = ''  
3     page << make_header()  
4     page << yield  
5     page << make_footer()  
6   end  
7   def one_page  
8     make_page do  
9       "Hello"  
10    end  
11  end  
12  def another_page  
13    make_page do  
14      "World"  
15    end  
16  end
```

In this example, when `one_page` calls `make_page`, the `yield` at line 4 returns control to the block at line 9. The block executes, and its return value (in this case, "`Hello`") is returned to line 4 as the result of the `yield`, and gets appended to `page` (using the `<<` operator), after which `make_page` continues.

We can exploit Ruby's idiom for single-line blocks to boil this down to:

<http://pastebin.com/2CJRu4DY>

```
1  def make_page  
2    make_header << yield << make_footer  
3  end  
4  
5  def one_page  
6    make_page { "Hello" }  
7  end  
8  def another_page  
9    make_page { "World" }  
10 end
```

As we'll see, `yield` is actually how Rails implements HTML template rendering for views: the common HTML code that goes at the beginning and end of each page is rendered, and then `yield` is used to render the page-specific content in between. In

Chapter 10, we'll see how the combination of blocks and the Factory design pattern gives an exceptional degree of conciseness and code beauty in separating the things



that change from those that stay the same.

With this brief introduction to Ruby's most distinctive features, we're ready to meet the Rails framework.

Summary

- In the body of a method that takes a block as a parameter, `yield` transfers control to the block and optionally passes it an argument.
- Because the block is a closure, its scope is the one that was in effect when the block was defined, even though the method yielding to the block is executing in a completely different scope.
- Yielding is the general mechanism behind iterators: an iterator is simply a method that traverses some data structure and uses `yield` to pass one element at a time to the iterator's receiver.

Self-Check 3.8.1. Referring to Figure 3.8, observe that `every_nth` uses `elt` as an instance variable name in lines 5 and 6. Suppose that in line 13 we used `elt` instead of `s` as the name of the local variable in our block. What would be the effect of this change, if any, and why?

There would be no effect. `every_nth` and the block we pass to it execute in different scopes, so there is no "collision" of the local variable name `elt`.

Pitfall: Writing Java in Ruby. It takes some mileage to learn a new language's idioms and how it fundamentally differs from other languages. Common examples for Java programmers new to Ruby include:

- Thinking in terms of casting rather than method calls: `100.0 * 3` doesn't cast 3 to a Float, but calls `Float#*`.
- Reading `a.b` as "attribute `b` of object `a`" rather than "call method `b` on object `a`."
- Thinking in terms of classes and traditional static types, rather than duck typing. When calling a method on an object, or doing a mix-in, all that matters is whether the object responds to the method. The object's type or class are irrelevant.
- 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.

- 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.

 **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 true, whereas `['foo', 'bar'].include?(:foo)` is legal but false.

 **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.

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*

If you're coming to Ruby without knowledge of languages such as Lisp or Scheme, the functional programming idioms may be new to you. Unless you're familiar with JavaScript, you probably haven't used closures before. And unless you know [CLU](#), Ruby's `yield` may take some getting used to.

There's an old quip among programmers that "you can write Fortran in any language." This comment is perhaps unfair to Fortran—you can write good or bad code in any language—but the intention of the expression is to discourage carrying programming habits from one language into another where they are inappropriate, thereby 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 Rubyistic way to code what you need. We'll repeat this advice when we tackle JavaScript in Chapter 11. 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.

3.11 To Learn More

- The online documentation for [Ruby](#) gives details on the language, its classes, and the Rails framework.
- It's worth spending some time skimming the documentation for the [Ruby standard library](#). 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).
- [Programming Ruby](#) and *The Ruby Programming Language* ([Flanagan and Matsumoto 2008](#)), co-authored by Ruby inventor Yukihiro "Matz" Matsumoto, are definitive references for Ruby.

D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly Media, 2008. ISBN 0596516177. URL <http://www.amazon.com/Ruby-Programming-Language-David-Flanagan/dp/0596516177>.

OO and Classes

Project 3.1. How many class ancestors does the object [5](#) have? (Hint: use method chaining to follow the superclass chain all the way up to [BasicObject](#))

Project 3.2. Given that [superclass](#) returns [nil](#) when called on [BasicObject](#) but a non-[nil](#) value otherwise, write a Ruby method that, if passed any object, will print the object's class and its ancestor classes all the way up to [BasicObject](#).

Metaprogramming

Project 3.3. Building on the example in Section 3.5, take advantage of [Time](#)'s duck typing to define a method [at_beginning_of_year](#) that lets you write:

<http://pastebin.com/4am2KFMN>

```
1   Time.now.at_beginning_of_year + 1.day
2   # => 2011-01-02 00:00:00 -0800
```

Hint 1: The [Time documentation](#) will tell you that the `local` class method can be used to create a new `Time` object with a specified year, month and day.

Hint 2: The receiver of `at_beginning_of_year` in the above code is `now`, just as it was in the example in Section [3.5](#). But unlike that example, think carefully about how you'd like `now` to quack.

Project 3.4.

Define a method `attr_accessor_with_history` that provides the same functionality as `attr_accessor` but also tracks every value the attribute has ever had:

<http://pastebin.com/jk51z4h6>

```
1  class Foo
2    attr_accessor_with_history :bar
3  end
4  f = Foo.new      # => #<Foo:0x127e678>
5  f.bar = 3        # => 3
6  f.bar = :wowzo  # => :wowzo
7  f.bar = 'boo!'   # => 'boo!'
8  f.history(:bar) # => [3, :wowzo, 'boo!']
```

Mix-ins and Iterators

Project 3.5.

The `Enumerable` module includes an iterator `each_with_index` that yields each enumerable element along with an index starting from zero:

<http://pastebin.com/0qi0Jr1n>

```
1  include Enumerable
2  %w(alice bob carol).each_with_index do |person,index|
3    puts ">> #{person} is number #{index}"
4  end
5  >> alice is number 0
6  >> bob is number 1
7  >> carol is number 2
```

Create an iterator `each_with_custom_index` in module `Enumerable` that lets you determine the starting value and step of the indices:

<http://pastebin.com/0xFu5Z5>

```
1 include Enumerable
2 %w(alice bob carol).each_with_custom_index(3,2) do | person, index |
3   puts ">> #{person} is number #{index}"
4 end
5 >> alice is number 3
6 >> bob is number 5
7 >> carol is number 7
```

Project 3.6.

Recall that the first two integers in the Fibonacci sequence are 1 and 1, and each successive Fibonacci number is the sum of the previous two. Create a class that returns an iterator for the first n Fibonacci numbers. You should be able to use the class as follows:

<http://pastebin.com/N3vXRxDc>

```
1 # Fibonacci iterator should be callable like this:
2
3 f = FibSequence.new(6) # just the first 6 Fibonacci numbers
4 f.each { |s| print(s,':') } # => 1:1:2:3:5:8:
5 f.reject { |s| s.odd? } # => [2, 8]
6 f.reject(&:odd?) # => [2, 8] (a shortcut!)
7 f.map { |x| 2*x } # => [2, 2, 4, 6, 10, 16]
```

HINT: as long as objects of your class implement `each`, you can mix in `Enumerable` to get `reject`, `map`, and so on.

Project 3.7. Referring to the solution for Exercise 3.6, observe that `each` uses `lastfib` as a local variable (line 11). What will the following code do, and why?

<http://pastebin.com/xS1g6Sqy>

```
1 lastfib = "Surprise!"
2 FibSequence.new(3).each do |f|
3   print lastfib
4 end
```

Project 3.8. Referring again to Exercise [3.6](#), what will the following code do, and why?

<http://pastebin.com/JMaeGi2S>

```
1 FibSequence.new(3).each do
2   print "Rah"
3 end
```

Project 3.9. Implement an iterator `each_with_flattening` that behaves as follows:

<http://pastebin.com/eg8byu3v>

```
1 [1, [2, 3], 4, [[5, 6], 7]].each_with_flattening { |
s| print "#{s},"
2 >> 1, 2, 3, 4, 5, 6, 7
```

What assumption(s) must your iterator make about its receiver? What assumption(s) must it make about the elements of its receiver?

Project 3.10. Augment the `Enumerable` module with a new iterator, `each_permuted`, that returns the elements of a collection in a random order. The iterator may assume that the collection responds to `each` but shouldn't make any other assumptions about the elements. Hint: you may want to use the `rand` method in the Ruby standard library.

Project 3.11. An ordered binary tree is one in which every node has an element value and up to 2 children, each of which is itself an ordered binary tree, and all elements in the left subtree of some node are less than any element in the right subtree of that node.

Define a `BinaryTree` collection class that provides the instance methods `<<` (insert element), `empty?` (returns true if tree has no elements), and `each` (the standard iterator that yields each element in turn, in any order you desire).

Project 3.12. Augment your ordered binary tree class so that it also provides the following methods, each of which takes a block: `include?(elt)` (true if tree includes `elt`), `all?` (true if the given block is true for all elements), `any?` (true if the given block is true for any element), `sort` (sorts the elements according to a block of two arguments that returns -1, 0 or 1 depending on whether the first argument is less than, equal to, or greater than the second). **HINT:** A single line of

code suffices to do all this.

Project 3.13. Similar to the `days.ago` example in Section 3.5, define the appropriate conversions between Euros, US Dollars, and Yen so that you can type the following conversions:

<http://pastebin.com/v1YDjnBB>

```
1 # assumes 1 Euro=1.3 US dollars, 1 Yen=0.012 US dollars
2 5.dollars.in(:euros) # => 6.5
3 (1.euro - 50.yen).in(:dollars) # => 0.700
```

Project 3.14.

Which of these methods actually cause mutations to happen the way you expect?

<http://pastebin.com/vLz6kW2j>

```
1 def my_swap(a,b)
2   b,a = a,b
3 end
4
5 class Foo
6 attr_accessor :a, :b
7 def my_swap_2()
8   @b,@a = @a,@b
9 end
10 end
11
12 def my_string_replace_1(s)
13   s.gsub( /Hi/, 'Hello')
14 end
15
16 def my_string_replace_2(s)
17   s.gsub!( /Hi/, 'Hello')
18 end
```

Project 3.15.

Extend the `Time` class with a `humanize` method that prints out an informative phrase describing the time of day to the nearest fifteen-minute division, in twelve-hour mode, and making a special case for midnight:

<http://pastebin.com/k6eAvL4m>

```
1  >> Time.parse("10:47 pm").humanize
2  # => "About a quarter til eleven"
3  >> Time.parse("10:31 pm").humanize
4  # => "About half past ten"
5  >> Time.parse("10:07 pm").humanize
6  # => "About ten"
7  >> Time.parse("23:58").humanize
8  # => "About midnight"
9  >> Time.parse("00:29").humanize
10 # => "About 12:30"
```



Charles Antony Richard Hoare (1934–, called “Tony” by almost everyone) received the Turing Award in 1980 for “fundamental contributions to the definition and design of programming languages.”

There are two ways of constructing a software design: One way is to make it so simple that there are *obviously no deficiencies*, and the other way is to make it so complicated that there are *no obvious deficiencies*. The first method is far more difficult...The price of reliability is the pursuit of the utmost simplicity. *Tony Hoare*

- 4.1 [Rails Basics: From Zero to CRUD](#)
- 4.2 [Databases and Migrations](#)
- 4.3 [Models: Active Record Basics](#)
- 4.4 [Controllers and Views](#)
- 4.5 [Debugging: When Things Go Wrong](#)
- 4.6 [Form Submission: New and Create](#)
- 4.7 [Redirection and the Flash](#)
- 4.8 [Finishing CRUD: Edit/Update and Destroy](#)
- 4.9 [Fallacies and Pitfalls](#)
- 4.10 [Concluding Remarks: Designing for SOA](#)
- 4.11 [To Learn More](#)
- 4.12 [Suggested Projects](#)

Rails is a Ruby-based framework that uses three patterns from Chapter 2 to organize SaaS apps: Model–View–Controller for the app as a whole, Active Record for models backed by a relational database in the persistence tier, and Template View for constructing HTML pages. For conciseness, DRYness and productivity, Rails makes pervasive use of Ruby’s reflection and metaprogramming (Chapter 3) as well as [*convention over configuration*](#), a design paradigm that automates some configuration based on the names of data structures and variables. Although Rails presents a lot of machinery for the simple examples developed in this chapter, you will quickly “grow into” these features as your apps become more sophisticated.

As we saw in Chapter 2, Rails is a SaaS application framework that defines a

particular structure for organizing your application's code and provides an interface to a Rails application server such as Rack. The app server waits for a Web browser to contact your app and maps every incoming request (URL and HTTP method) to a particular action in one of your app's controllers. Rails consists of both the framework itself and a new command `rails` that is used to set up and manipulate Rails apps. Three main modules make up the heart of Rails' support for MVC: ActiveRecord for creating models, ActionView for creating views, and ActionController for creating controllers.

Using the explanation of Model–View–Controller in Chapter [2](#) as a reference framework, we will start from zero and create the Rotten Potatoes app described in Chapter [2](#) for maintaining a simple database of movie information. We will briefly visit each of the “moving parts” of a basic Rails application with a single model, in the following order:

- Creating the skeleton of a new app
- Routing
- The database and migrations
- Models and Active Record
- Controllers, views, forms, and CRUD

-T omits directories for tests that use Ruby's Test::Unit framework, since in Chapter [6](#) we will use the RSpec testing framework instead. `rails --help` shows more options for creating a new app.

Begin by logging into the bookware VM, changing to a convenient directory such as Documents (`cd Documents`), and creating a new, empty Rails app with `rails new myrottenpotatoes -T`. If all goes well, you'll see several messages about files being created, ending with “Your bundle is complete.” You can now `cd` to the newly-created `myrottenpotatoes` directory, called the **app root** directory for your new app. From now on, unless we say otherwise, all file names will be relative to the app root. Before going further, spend a few minutes examining the contents of the new app directory `myrottenpotatoes`, as described in Figure [4.1](#), to familiarize yourself with the directory structure common to all Rails apps.

Gemfile	list of Ruby gems (libraries) this app uses (Chapter 3)
Rakefile	commands to automate maintenance and deployment (Chapter 12)
app	your application
app/models	model code
app/controllers	controller code
app/views	view templates
app/views/layouts	page templates used by all views in the app (see text)
app/helpers	helper methods to streamline view templates
app/assets	static assets (JavaScript, images, stylesheets)

config	basic configuration information
config/environments	settings for running in development vs. production
config/database.yml	database configuration for development vs. production
config/routes.rb	mappings of URIs to controller actions
db	files describing the database schema
db/development.sqlite3	Data storage for SQLite development database
db/test.sqlite3	Database used for running tests
db/migrate/	Migrations (descriptions of changes to database schema)
doc	generated documentation
lib	additional app code shared among M, V, C
log	log files
public	error pages served by Web server
script	development tools, not part of app
tmp	temporary data maintained at runtime

Figure 4.1: The standard directory structure of a Rails project includes an app directory for the actual application logic with subdirectories for the app's models, views, and controllers, showing how Rails exposes the MVC architectural choice even in the arrangement of project files.

The message “Your bundle is complete” refers to the Gemfile that was automatically created for your app. While the Ruby standard library includes a [vast collection of useful classes](#), Rubygems is a system for managing external user-contributed Ruby libraries or **gems**. Bundler, a gem preinstalled with the bookware, looks for a Gemfile in the app’s root directory that specifies not only what gems your app depends on, but what versions of those gems. It might surprise you that there are already gem names in this file even though you haven’t written any app code, but that’s because Rails itself is a gem and also depends on several other gems. For example, if you open the Gemfile in an editor, you can see that sqlite3 is listed, because the default Rails development environment expects to use the SQLite3 database.



Pastebin is a service for copying-and-pasting book code. (You need to type URI if you’re reading the print book; it’s a link in ebooks.)

Edit your Gemfile by adding the following (anywhere in the file, as long as it’s not inside a block beginning with **group**).

<http://pastebin.com/AGMAxaag>

```
1 # use Haml for templates
```

```
2   gem 'haml'  
3   # use Ruby debugger  
4   group :development, :test do  
5     gem 'ruby-debug19'  
6   end
```

This change does two things. First, it specifies that we will use the Haml templating system rather than the built-in erb. Second, it specifies that we want to use the interactive debugger `ruby-debug19` during development and testing, but not in production.



Once you've made these changes to the Gemfile, run `bundle install --without production`, which checks if any gems specified in our Gemfile are missing and need to be installed. In this case no installation should be needed, since we've preloaded most gems you need in the bookware VM, so you should see "Your bundle is complete" as before. As the margin icon suggests, Bundler is the first of many examples we'll encounter of *automation for repeatability*: rather than manually installing the gems your app needs, listing them in the Gemfile and letting Bundler install them automatically ensures that the task can be repeated consistently in a variety of environments, eliminating mistakes in doing such tasks as a possible source of app errors. This is important because when you deploy your app, the information is used to make the deployment environment match your development environment.

Address already in use? If you see this error, you already have an app server listening on the default port of 3000, so find the terminal window where you started it and type Control-C to stop it if necessary.

Start the app with `rails server` and point a browser to `http://localhost:3000`. Recall from Chapter 2 that a URI that specifies only the hostname and port will fetch the home page. Most Web servers implement the convention that unless the app specifies otherwise, the home page is `index.html`, and indeed the welcome page you should be looking at is stored at `public/index.html`—the generic welcome page for new Rails apps.

If you now visit `http://localhost:3000/movies`, you should get a Routing Error from Rails. Indeed, you should verify that *anything* you add to the URI results in this error, and it's because we haven't specified any `routes` mapping URIs to app methods. Try `rake routes` and verify that unlike the result in Chapter 2, it prints nothing since there are no routes in our brand-new app. (You may want to open multiple Terminal windows so that the app can keep running while you try other commands.) More importantly, use an editor to open the file `log/development.log` and observe that the error message is logged there; this is where you look to find detailed error information when something goes wrong.

We'll show other problem-finding and debugging techniques in Section [4.5](#). To fix this error we need to add some routes. Since our initial goal is to store movie information in a database, we can take advantage of a Rails shortcut that creates RESTful routes for the four basic CRUD actions (Create, Read, Update, Delete) on a model. (Recall that RESTful routes specify self-contained requests of what operation to perform and what entity, or resource, to perform it on.) Edit config/routes.rb, which was auto-generated by the rails new command and is heavily commented. Replace the contents of the file with the following (the file is mostly comments, so you're not actually deleting much):

<http://pastebin.com/piLDY4eM>

```
1 Myrottenpotatoes::Application.routes.draw do
2   resources :movies
3   root :to => redirect('/movies')
4 end
```

Very important: In addition, **delete** the file public/index.html if it exists. Save the file and run rake routes again, and observe that because of our change to routes.rb, the first line of output says that the URI GET /movies will try to call the index action of the movies controller; this and most of the other routes in the table are the result of the line **resources :movies**, as we'll soon see. The root route '/ ', RottenPotatoes' "home page," will take us to the main Movie listings page by a mechanism we'll soon see called an [HTTP redirect](#).

Symbol or string? As with many Rails methods, **resources 'movies'** would also work, but idiomatically, a symbol indicates that the value is one of a fixed set of choices rather than an arbitrary string.

Using convention over configuration, Rails will expect this controller's actions to be defined in the class **MoviesController**, and if that class isn't defined at application start time, Rails will try to load it from the file app/controllers/movies_controller.rb. Sure enough, if you now reload the page <http://localhost:3000/movies> in your browser, you should see a different error: uninitialized constant MoviesController. This is good news: a Ruby class name is just a constant that refers to the class object, so Rails is essentially complaining that it can't find the **MoviesController** class, indicating that our route is working correctly! As before, this error message and additional

information are captured in the log file log/development.log.



Having covered the first two steps in the list—setting up the app skeleton and creating some initial routes—we can move on to setting up the database that will store the models, the "M" of MVC.

Summary: You used the following commands to set up a new Rails app:

- `rails new` sets up the new app; the `rails` command also has subcommands to run the app locally with WEBrick (`rails server`) and other management tasks.
- Rails and the other gems your app depends on (we added the Haml templating system and the Ruby debugger) are listed in the app's `Gemfile`, which Bundler uses to automate the process of creating a consistent environment for your app whether in development or production mode.
- To add routes in `config/routes.rb`, the one-line `resources` method provided by the Rails routing system allowed us to set up a group of related routes for CRUD actions on a RESTful resource.
- The log files in the `log` directory collect error information when something goes wrong.

ELABORATION: Automatically reloading the app

You may have noticed that after changing `routes.rb`, you didn't have to stop and restart the app in order for the changes to take effect. In development mode, Rails reloads all of the app's classes on every new request, so that your changes take effect immediately. In production this would cause serious performance problems, so Rails provides ways to change various app behaviors between development and production mode, as we'll see in Section [4.2](#).

ELABORATION: Non-resource-based routes

The shortcut `resources :movies` creates RESTful routes for CRUD, but any nontrivial app will have many additional controller actions beyond CRUD. The [Rails Routing from the Outside In guide](#) has much more detail, but one way to set up routes is to map components of the URI directly to controller and action names. As the table shows, the "wildcard" tokens `:controller` and `:action`, if present, determine the controller and action that will be invoked, and any other tokens beginning with `:` plus any additional parameters encoded in the URI will be made available in the `params` hash.

Route: `match ':controller/:action/:id'` or `match 'photos/preview/:id'`

Example URI: `/photos/preview/3`

Behavior: call `PhotosController#preview`
`params[]: { :id=>3 }`

Route: `match 'photos/preview/ :id'`

Example URI: `/photos/look/3?color=true`

Behavior: no route will match (look doesn't match `preview`)

Route: `match 'photos/:action/:id'`

Example URI: `/photos/look/3?color=true`

Behavior: call `PhotosController#look` (look matches `:action`)

`params[]: { :id=>3, :color=>'true' }`

Route: `match ':controller/:action/:vol/:num'`

Example URI: `/magazines/buy/3/5?newuser=true&discount=2`

Behavior: call `MagazinesController#buy`

`params[]: { :vol=>3, :num=>5, :newuser=>'true', :discount=>'2' }`

Self-Check 4.1.1. Recall the generic Rails welcome page you saw when you first created the app. In the development.log file, what is happening when the line `Started GET "assets/rails.png"` is printed? (Hint: recall the steps needed to render a page containing embedded assets, as described in Section 2.3.)

➊ The browser is requesting the embedded image of the Rails logo for the welcome page.

Self-Check 4.1.2. What are the two steps you must take to have your app use a particular Ruby gem?

➋ You must add a line to your `Gemfile` to add a gem and re-run `bundle install`. The persistence tier of a Rails app (see Figure 2.7) uses a relational database (RDBMS) by default, for the reasons we discussed in Chapter 2. Amazingly, you don't need to know much about RDBMSs to use Rails, though as your apps become more sophisticated it definitely helps. Just as we use the "lite" Web server WEBrick for development, Rails apps are configured by default to use SQLite3, a "lite" RDBMS, for development. In production you'd use a production-ready database such as MySQL, PostgreSQL or Oracle.

But more important than the "lightweight" aspect is that you wouldn't want to develop or test your app against the production database, as bugs in your code might accidentally damage valuable customer data. So Rails defines three *environments*—production, development, and test—each of which manages its own separate copy of the database, as specified in `config/database.yml`. The test database is entirely managed by the testing tools and should never be

modified manually: it is wiped clean and repopulated at the beginning of every testing run, as we'll see in Chapter 6.

An empty database was created for us by the `rails new` command in the file `db/development.sqlite3`, as specified in `config/database.yml`. We need to create a table for movie information. We could use the `sqlite3` command-line tool or a SQLite GUI tool to do this manually, but how would we later create the table in our production database when we deploy? Typing the same commands a second time isn't DRY, and the exact commands might be hard to remember. Further, if the production database is something other than SQLite3 (as is almost certainly the case), the specific commands might be different. And in the future, if we add more tables or make other changes to the database, we'll face the same problem.



A better alternative is a **migration**—a portable script for changing the database schema (layout of tables and columns) in a consistent and repeatable way, just as Bundler uses the Gemfile to identify and install necessary gems (libraries) in a consistent and repeatable way. Changing the schema using migrations is a three-step process:



Create a migration describing what changes to make. As with `rails new`, Rails provides a migration **generator** that gives you the boilerplate code, plus various helper methods to describe the migration.

Apply the migration to the development database. Rails defines a `rake` task for this.

Assuming the migration succeeded, update the test database's schema by running `rake db:test:prepare`.

Run your tests, and if all is well, apply the migration to the production database and deploy the new code to production. The process for applying migrations in production depends on the deployment environment; Chapter 9 covers how to do it using Heroku, the cloud computing deployment environment used for the examples in this book.



We'll use this 3-step process to add a new table that stores each movie's title, rating, description, and release date, to match Chapter 2. Each migration needs a name, and since this migration will create the `movies` table, we choose the name `CreateMovies`. Run the command `rails generate migration create_movies`, and if successful, you will find a new file under `db/migrate` whose name begins with the creation time and date and ends with the name you supplied, for example, `20111201180638_create_movies.rb`. (This naming scheme lets Rails apply migrations in the order they were created, since the file names will sort in date order.) Edit this file to make it look like Figure 4.2. As you can see, migrations illustrate an idiomatic use of blocks: the `ActiveRecord::Migration#create_table` method takes a block of 1

argument and yields to that block an object representing the table being created. The methods `string`, `datetime`, and so on are provided by this table object, and calling them results in creating columns in the newly-created database table; for example, `t.string 'title'` creates a column named `title` that can hold a string, which for most databases means up to 255 characters.

<http://pastebin.com/tmSkQX8b>

```
1 class CreateMovies < ActiveRecord::Migration
2   def up
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
14  def down
15    drop_table 'movies' # deletes the whole table and all its data!
16  end
17 end
```

Figure 4.2: A migration that creates a new `Movies` table, specifying the desired fields and their types. The documentation for the `ActiveRecord::Migration` class (from which all migrations inherit) is part of the [Rails documentation](#), and gives more details and other migration options.

Save the file and type `rake db:migrate` to actually apply the migration and create this table. Note that this housekeeping task also stores the migration number itself in the database, and by default it only applies migrations that haven't already been applied. (Type `rake db:migrate` again to verify that it does nothing the second time.) `rake db:rollback` will “undo” the last migration by running its `down` method. (Try it.) However, some migrations, such as those that delete data, can't be “undone”; in these cases, the `down` method should raise an

`ActiveRecord::IrreversibleMigration` exception.

Summary

- Rails defines three environments—development, production and test—each with its own copy of the database.
- A migration is a script describing a specific set of changes to the database. As apps evolve and add features, migrations are added to express the database changes required to support those new features.
- Changing a database using a migration takes three steps: create the migration, apply the migration to your development database, and (if applicable) after testing your code apply the migration to your production database.
- The `rails generate migration` generator fills in the boilerplate for a new migration, and the `ActiveRecord::Migration` class contains helpful methods for defining it.
- `rake db:migrate` applies only those migrations not already applied to the development database. The method for applying migrations to a production database depends on the deployment environment.

ELABORATION: Environments

Different environments can also override specific app behaviors. For example, production mode might specify optimizations that give better performance but complicate debugging if used in development mode. Test mode may “stub out” external interactions, for example, saving outgoing emails to a file rather than actually sending them. The file `config/environment.rb` specifies general startup instructions for the app, but `config/environments/production.rb` allows setting specific options used only in production mode, and similarly `development.rb` and `test.rb` in the same directory.

Self-Check 4.2.1. In line 3 of Figure 4.2, how many arguments are we passing to `create_table`, and of what types?

- Two arguments: the first is a string and the second is a block. We used poetry mode, allowing us to omit parentheses.

Self-Check 4.2.2. In Figure 4.2, the _____ method yields _____ to the block.

- `create_table`; the variable `t`

With our `Movies` table ready to go, we’ve completed the first three steps—app creation, routing, and initial migration—so it’s time to write some app code. The database stores the model objects, but as we said in Chapter 2, Rails uses the Active Record design pattern to “connect” models to the database, so that’s what we will explore next. Create a file `app/models/movie.rb` containing just these two lines:



<http://pastebin.com/PbFnAGt1>

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

Thanks to convention over configuration, those two lines in `movie.rb` enable a great deal of behavior. To explore some of it, stop the running application with Control-C and instead run `rails console`, which gives you an interactive Ruby prompt like `irb(main):001.0>` with the Rails framework and all of your application's classes already loaded. Figure 4.3 illustrates some basic ActiveRecord features by creating some movies in our database, searching for them, changing them, and deleting them (CRUD). As we describe the role of each set of lines, you should copy and paste them into the console to execute the code. The URI accompanying the code example will take you to a copy-and-pastable version of the

code on Pastebin.



<http://pastebin.com/2HETvZyz>

```
1 ##### Create  
2 starwars = Movie.create!(:title => 'Star Wars',  
3   :release_date => '25/4/1977', :rating => 'PG')  
4 # note that numerical dates follow European format: dd/mm/yy  
yy  
5 requiem = Movie.create!(:title => 'Requiem for a Dream',  
6   :release_date => 'Oct 27, 2000', :rating => 'R')  
7 # Creation using separate 'save' method, used when updating  
existing records  
8 field = Movie.new(:title => 'Field of Dreams',  
9   :release_date => '21-Apr-89', :rating => 'PG')  
10 field.save!  
11 field.title = 'New Field of Dreams'  
12 ##### Read  
13 pg_movies = Movie.where("rating = 'PG'")  
14 ancient_movies = Movie.where('release_date < :cutoff and rat  
ing = :rating',
```

```

15   :cutoff => 'Jan 1, 2000', :rating => 'PG')
16 pg_movies_2 = Movie.find_by_rating('PG')
17 Movie.find(3) # exception if key not found; find_by_id returns nil instead
18 ##### Update
19 starwars.update_attributes(:description => 'The best space western EVER',
20   :release_date => '25/5/1977')
21 requiem.rating = 'NC-17'
22 requiem.save!
23 ##### Delete
24 requiem.destroy
25 Movie.where('title = "Requiem for a Dream"')
26 ##### Find returns an enumerable
27 Movie.find_all_by_rating('PG').each do |mov|
28   mov.destroy
29 end

```

Figure 4.3: Although Model behaviors in MVC are usually called from the controller, these simple examples will help familiarize you with ActiveRecord's basic features before writing the controller.



Lines 1–6 (Create) create new movies in the database. `create!` is a method of `ActiveRecord::Base`, from which `Movie` inherits, as do nearly all models in Rails apps. ActiveRecord uses convention over configuration in three ways. First, it uses the name of the class (`Movie`) to determine the name of the database table corresponding to the class (`movies`). Second, it queries the database to find out what columns are in that table (the ones we created in our migration), so that methods like `create!` know which attributes are legal to specify and what types they should be. Third, it gives every `Movie` object attribute getters and setters similar to `attr_accessor`, except that these getters and setters do more than just modify an instance variable. To demonstrate that the getters and setters really are based on the database schema, try

`Movie.create(:chunky=>'bacon')`. The error message you get looks discouragingly long, but the first line should tell you that the actual error was that you specified an unknown attribute; the rest is Ruby's way of giving you more context, as we'll discuss further in Section 4.5. Before going on, type `Movie.all`, which returns a collection of all the objects in the table associated with the `Movie` class.

For the purposes of demonstration, we specified the release date in line 6 using a

different format than in line 3. Because Active Record knows from the database schema that `release_date` is a datetime column (recall the migration file in Figure 4.2), it will helpfully try to convert whatever value we pass for that attribute into a date.

Recall from Figure 3.1 that methods whose names end in ! are “dangerous.” `create!` is dangerous in that if anything goes wrong in creating the object and saving it to the database, an exception will be raised. The non-dangerous version, `create`, returns the newly-created object if all goes well or `nil` if something goes wrong. For interactive use, we prefer `create!` so we don’t have to check the return value each time, but in an application it’s much more common to use `create` and check the return value.

Lines 7–11 (Save) show that Active Record model objects in memory are independent of the copies in the database, which must be updated explicitly. For example, lines 8–9 create a new `Movie` object *in memory* without saving it to the database. (You can tell by trying `Movie.all` after executing lines 8–9. You won’t see *Field of Dreams* among the movies listed.) Line 10 actually persists the object to the database. The distinction is critical: line 11 changes the value of the movie’s `title` field, but *only on the in-memory copy*—do `Movie.all` again and you’ll see that the database copy hasn’t been changed. `save` and `create` both cause the object to be written to the database, but simply changing the attribute values doesn’t.

Lines 12–15 (Read) show one way to look up objects in the database. The `where` method is named for the WHERE keyword in SQL, the Structured Query Language used by most RDBMSs including SQLite3. You can specify a constraint directly as a string as in line 13, or use keyword substitution as in lines 14–15. Keyword substitution is always preferred because, as we will see in Chapter 12, it allows Rails to thwart **SQL injection** attacks against your app. As with `create!`, the time was correctly converted from a string to a `Time` object and thence to the database’s internal representation of time. Since the conditions specified might match multiple objects, `where` always returns an `Enumerable` on which you can call any of



`Enumerable`’s methods, such as those in Figure 3.7.

Line 16 (Read) shows another way to look up objects that uses `method_missing`. If you make a method call on a model class of the form `find_by_attribute` where `attribute` matches one of the model’s attributes, ActiveRecord overrides `method_missing` (just as we did in Section 3.5) to intercept that call, turning line 16 into something like `find.where("rating='PG'")`. You could even say `find_by_release_date_and_rating` and pass two arguments. These methods return *one* such object found (and you shouldn’t rely on which one); if you instead say `find_all_by_attribute`, you get back an `Enumerable` of all matching elements.

Line 17 (Read) shows the most primitive way of looking up objects, which is to

return a single object corresponding to a given primary key. Recall from Figure 2.11 that every object stored in an RDBMS is assigned a primary key that is devoid of semantics but guaranteed to be unique within that table. When we created our table in the migration, Rails included a numeric primary key by default. Since the primary key for an object is permanent and unique, it often identifies the object in RESTful URLs, as we saw in Section 2.7.

Lines 18–22 (Update) show how to Update an object. As with `create` vs. `save`, we have two choices: use `update_attributes` to update the database immediately, or change the attribute values on the in-memory object and then persist it with `save!` (which, like `create!`, has a “safe” counterpart `save` that returns `nil` rather than raising an exception if something goes wrong).

Lines 23–25 (Delete) show how to Delete an object. The `destroy` method (line 24) deletes the object from the database permanently. You can still inspect the in-memory copy of the object, but trying to modify it, or to call any method on it that would cause a database access, will raise an exception. (After doing the `destroy`, try `requiem.update_attributes(...)` or even `requiem.rating='R'` to prove this.)

Lines 26–29 show that the result of `find_all_by_attribute` does indeed quack like a collection—we can use `each` to iterate over it and delete each movie in turn.

This whirlwind overview of Active Record barely scratches the surface, but it should clarify how the methods provided by `ActiveRecord::Base` support the basic CRUD actions.

As a last step before continuing, you should `seed` the database with some movies to make the rest of the chapter more interesting, using the code in Figure 4.4.

<http://pastebin.com/hNB7kpWz>

```
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 # NOTE: the following line temporarily allows mass assignment
13 # (needed if you used attr_accessible/attr_protected in movi
```

```
e.rb)
14 Movie.send(:attr_accessible, :title, :rating, :release_date)
15 more_movies.each do |movie|
16   Movie.create!(movie)
17 end
```

Figure 4.4: Adding initial data to the database is called **seeding**, and is distinct from migrations, which are for managing changes to the schema. Copy this code into db/seeds.rb and run rake db:seed to run it.

Summary

- Active Record 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.
- Model instances can be **Created** either by calling **new** followed by **save** or by calling **create**, which combines the two.
- 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 stored in the database, and is therefore how objects are usually referenced in RESTful routes.
- Model instances can be **Read** (looked up) by using **where** to express the matching conditions or **find** to look up the primary key (ID) directly, as might occur if processing a RESTful URI that embeds an object ID.
- Model instances can be **Updated** with **update_attributes**.
- Model instances can be **Deleted** with **destroy**, after which the in-memory copy can still be read but not modified or asked to access the database.

ELABORATION: It quacks like a collection, but it isn't one

The object returned by ActiveRecord's **all**, **where** and **find**-based methods certainly quacks like a collection, but as we will see in Chapter 10, 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, allowing you to build up complex queries with multiple `where`s without paying the cost of doing the query each time.

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 4.3.1. Why are `where` and `find` class methods rather than instance methods?

 Instance methods operate on one instance of the class, but until we look up one or more objects, we have no instance to operate on.

Self-Check 4.3.2. Do Rails models acquire the methods `where` and `find` via (a) inheritance or (b) mix-in? (Hint: check the `movie.rb` file.)

 (a) they inherit from `ActiveRecord::Base`.

We'll complete our tour by creating some views to support the CRUD actions we just learned about. The RESTful routes we defined previously (`rake routes` to remind yourself what they are) expect the controller to provide actions for `index`, `show`, `new/create` (recall from Chapter 2 that creating an object requires two interactions with the user), `edit/update` (similarly), and `destroy`. Starting with the two easiest actions, `index` should display a list of all movies, allowing us to click on each one, and `show` should display details for the movie we click on.

For the `index` action, we know from the walk-through examples in Section 4.3 that `Movie.all` returns a collection of all the movies in the `Movies` table. Thus we need a controller method that sets up this collection and an HTML view that displays it. By convention over configuration, Rails expects the following for a method implementing the Show RESTful action on a Movie resource (note the uses of

singular vs. plural and of CamelCase vs. `snake_case`): 

- The model code is in class `Movie`, which inherits from `ActiveRecord::Base` and is defined in `app/models/movie.rb`
- 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 in turn inherits from `ActionController::Base`.
- Each instance method of the controller is named using `snake_lower_case`

according to the action it handles, so the `show` method would handle the Show action

- The Show view template is in `app/views/movies/show.html.haml`, with the `.haml` extension indicating use of the Haml renderer. Other extensions include `.xml` for a file containing XML Builder code (as we saw in Section 3.6), `.erb` (which we'll meet shortly) for Rails' built-in Embedded Ruby renderer, and many others.

The Rails module that choreographs how views are handled is `ActionView::Base`. Since we've been using the Haml markup for our views (recall we added the Haml gem to our Gemfile dependencies), our view files will have names ending in `.html.haml`. Therefore, to implement the Index RESTful action, we must define an `index` action in `app/controllers/movies_controller.rb` and a view template in `app/views/movies/index.html.haml`. Create these two files using Figure 4.5 (you will need to create the intermediate directory `app/views/movies/`).

<http://pastebin.com/KGWiEt09>

```
1 # This file is app/controllers/movies_controller.rb
2 class MoviesController < ApplicationController
3   def index
4     @movies = Movie.all
5   end
6 end
```

<http://pastebin.com/Bz9fuk34>

```
1 -# This file is app/views/movies/index.html.haml
2 %h2 All Movies
3
4 %table#movies
5   %thead
6     %tr
7       %th Movie Title
8       %th Rating
9       %th Release Date
```

```

10      %th More Info
11  %tbody
12    - @movies.each do |movie|
13      %tr
14        %td= movie.title
15        %td= movie.rating
16        %td= movie.release_date
17        %td= link_to "More about #{movie.title}", movie_path
(movie)

```

Figure 4.5: The controller code and template markup to support the Index RESTful action.

The controller method just retrieves all the movies in the Movies table using the `all` method introduced in the previous section, and assigns it to the `@movies` instance variable. Recall from the tour of a Rails app in Chapter 2 that instance variables defined in controller actions are available to views; line 12 of `index.html.haml` iterates over the collection `@movies` using `each`. There are three things to notice about this simple template.

First, the columns in the table header (`th`) just have static text describing the table columns, but the columns in the table body (`td`) use Haml's `=` syntax to indicate that the tag content should be evaluated as Ruby code, with the result substituted into the HTML document. In this case, we are using the attribute getters on `Movie` objects supplied by `ActiveRecord`.

Sanitization Haml's `=` syntax [sanitizes](#) the result of evaluating the Ruby code before inserting it into the HTML output, to help thwart cross-site scripting and similar attacks described in Chapter 12.

Second, we've given the table of movies the HTML ID `movies`. We will use this later for visually styling the page using CSS, as we learned about in Chapter 2.

Third is the call in line 17 to `link_to`, one of many helper methods provided by `ActionView` for creating views. As its [documentation](#) states, the first argument is a string that will appear as a link (clickable text) on the page and the second argument is used to create the URI that will become the actual link target. This argument can take several forms; the form we've used takes advantage of the URI helper `movie_path()` (as shown by `rake routes` for the `show` action), which takes as its argument an instance of a RESTful resource (in this case an instance of `Movie`) and generates the RESTful URI for the Show RESTful route for that object. As `rake routes` reminds you, the Show action for a movie is expressed by a URI `/movies/:id` where `:id` is the movie's primary key in the `Movies` table, so that's what the link target created by `link_to` will look like. To verify this, restart the application (`rails server` in the app's root directory) and visit

`http://localhost:3000/movies/`, the URI corresponding to the `index` action. If all is well, you should see a list of any movies in the database. If you use your browser's View Source option to look at the generated source, you can see that the links generated by `link_to` have URIs corresponding to the `show` action of each of the movies. (Go ahead and click one, but expect an error since we haven't yet created the controller's `show` method.)



The resources `:movies` line that we added in Section 4.1 actually creates a whole variety of helper methods for RESTful URIs, summarized in Figure 4.6. As you may have guessed, convention over configuration determines the names of the helper methods, and metaprogramming is used to define them on the fly as the result of calling `resources :movies`. The creation and use of such helper methods may seem gratuitous until you realize that it is possible to define much more complex and irregular routes beyond the standard RESTful ones we've been using so far, or that you might decide during the development of your application that a different routing scheme makes more sense. The helper methods insulate the views from such changes and let them focus on *what* to display rather than including code for *how* to display it.

Helper method	URI returned	RESTful Route and action	
<code>movies_path</code>	<code>/movies</code>	GET <code>/movies</code>	<code>index</code>
<code>movies_path</code>	<code>/movies</code>	POST <code>/movies</code>	<code>create</code>
<code>new_movie_path</code>	<code>/movies/new</code>	GET <code>/movies/new</code>	<code>new</code>
<code>edit_movie_path(m)</code>	<code>/movies/1/edit</code>	GET <code>/movies/:id/edit</code>	<code>edit</code>
<code>movie_path(m)</code>	<code>/movies/1</code>	GET <code>/movies/:id</code>	<code>show</code>
<code>movie_path(m)</code>	<code>/movies/1</code>	PUT <code>/movies/:id</code>	<code>update</code>
<code>movie_path(m)</code>	<code>/movies/1</code>	DELETE <code>/movies/:id</code>	<code>destroy</code>

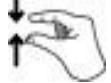
Figure 4.6: As described in the documentation for the `ActionView::Helpers` class, Rails uses metaprogramming to create route helpers based on the name of your ActiveRecord class. `m` is assumed to be an ActiveRecord `Movie` object. The RESTful routes are as displayed by the output of `rake routes`; recall that different routes may have the same URI but different HTTP methods, for example `create` vs. `index`.

ELABORATION: Reflection and metaprogramming for conciseness

More concisely, if `link_to`'s second argument is an ActiveRecord model of class `Movie`, `link_to` will implicitly call `movie_path` on it so you don't have to. So the `link_to` call could have been written as `link_to "More about #{movie.title}", movie`.

There's one last thing to notice about these views. If you View Source in your browser, you'll see it includes HTML markup that doesn't appear in our Haml template, such as a head element containing links to the assets/application.css stylesheet and a <title> tag. This markup comes from the application template, which "wraps" all views by default. The default file app/views/layouts/application.html.erb created by the rails new command uses Rails' erb templating system, but since we like Haml's conciseness, we recommend deleting that file and replacing it with Figure 4.7, then watch

Screencast [4.4.1](#) to understand how the "wrapping" process works.



<http://pastebin.com/0RU47cUy>

```
1 !!! 5
2 %html
3   %head
4     %title Rotten Potatoes!
5     = stylesheet_link_tag 'application'
6     = javascript_include_tag 'application'
7     = csrf_meta_tags
8
9   %body
10    = yield
```

Figure 4.7: Save this file as app/views/layouts/application.html.haml and delete the existing application.html.erb in that directory; this file is its Haml equivalent. Line 6 loads some basic JavaScript support; while we won't discuss JavaScript programming until Chapter 7, some of Rails' built-in helpers use it transparently. Line 7 introduces protection against [cross-site request forgery](#) attacks described in Chapter 12. We also made the title element a bit more human-friendly.

Screencast 4.4.1: [The Application layout](#)

The screencast shows that the app/views/layouts/application.html.haml template is used to "wrap" action views by default, using `yield` much like the example in Section 3.8.

<http://pastebin.com/TESrHmkk>

```
1 # in app/controllers/movies_controller.rb
2
3 def show
4   id = params[:id] # retrieve movie ID from URI route
5   @movie = Movie.find(id) # look up movie by unique ID
6   # will render app/views/movies/show.html.haml by default
7 end
```

Figure 4.8: An example implementation of the controller method for the Show action. A more robust implementation would use `find_by_id` and check the result, as we warned in Section 4.3. We'll show how to handle such cases in Chapter 7.

<http://pastebin.com/hnmsYYB5>

```
1 -# in app/views/movies/show.html.haml
2
3 %h2 Details about #{@movie.title}
4
5 %ul#details
6   %li
7     Rating:
8     = @movie.rating
9   %li
10    Released on:
11    = @movie.release_date.strftime("%B %d, %Y")
12
13 %h3 Description:
14
15 %p#description= @movie.description
16
17 = link_to 'Back to movie list', movies_path
```

Figure 4.9: An example view to go with Figure 4.8. For future CSS styling, we gave unique ID's to the bullet-list of details (ul) and the one-paragraph description (p). We used the [strftime library function](#) to format the date more attractively, and the `link_to` method with the RESTful helper `movies_path` (Figure 4.6) to provide a convenient link back to the listings page. In general, you can append `_path` to any of the RESTful resource helpers in the leftmost column of the `rake routes` output to call a method that will generate the corresponding RESTful URI.



On your own, try creating the controller action and view for `show` using a similar process:

Use `rake routes` to remind yourself what name you should give to the controller method and what parameters will be passed in the URI

In the controller method, use the appropriate ActiveRecord method introduced in Section 4.3 to retrieve the appropriate `Movie` object from the database and assign it to an instance variable

Create a view template in the right location in the `app/views` hierarchy and use Haml markup to show the various attributes of the `Movie` object you set up in the controller method

Exercise your method and view by clicking on one of the movie links on the index view

Once you're done, you can check yourself against the sample controller method in Figure 4.8 and the sample view in Figure 4.9. Experiment with other values for the arguments to `link_to` and `strftime` to get a sense of how they work.

Since the current "bare-bones" views are ugly, as long as we're going to keep working on this app we might as well have something more attractive to look at.

Copy the simple CSS styling below into

`app/assets/stylesheets/application.css`, which is already included by line 5 of the `application.html.haml` template.

<http://pastebin.com/LsLngdin>

```
1  /* Simple CSS styling for RottenPotatoes app */
2
3  /* Add these lines to app/assets/stylesheets/application.css */
4
5  html, body {
6      margin: 0;
7      padding: 0;
8      background: White;
9      color: DarkSlateGrey;
10     font-family: Tahoma, Verdana, sans-serif;
11     font-size: 10pt;
12 }
13 div#main {
14     margin: 0;
15     padding: 0 20px 20px;
```

```
16  a {
17      background: transparent;
18      color: maroon;
19      text-decoration: underline;
20      font-weight: bold;
21  }
22  h1 {
23      color: maroon;
24      font-size: 150%;
25      font-style: italic;
26      display: block;
27      width: 100%;
28      border-bottom: 1px solid DarkSlateGrey;
29  }
30  h1.title {
31      margin: 0 0 1em;
32      padding: 10px;
33      background-color: orange;
34      color: white;
35      border-bottom: 4px solid gold;
36      font-size: 2em;
37      font-style: normal;
38  }
39  table#movies {
40      margin: 10px;
41      border-collapse: collapse;
42      width: 100%;
43      border-bottom: 2px solid black;
44  }
45  table#movies th {
46      border: 2px solid white;
47      font-weight: bold;
48      background-color: wheat;
49  }
50  table#movies th, table#movies td {
51      padding: 4px;
52      text-align: left;
53  }
54  #notice, #warning {
55      background: rosybrown;
56      margin: 1em 0;
57      padding: 4px;
```

```
58    }
59    form label {
60      display: block;
61      line-height: 25px;
62      font-weight: bold;
63      color: maroon;
64    }
65
66
```

Summary:

- The Haml templating language allows you to intersperse HTML tags with Ruby code for your views. The result of evaluating Ruby code can either be discarded or interpolated into the HTML page.
- For conciseness, Haml relies on indentation to reveal HTML element nesting.
- 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.
- Rails provides various helper methods that take advantage of the RESTful route URLs, including `link_to` for generating HTML links whose URLs refer to RESTful actions.

Self-Check 4.4.1. In Figure 4.6, why don't the helper methods for the New action (`new_movie_path`) and Create action (`movies_path`) take an argument, as the Show or Update helpers do?

 Show and Update operate on existing movies, so they take an argument to identify which movie to operate on. New and Create by definition operate on not-yet-existing movies.

Self-Check 4.4.2. In Figure 4.6, why doesn't the helper method for the Index action take an argument? (HINT: The reason is different than the answer to Self-Check 4.4.1.)

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

Self-Check 4.4.3. In Figure 4.5, why is there no `end` corresponding to the `do` in line

12?

 Unlike Ruby itself, Haml relies on indentation to indicate nesting, so the `end` is supplied by Haml when executing the Ruby code in the `do`.

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. Although we take steps in this book to minimize the pain, such as using Test-Driven Development (Chapter 6) to avoid many problems and providing a VM image with a consistent environment, errors *will* occur. You can react most productively by remembering the acronym **RASP**: Read, Ask, Search, Post.

Read the error message. Ruby's error messages can look disconcertingly long, but a long error message is often 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. This will require some understanding of the syntax of the erroneous code, which you might lack if you blindly cut-and-pasted someone else's code with no understanding of how it works or what assumptions it makes. Of course, a syntax error due to cut-and-paste is just as likely to occur when reusing your own code as someone else's, but at least you understand your own code (right?).

An amusing perspective on the perils of blind "shotgun problem solving" is the Jargon File's [hacker koan "Tom Knight and the Lisp Machine."](#)

A particularly common proximate cause of Ruby errors is "Undefined method `foobar` for `nil:NilClass`", which means "You tried to call method `foobar` on an object whose value is `nil` and whose class is `NilClass`, which doesn't define `foobar`." (`NilClass` is a special class whose only instance is the constant `nil`.) This often 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. But if the computation occurred in another method "upstream," the backtrace can help you figure out where.

In SaaS apps using Rails, this confusion can be compounded if the failed computation happens in the controller action but the invalid object is passed as an instance variable and then dereferenced in the view, as in the following excerpts from a controller and view:

<http://pastebin.com/yJY97jmE>

```
1  # in controller action:  
2  def show  
3  
@movie = Movie.find_by_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 Haml view:
8
9  %h1= @movie.title
10 -
# will give "undefined method 'title' for nil:NilClass" if @movie is nil
```

Ask a coworker. If you are programming in pairs, two brains are better than one. If you're in an "open seating" configuration, or have instant messaging enabled, put the message out there.

Search for the error message. You'd be amazed at how often experienced developers deal with an error by using a search engine such as Google to look up key words or key phrases in the error message. You can also search sites like [StackOverflow](#) and [ServerFault](#), 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.

Post a question on one of those sites if all else fails. Be as specific as possible about what went wrong, what your environment is, and how to reproduce the problem:

- **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 transcript](#) of this question on StackOverflow. At 6:02pm, the developer provided specific information, such as the name and version of his operating system, the specific commands he successfully ran, and the unexpected error that resulted. Other helpful voices chimed in asking for specific additional information, by 7:10pm, two of the answers had identified the problem.

While it's impressive that he got his answer in just over an hour, it means he also lost an hour of coding time, which is why you should post a question only after you've exhausted the other alternatives. How can you make progress on debugging problems on your own? There are two kinds of problems. In the first kind, an error or exception of some kind stops the app in its tracks. Since Ruby is an interpreted

language, syntax errors can cause this (unlike Java, which won't even compile if there are syntax errors). Here are some things to try if the app stops in its tracks.

- Exploit automatic indentation and syntax highlighting. If your text editor insists on indenting a line farther than you want it to be indented, you may have forgotten to close a parenthesis, brace, or `do...end` block somewhere upstream, or you may have forgotten to "escape" a special character (for example, a single-quote inside a single-quoted string). If your editor isn't so equipped, you can either write your code on stone tablets, or switch to one of the more productive modern editors suggested in Appendix A.
- Look in the log file, usually `log/development.log`, for complete error information including the backtrace. In production apps, this is often your only alternative, as Rails apps are usually configured to display a more user-friendly error page in production mode, rather than the error backtrace you'd see if the error occurred in development mode.

In the second kind of problem, the app runs but produces an incorrect result or behavior. Most developers use a combination of two approaches to debug such problems. The first is to insert **instrumentation**—extra statements to record values of important variables at various points during program execution. There are various places we can instrument a Rails SaaS app—try each of the below to get a feel for how they work:

`printf debugging` is an old name for this technique, from the C library function that prints a string on the terminal.



- Display a detailed description of an object in a view. For example, try inserting `= debug(@movie)` or `= @movie.inspect` in any view (where the leading `=` tells Haml to execute the code and insert the result into the view).
- "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 `logger.debug(message)` to print `message` to the log. `logger` is available in models and controllers and can record messages with a variety of urgencies; compare `config/environments/production.rb` with `development.rb` to see how the default logging level differs in production vs. development environments.



The second way to debug correctness problems is with an interactive debugger. We already installed the `ruby-debug19` gem via our Gemfile; to use the

debugger in a Rails app, start the app server using `rails server --debugger`, and insert the statement `debugger` 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. In Section 4.7, we'll show how to use the debugger to shed some light on Rails internals.

To debug non-Rails Ruby apps, insert the line `require 'ruby-debug'` at the beginning of the program.

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 `ruby-debug19`, start the app server with `rails server --debugger`, and place the statement `debugger` at the point in your code where you want to break.

Self-Check 4.5.1. 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 4.5.2. Of the three debugging methods described in this section, which ones 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.

Our last look at views will deal with a slightly more complex situation: that of submitting a form, such as for creating a new movie or updating an existing one. There are three problems we need to address:

How do we display a fill-in form to the user?

How is the information filled in by the user actually made available to the controller action, so that it can be used in a `create` or `update` ActiveRecord call?

What resource should be returned and displayed as the result of a RESTful request to create or update an item? Unlike when we ask for a list of movies

or details about a movie, it's not obvious what to display as the result of a create or update.



Of course, before we go further, we need to give the user a way to get to the fill-in form we're about to create. Since the form will be for creating a new movie, it will correspond to the RESTful action `new`, and we will follow convention by placing the form in `app/views/movies/new.html.haml`. We can therefore take advantage of the automatically-provided RESTful URI helper `new_movie_path` to create a link to the form. Do this by adding a single line to the end of `index.html.haml`:

<http://pastebin.com/6wLiit6M>

```
1  -# add to end of index.html.haml
2
3  = link_to 'Add new movie', new_movie_path
```

What controller action will be triggered if the user clicks on this link? Since we used the URI helper `new_movie_path`, it will be the `new` controller action. We haven't defined this action yet, but for the moment, since the user is creating a brand-new movie entry, the only thing the action needs to do is cause the corresponding view for the new action to be rendered. Recall that by default, every controller method automatically tries to render a template with the corresponding name (in this case `new.html.haml`), so you can just add the following trivial `new` method to `movies_controller.rb`:

<http://pastebin.com/uFdJNmB>

```
1  def new
2      # default: render 'new' template
3  end
```

Rails makes it easy to describe a fill-in form using [form tag helpers](#) available to all views. Put the code in Figure 4.10 into `app/views/movies/new.html.haml` and watch Screencast [4.6.1](#) for a description of what's going on in it.

<http://pastebin.com/w7jdzawW>

```

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

```

Figure 4.10: The form the user sees for creating and adding a new movie to Rotten Potatoes.

Screencast 4.6.1: [Views with fill-in Forms](#)

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.

As the screencast mentions, 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.

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.haml`, and arranged to have the form submitted to the `create` controller method. All that remains is to use the information in `params` (the form field values) to actually create the new movie in the database.

Summary

- Rails provides form helpers to generate a fill-in form whose fields are related

to the attributes of a particular type of ActiveRecord object.

- 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`).
- When the form is submitted, the controller action can inspect `params[]`, which will contain a key for each form field whose value is the user-supplied contents of that field.

Self-Check 4.6.1. In line 3 of Figure 4.10, 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. As Figure 4.6 shows, 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 4.6.2. Given that submitting the form shown in Figure 4.10 will create a new movie, why is the view called `new.html.haml` rather than `create.html.haml`?

 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.

Recall from the examples in Section 4.3 that the `Movie.create!` call takes a hash of attribute names and values to create a new object. As Screencast 4.7.1 shows, the form field names created by the `form tag` helpers all have names of the form `params['movie']['title']`, `params['movie']['rating']`, and so on. As a result, the value of `params[:movie]` is exactly a hash of movie attribute names and values, which we can pass along directly using `Movie.create!(params[:movie])`. The screencast also shows the helpful technique of using debug breakpoints to provide a detailed look “under the hood” during execution of

a controller action. 

Screencast 4.7.1: [The Create action](#)

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 4.11 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. Like many Rails methods, `params[]` can take either a symbol or a string—in fact `params` is not a regular hash at all, but a `HashWithIndifferentAccess`, a Rails class that quacks like a hash but allows its keys to be accessed as either symbols or strings.

n	execute next line
s	execute next statement
f	finish current method call and return
p <i>expr</i>	print <i>expr</i> , which can be anything that's in scope within the current stack frame
eval <i>expr</i>	evaluate <i>expr</i> ; can be used to set variables that are in scope, as in <code>eval x=5</code>
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 <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	continue execution until next breakpoint
q	quit program

Figure 4.11: Command summary of the interactive Ruby debugger.

That brings us to the third question posed at the beginning of Section 4.6: what view should we display when the `create` action completes? To be consistent with other actions like `show`, we could create a view `app/views/movies/create.html.haml` containing a nice message informing the user of success, but it seems gratuitous to have a separate view just to do that. What most web apps do instead is return the user to a more useful page—say, the home page, or the list of all movies—but they display a success message as an added element on that page to let the user know that their changes were



successfully saved.

Rails makes it easy to implement this behavior. To send the user to a different page, `redirect_to` causes a controller action to end not by rendering a view, but by restarting a whole new request to a different action. Thus, `redirect_to movies_path` is just as if the user suddenly requested the RESTful Index action GET `movies` (that is, the action corresponding to the helper

`movies_path`): the `index` action will run to completion and render its view as usual. In other words, a controller action must finish by either rendering a view or redirecting to another action. Remove the debug breakpoint from the controller action (which you inserted if you modified your code according to Screencast [4.7.1](#)) and modify it to look like the listing below; then test out this behavior by reloading the movie listing page, clicking Add New Movie, and submitting the form. 

```
1 # in movies_controller.rb
2 def create
3   @movie = Movie.create!(params[:movie])
4   redirect_to movies_path
5 end
```

For simplicity we defer discussion until Section [7.2.1](#) of `attr_protected`, without which an attacker could manipulate the form passed to `Movie.create!` to [set arbitrary attributes](#) that shouldn't be changeable by regular users.

Of course, to be user-friendly, we would like to display a message acknowledging that creating a movie succeeded. (We'll soon deal with the case where it fails.) The hitch is that when we call `redirect_to`, it starts a whole new HTTP request; and since HTTP is stateless, all of the variables associated with the `create` request are gone.

 To address this common scenario, the `flash[]` is a special method that quacks like a hash, but persists from the current request to the next. (In a moment we'll explore how Rails accomplishes this.) In other words, if we put something into `flash[]` during the current controller action, we can access it during the *subsequent* action. The entire hash is persisted, but by convention, `flash[:notice]` is used for informational messages and `flash[:warning]` is used for messages about things going wrong. Modify the controller action to store a useful message in the `flash`, and try it out: 

<http://pastebin.com/N1n4Pkr0>

```
1 # in movies_controller.rb
2 def create
3   @movie = Movie.create!(params[:movie])
4   flash[:notice] = "#{@movie.title} was successfully created."
```

```
5      redirect_to movies_path  
6  end
```

What happened? Even though creating a new movie appears to work (the new movie shows up in the list of all movies), there's no sign of the helpful message we just created. As you've probably guessed, that's because we didn't actually modify any of the views to display that message!

But which view should we modify? 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.haml` 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.

Make `application.html.haml` look like Figure 4.12—this requires adding four lines of code between `%body` and `=yield` to display any pending flash messages at the beginning of the page body.

<http://pastebin.com/bW1FzEr1>

```
1 -# this goes just inside %body:  
2 - if flash[:notice]  
3   #notice.message= flash[:notice]  
4 - elsif flash[:warning]  
5   #warning.message= flash[:warning]
```

Figure 4.12: Note the use of CSS for styling the flash messages: each type of message is displayed in a `div` whose unique ID is either `notice` or `warning` depending on the message's type, but that share the common class `message`. This gives us the freedom in our CSS file to either style the two types of messages the same by referring to their class, or style them differently by referring to their IDs. Remarkably, Haml's conciseness allows expressing each `div`'s class and ID attributes *and* the message text to be displayed all on a single line.

Try styling all `flash` messages to be printed in red text and centered. You'll need to add the appropriate CSS selector(s) in

app/assets/stylesheets/application.css to match the HTML elements that display the `flash` in the Application page template. The CSS properties `color: red` and `text-align: center` will get these effects, but feel free to experiment with other visual styles, colors, borders, and so on.

If you do any nontrivial CSS work, you'll want to use a dedicated CSS editor, such as the open-source and cross-platform [Amaya](#) or one of many commercial products.

Summary

- 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.
- You can DRY out your views by putting markup to display `flash` messages in one of the application's templates, rather than having to replicate it in every view that might need to display such messages.

ELABORATION: The Session

Actually, the flash is just a special case of the more general facility `session[]`. Like the flash, the session quacks like a hash whose contents persist across requests from the same browser, but unlike the flash, which is automatically erased following the next request, anything you put in the session stays there permanently until you delete it. You can either `session.delete(:key)` to delete individual items just as with a regular hash, or use the `reset_session` method to nuke the whole thing. Keep in mind that the session is based on cookies, so sessions from different users are independent. Also, as we note in Fallacies & Pitfalls, be careful how much you store in the session.

Self-Check 4.7.1. 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 4.7.2. In Figure 4.12, given that we are going to output an HTML tag, why does line 2 begin with `-` rather than `=`?

`=` directs Haml to evaluate the Ruby expression and substitute it into the view, but we don't want the value of the `if`-expression to be placed in the view—we want

the actual HTML tag, which Haml generates from `#notice.message`, plus the result of evaluating `flash[:notice]`, which is correctly preceded by `=`.

 We can now follow a similar process to add the code for the `update` functionality. Like `create`, this requires two actions—one to display the form with editable information (`edit`) and a second to accept the form submission and apply the updated information (`update`). Of course, we first need to give the user a way to specify the Edit action, so before going further, modify the `show.html.haml` view so its last two lines match the code below, where line 2 uses the helper `edit_movie_path` to generate a RESTful URI that will trigger the `edit` action for `@movie`.

<http://pastebin.com/ZD1y6TYc>

```
1  -
# modify last 2 lines of app/views/movies/show.html.haml to:
2  = link_to 'Edit info', edit_movie_path(@movie)
3  = link_to 'Back to movie list', movies_path
```

In fact, as Figure [4.13](#) shows, the new/create and edit/update action pairs are similar in many respects.

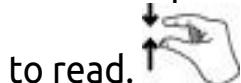
	Create	Update
Parameters passed to view	none	existing instance of <code>Movie</code>
Default form field values	blank	existing movie attributes
Submit button label	“Create Movie” (or “Save Changes”)	“Update Movie” (or “Save Changes”)
Controller actions	<code>new</code> serves form, <code>create</code> receives form and modifies database	<code>edit</code> serves form, <code>update</code> receives form and modifies database
<code>params[]</code>	Attribute values for new movie	Updated attribute values for existing movie

<http://pastebin.com/PhapSLEN>

Figure 4.13: The `edit/update` action pair is very similar to the `new/create` action pair we've already implemented (top); in fact, The Haml markup for the `edit` view differs from the `new` view only in line 3 (bottom). In Chapter 7 we will discover ways to DRY out the common parts.

Shouldn't we DRY out similar things? In Chapter 7 we'll show a way to take advantage of this similarity to DRY out the views, but for now we'll tolerate a little duplication in order to finish the example.)

Use Figure 4.13 to create the `edit.html.haml` view, which is almost identical new view (Figure 4.10)—the only difference is line 3, which specifies the RESTful route for form submission. As `rake routes` tells us, the `new` action requires an HTTP POST to the URI `/movies`, so Figure 4.10 uses `:method=>:post` and the URI helper `movies_path` in the form action. In contrast, the `update` action requires an HTTP PUT to `/movies/:id` where `:id` is the primary key of the resource to be updated, so line 3 of Figure 4.13 specifies `:method=>:put` and uses the URI helper `movie_path(@movie)` to construct the URI for editing this specific movie. We could have constructed the URIs manually, using `form_tag "/movies"` in `new.html.haml` and `form_tag "/movies/#{@movie.id}"` in `edit.html.haml`, but the URI helpers are more concise, convey intent more clearly, and are independent of the actual URI strings, should those have a reason to change. As we'll see, when your app introduces relationships among different kinds of resources, such as a moviegoer having favorite movies, the RESTful URIs become more complicated and the helpers become correspondingly more concise and easy



to read.
Below are the actual controller methods you'll need to add to `movies_controller.rb` to try out this feature, so go ahead and add them.
<http://pastebin.com/jdTS5P7Q>

```
1  # in movies_controller.rb
2
3  def edit
4      @movie = Movie.find params[:id]
5  end
6
7  def update
8      @movie = Movie.find params[:id]
9      @movie.update_attributes!(params[:movie])
10
11     flash[:notice] = "#{@movie.title} was successfully updated."
12     redirect_to movie_path(@movie)
```

```
12 end  
13
```

Try clicking on the Edit link you inserted above to edit an existing movie. Observe that when updating an existing movie, the default filled-in values of the form fields correspond to the movie's current attributes. This is because helpers such as `text_field` in line 6 of the new or edit templates will by default look for an instance variable whose name matches their first argument—in this case, the first argument is `:movie` so the `text_field` helper will look for a variable `@movie`. If it exists and corresponds to an ActiveRecord model, the helper assumes that this form is for editing an existing object, and `@movie`'s current attribute values will be used to populate the form fields. If it doesn't exist or doesn't respond to the attribute method in the second argument (`'title'`), the form fields will be blank. This behavior is a good reason to name your instance variable `@movie` rather than (say) `@my_movie`: you can still get the extra functionality from the helpers, but you'll



have to pass extra arguments to them.

The last CRUD action is Delete, which Figure 4.3 shows can be accomplished by calling `destroy` on an ActiveRecord model. As with the Update action, it's common practice to respond to a Delete by destroying the object and then returning the user to some other useful page (such as the Index view) and displaying a confirmation message that the item was deleted, so we already know how to write the controller method—add the following lines to `movies_controller.rb`:

<http://pastebin.com/8ZYbFUcb>

```
1 def destroy  
2   @movie = Movie.find(params[:id])  
3   @movie.destroy  
4   flash[:notice] = "Movie '#{@movie.title}' deleted."  
5   redirect_to movies_path  
6 end
```

(Recall from the explanation accompanying Figure 4.3 that even after destroying an object in the database, the in-memory copy can still be queried for its attributes as long as we don't try to modify it or ask it to persist itself.)

As we did with Edit, we'll provide access to the Delete action from each movie's Show page. What kind of HTML element should we use? `rake routes` tells us the Delete action requires the URI `/movies/:id` with the HTTP verb `DELETE`. This

seems similar to Edit, whose URI is similar but uses the GET method. Since we've been using the URI helpers to generate routes, we can still use `link_to` in this case, but its behavior is a bit different from what you might expect.

<http://pastebin.com/Cr8EXQaH>

```
1      -# Our Edit link from previous example:  
2      = link_to 'Edit info', edit_movie_path(@movie)  
3      -  
# This Delete link will not really be a link, but a form:  
4  
= link_to 'Delete', movie_path(@movie), :method => :delete
```

If you examine the HTML generated by this code, you'll find that Rails generates a link that includes the unusual attribute `data-method="delete"`. As we will see in Chapter 11, this is a way to allow JavaScript-enabled browsers to more safely handle the destructive delete operation. From a user interface point of view, destructive actions such as Delete are better handled using a button rather than a link, so the following code is usually more appropriate:

<http://pastebin.com/mTXuE5up>

```
1  
= button_to 'Delete', movie_path(@movie), :method => :delete,  
:confirm => 'Are you sure?'
```

`button_to` is very similar to `link_to`—Screencast 4.8.1 shows how it generates the form.

Screencast 4.8.1: [The button_to helper](#)

The `button_to` helper takes three arguments: the button's textual name, an options hash specifying the route or a URI encoding the route, and another hash of optional HTML options. In our usage, the second argument is supplied by the `movie_path` helper, and we provide keys in the HTML options to specify the HTTP submission method and to display a confirmation dialog for the user. The result is a form with only a single user-visible control, namely the button itself, that uses some JavaScript tricks we'll learn in Chapter 11 to supply the confirmation dialog.

ELABORATION: GET vs. POST

Long before RESTfulness became a prominent SaaS concept, there was already a general guideline that SaaS app requests that used GET should always be “safe”—they should not cause any side effects such as deleting an item or purchasing something, and should be safely repeatable. In the language of computer science, GETs should be **idempotent**—doing it many times has the same effect as doing it once. Your browser reflects this guideline: you can always reload or refresh a page that was fetched via GET, but if you try to reload a page that resulted from a POST operation, most browsers will display a warning asking if you really want to resubmit a form. The “GETs should be safe” guideline became particularly important when search engines became widespread, since they crawl (explore the Web) by following links. It would be bad if Google triggered millions of spurious purchases every time it crawled an e-commerce site. So in this sense, Rails’ helpers are just providing convenient shortcuts that implement existing best practice.



Try modifying the index view (list of all movies) so that each table row displaying a movie title also includes an Edit link that brings up the edit form for that movie and a Destroy button that deletes that movie with a confirmation dialog.

Self-Check 4.8.1. Why does the form in new.html.haml submit to the `create` method rather than the `new` method?

As we saw in Chapter 2, creating a new record requires two interactions. The first one, `new`, loads the form. The second one, `create`, submits the form and causes the actual creation of the new record.

Self-Check 4.8.2. 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?

Render and redirect are two different ways to reply to a request. Each request needs exactly one reply.

Summary

- Rails provides various helpers for creating HTML forms that refer to ActiveRecord models. In the controller method that receives the form submission, the keys in the `params` hash are the form fields’ name attributes and the corresponding values are the user-selected choices for those fields.
- Creating and updating an object are resources whose visible representation is just the success or failure status of the request. For user friendliness, rather than displaying a web page with just success or failure and requiring the user to click to continue, we can instead `redirect_to` a more useful page such as `index`. Redirection is an alternative way for a controller action to finish, rather than rendering a view.
- For user friendliness, it’s typical to modify the application layout to display messages stored in `flash[:notice]` or `flash[:warning]`, which persist until the next request so they can be used with `redirect_to`.

- To specify the URIs required by both form submissions and redirections, we can use RESTFUL URI helpers like `movies_path` and `edit_movie_path` rather than creating the URIs manually.
-

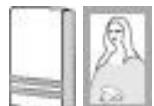
 **Pitfall: Modifying the database manually rather than using migrations, or managing gems manually rather than using Bundler.** Especially if you've come from other SaaS frameworks, it may be tempting to use the SQLite3 command line or a GUI database console to manually add or change database tables or to install libraries. But if you do this, 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: Fat controllers and fat views.** Because controller actions are the first place in your app's code that are called when a user request arrives, it's remarkably easy for the actions' methods to get fat—putting all kinds of logic in the controller that really belongs in the model. 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 7. 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 2.2.1 and Screencast 2.2.1) at the end of each request and unpacked when the cookie is received with the next request, and the HTTP specification limits cookies to 4 KBytes in size. Second, although you can change Rails' configuration to allow larger session objects by storing them in their own database table instead of a cookie, bulky sessions are a warning that your app's actions aren't very self-contained. That would mean your app isn't very 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.



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 7 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.

One simple example we can show immediately relates to Service-Oriented Architecture, an important concept introduced in Chapter 1 and to which we'll return often. If we intended Rotten Potatoes to be used in an SOA, its RESTful actions might be performed either by a human who expects to see a Web page as a result of the action or by another service that expects (for example) an XML response. To simplify the task of making your app work with SOA, you can return different formats for the same resource using the `respond_to` method of `ActionController` (not to be confused with Ruby's built-in `respond_to?` introduced in Section 3.2). `ActionController::MimeResponds#respond_to` yields an object that can be used to select the format in which to render a response. Here's how the `update` action can be immediately converted into an SOA-friendly RESTful API that updates a movie's attributes and returns an XML representation of the updated object, while preserving the existing user interface for human users:

<http://pastebin.com/yZxnnWFZ>

```
1  def update
2      @movie = Movie.find params[:id]
3      @movie.update_attributes!(params[:movie])
4      respond_to do |client_wants|
5
6          client_wants.html { redirect_to movie_path(@movie) } # as before
7
8      client_wants.xml { render :xml => @movie.to_xml }
9      end
10 end
```

Similarly, the only reason `new` requires its own controller action is that the human user needs an opportunity to fill in the values that will be used for `create`. Another service would never call the `new` action at all. Nor would it make sense to redirect back to the list of movies after a `create` action: the `create` method could just return an XML representation of the created object, or even just the created object's ID.



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.

4.11 To Learn More

- The online documentation for [Rails](#) gives details on the language, its classes, and the Rails framework.
- The Ruby Way ([Fulton 2006](#)) and The Rails 3 Way ([Fernandez 2010](#)) go into great depth on Ruby and Rails advanced features and wizardry.
- [PeepCode](#) publishes high-quality screencasts covering almost every tool and technique in the Rails ecosystem for a very reasonable price (in the authors' opinion). The two-part [Meet Rails 3](#) screencast is a particularly good complement to the information in this chapter.
- Before writing new code for any functionality that isn't specific to your app, check [rubygems](#) and [rubyforge](#) (at least) to see if someone has created a gem that does most of what you need. As we saw in this chapter, using a gem is as easy as adding a line to your Gemfile and re-running bundle install.

O. Fernandez. *Rails 3 Way, The (2nd Edition)* (Addison-Wesley Professional Ruby Series). Addison-Wesley Professional, 2010. ISBN 0321601661. URL <http://www.amazon.com/Rails-Way-Addison-Wesley-Professional-Ruby/dp/0321601661>.

H. Fulton. *The Ruby Way, Second Edition: Solutions and Techniques in Ruby Programming (2nd Edition)*. Addison-Wesley Professional, 2006. ISBN 0672328844. URL <http://www.amazon.com/Ruby-Way-Second-Techniques-Programming/dp/0672328844>.

Unless otherwise indicated, these suggested projects are based on the myrottenpotatoes app you created in this chapter.

Project 4.1. Modify the app's routes so that visiting `http://localhost:3000` takes you to the list of movies, rather than the generic Rails welcome page. (Hint: consult the [ActionDispatch::Routing](#) documentation.)

Project 4.2.

Add a default banner to the main application layout that will appear on every page of Rotten Potatoes. It should display "Rotten Potatoes" in large red letters, but no visual styling information should go into the template itself. (Hint: pick an element type that reflects the role of this banner, assign it a unique ID, and modify the CSS style file to style the element.) Make it so that clicking on the banner always takes you to RP homepage.

Project 4.3. Instead of redirecting to the Index action after a successful `create`, redirect to the `show` action for the new movie that was just created. Hint: you can use the `movie_path` URI helper but you'll need to supply an argument identifying which movie. To obtain the `this` argument, recall that `Movie.create` if successful returns the newly-created object in addition to creating it.

Project 4.4. Modify the listing of movies as follows. Each modification task will require making a change at a different layer of abstraction:

Modify the Index view to include a row number for each row in the movies table. HINT: look up the documentation of the `each_with_index` function used in line 11 of the view template.

Modify the Index view so that hovering a mouse over a row in the movies table causes the row to temporarily assume a yellow background. HINT: look up the `hover pseudo-class` supported by CSS.

Modify the Index controller action to return the movies ordered alphabetically by title, rather than by release date. HINT: *Don't* try to sort the result of the controller's call to the database. RDBMS's provide ways to specify the order in which a list of results is delivered, and because of Active Record's tight coupling to the underlying RDBMS, the Rails ActiveRecord library's `find` and `all` methods provide a way to ask the underlying RDBMS to do this.

Pretend you didn't have the tight coupling of Active Record, and so you could not assume the underlying storage system can return collection items in any particular order. Modify the Index controller action to return the movies ordered alphabetically by title. HINT: Look up the `sort` method in Ruby's `Enumerable` module.

Project 4.5. What if the user changes his mind before submitting a Create or Update form and decides not to proceed after all? Add a "Cancel" link to the form that just takes the user back to the list of movies.

Project 4.6. Modify the "Cancel" link so that if it's clicked as part of a Create flow, the user is taken back to the list of movies, but if clicked as part of an Update flow, the user is taken back to the Show template (view) for the movie he began to edit. Hint: the instance method `ActiveRecord::Base#new_record?` returns true if its receiver is a new model object, that is, one that has never been saved in the database. Such objects won't have ID's.

Project 4.7. The dropdown menus for Release Date don't allow adding movies released earlier than 2006. Modify it to allow movies as early as 1930. (Hint: check the [documentation](#) for the `date_select` helper used in the form.)

Project 4.8. The `description` field of a movie was created as part of the initial migration, but so far isn't displayed and cannot be edited. Make the necessary changes so that the description is visible in the Show view and editable in the New and Edit views. Hint: you should only need to change two files.

Project 4.9. Our current controller methods aren't very robust: if the user manually enters a URI to Show a movie that doesn't exist (for example `/movies/99999`), she will see an ugly exception message. Modify the `show` method in the controller so that if the requested movie doesn't exist, the user is redirected to the Index view with a friendly message explaining that no movie with the given ID could be found. (Hint: you can either use `find_by_id` and check the result, or you can continue to use `find` but place it in a `begin...rescue...end` and rescue from

`ActiveRecord::RecordNotFound`. Opinions differ as to which is more idiomatic; normally, exceptions are used to indicate unexpected events, but it's not obvious

whether such an event is truly unexpected or simply rare.)

Project 4.10. Putting it all together exercise: Write and deploy a Rails app that scrapes some information from a Web page using Nokogiri's XPath features, and turns it into an RSS feed using Builder. Verify that you can subscribe to the RSS feed in your browser or RSS news reader.



Niklaus Wirth (1934–) received the Turing Award in 1984 for developing a sequence of innovative programming languages, 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*

- 5.1 [Introduction to Behavior-Driven Design and User Stories](#)
- 5.2 [SMART User Stories](#)
- 5.3 [Introducing Cucumber and Capybara](#)
- 5.4 [Running Cucumber and Capybara](#)
- 5.5 [Lo-Fi User Interface Sketches and Storyboards](#)
- 5.6 [Enhancing Rotten Potatoes](#)
- 5.7 [Explicit vs. Implicit and Imperative vs. Declarative Scenarios](#)
- 5.8 [Fallacies and Pitfalls](#)
- 5.9 [Concluding Remarks: Pros and Cons of BDD](#)
- 5.10 [To Learn More](#)
- 5.11 [Suggested Projects](#)

The first step in the Agile cycle, and often the most difficult, is a dialogue with each of the stakeholders to understand the requirements. We first derive [user stories](#), which are short narratives each describing a specific interaction between some stakeholder and the application. The [Cucumber](#) tool turns these stylized but informal English narratives into acceptance and integration tests. As SaaS usually involves end-users, we also need a user interface. We do this with *low-fidelity (Lo-Fi)* drawings of the Web pages and combine them into [storyboards](#) before creating the UI in HTML.

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.

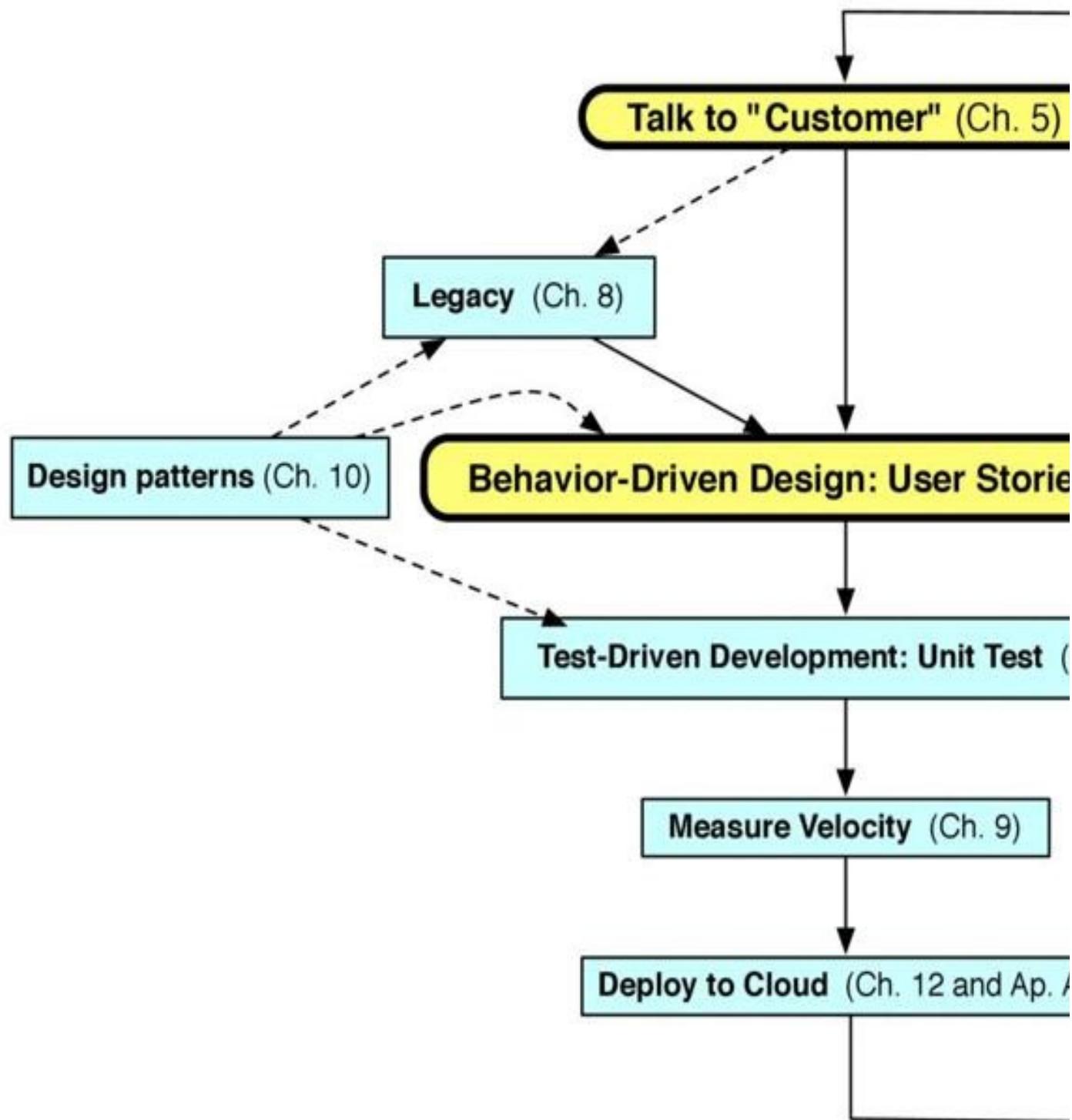


Figure 5.1: An iteration of the Agile software lifecycle and its relationship to the chapters in this book. This chapter emphasizes talking to customers as part of Behavior-Driven Design.

The Agile lifecycle was invented to attack these problems for many common types of software. Figure 5.1 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 ***Big Design Up Front (BDUF)*** 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 BDUF, but unlike BDUF, 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 BDUF, you start with requirements, but in BDD user stories take the place of design documents in BDUF.

By concentrating on the *behavior* of the application versus the *implementation* of application, 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 (76 mm by 127 mm) index cards, known as "3-by-5 cards." (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 5.2 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 Rotten Potatoes app from Chapters [2](#) and [3](#) as the running example in this chapter and the next one. We start with the stakeholders, which are simple for this simple app:

- The operators of Rotten Potatoes, and
- The movie fans who are end-users of Rotten Potatoes.

We'll introduce a new feature in Section [5.6](#), but to help understand all the moving parts, we'll start with a user story for an existing feature of Rotten Potatoes 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 Rotten Potatoes database:

Pastebin is a service for copying-and-pasting book code. (You need to type URI if you're reading the print book; it's a link in ebooks.)

<http://pastebin.com/4zkRbaPT>

```
1  Feature: Add a movie to Rotten Potatoes
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 Rotten Potatoes database
```

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:

<http://pastebin.com/M2ysXHE6>

```
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 to link 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 are not always 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 nontechnical stakeholders to help create requirements.
- **3x5 cards**, each with a user story of one to three sentences, are an easy 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.

Self-Check 5.1.1. True or False: User stories on 3x5 cards in BDD play the same role as design requirements in BDUF.

 True.

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 UML-like diagram (see Chapter 10) 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.

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 are examples of a vague feature paired with a specific version: <http://pastebin.com/JJFhUDC5>

-
- 1 Feature: User can search for a movie (vague)
 - 2 Feature: User can search for a movie by title (specific)
 - 3

-
- *Measurable.* Adding Measurable to Specific means that each story should be testable, which implies that there are known expected results for some good inputs. An example of a pair of an unmeasurable versus measurable feature is <http://pastebin.com/pRbu6iT>

- 1 Feature: Rotten Potatoes should have good response time (unmeasurable)
- 2 Feature: When adding a movie, 99% of Add Movie pages should appear within 3 seconds (measurable)
- 3
- 4

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. Chapter 9 describes the tool **Pivotal Tracker** that 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:

Why add the Facebook feature? As box office manager, I think more people will go with friends and enjoy the show more.

Why does it matter if they enjoy the show more? I think we will sell more tickets.

Why do you want to sell more tickets? Because then the theater makes more money.

Why does theater want to make more money? We want to make more money so that we don’t go out of business.

Why does it matter that 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.

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 5.2.1. Which SMART guideline(s) does the feature below violate?

<http://pastebin.com/t8Lh90Uy>

1

Feature: Rotten Potatoes 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 5.2.2. Rewrite this feature to make it SMART.

<http://pastebin.com/gTugmNzn>

1 Feature: I want to see a sorted list of movies sold.

Here is one SMART revision of this user story:

<http://pastebin.com/6NNUTcY4>

```
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.
```

Remarkably enough, the tool **Cucumber** turns customer-understandable user stories into **acceptance tests**, which ensure the customer is satisfied, and **integration tests**, which 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 freeform. This section explains how Cucumber accomplishes this minor miracle.

In the Cucumber context we will use the term **user story** to refer 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 3 to 8 **steps**.

Figure 5.2 is an example user story, showing a feature with one scenario of adding the movie *Men In Black*; the scenario has eight 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 and help develop, which is a founding principle of Agile and BDD.

Cucumber keywords **Given**, **When**, **Then** **And**, and **But** have different names just for benefit of human readers, but they are all aliases to the same method. Thus, you don't have to remember the syntax for many different keywords.

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 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**.

<http://pastebin.com/mqQncg1s>

```
1 Feature: User can manually add movie
2
3 Scenario: Add a movie
4   Given I am on the RottenPotatoes home page
5   When I follow "Add new movie"
6   Then I should be on the Create New Movie page
7   When I fill in "Title" with "Men In Black"
8   And I select "PG-13" from "Rating"
9   And I press "Save Changes"
10  Then I should be on the RottenPotatoes home page
11  And I should see "Men In Black"
```

Figure 5.2: A Cucumber scenario associated with the adding a movie feature for Rotten Potatoes.

A separate set of files defines the Ruby code that tests these steps. These are called **step definitions**. Generally, many steps map onto a single step definition.

How does Cucumber match the steps of the scenarios with the **step definitions** that perform these tests? The trick is that Cucumber uses regular expressions or **regexes** (Chapter 3) to match the English phrases in the steps of the scenarios to the step definitions of the testing harness.

For example, below is a string from a step definition in the scenario for Rotten Potatoes:

Given /^(?:|I)am on (.+)/

This regex can match the text “I am on the Rotten Potatoes home page” on line 4 of Figure 5.2. The regex also captures the string after the phrase “am on ” until the end of the line (“the Rotten Potatoes home page”). The body of the step definition contains Ruby code that tests that the step, likely using captured strings such as the one above.

Thus, a way to think of the relationship between step definitions and steps is that step definitions are like method definitions, and the steps of the scenarios are like method calls.

We then need a tool that will act as a user and pretend to use the feature under different scenarios. The part of Cucumber that “pretends to be a user” (taking actions in a web browser like pressing a button) is called **Capybara**. By default, it “simulates” many aspects of a browser—it can interact with the app to receive

pages, parse the HTML, and submit forms as a user would.



Summary of Cucumber Introduction

- Cucumber combines a **feature** that you want to add with a set of **scenarios**. We call this combination a **user story**.
- The steps of the scenarios use the keyword **Given** to identify the current state, **When** to identify the action, and **Then** to identify the consequence of the action.
- The scenario steps also use the keywords **And** and **But** to act as conjunctions to make more complex descriptions of state, action, and consequences.
- **Cucumber** matches **steps** to **step definitions** using **regular expressions**.
- **Capybara** puts the SaaS application through its paces by simulating a user and browser performing the steps of the scenarios.
- By storing **features** in files along with different **scenarios** of feature use composed of many **steps**, and storing Ruby code in separate files containing **step definitions** that tests each type of step, the Rails tools **Cucumber** and **Capybara** automatically test the behavior of the SaaS app.

Self-Check 5.3.1. True or False: Cucumber matches scenario steps to step definitions using regexes and Capybara pretends to be a user that interacts with the SaaS application according to these scenarios.

 True.

ELABORATION: Stubbing the web

Alas, Capybara is not smart enough to do JavaScript, which we'll meet in Chapter [7](#). Fortunately, with appropriate options, Capybara is also able to interface with Webdriver, which actually fires up a REAL browser and "remote controls" it to make it do what the stories say. For this chapter, we won't enable that mode because it is much slower than the "simulated" mode, and the "simulated" mode is appropriate for everything except testing JavaScript. Figure [5.11](#) towards the end of the chapter shows the relationship between these tools.

Cucumbers are green The test-passing green color of the cucumber plant gives this tool its name.

A major benefit of user stories in Cucumber is **Red-Yellow-Green analysis**. Once a user story is written, we can try to run it immediately. In the beginning, steps may initially be highlighted either in Red (for failing) or Yellow (not yet implemented). Our goal is to take each step and go from Yellow or Red to Green (for passing), by incrementally adding what's needed to make it pass. In some cases, this is really easy. In the next chapter we similarly try to go from Red to Green at the level of

unit tests. Recall that unit tests are for individual methods whereas Cucumber scenarios test entire paths through the app and thus can be acceptance tests or integration tests.

 Like other useful tools we've seen, Cucumber is supplied as a Ruby gem, so the first thing we need to do is declare that our app depends on this gem and use Bundler to install it. Building on the myrottenpotatoes app you started in Chapter 3, add the following lines to Gemfile; we've indicated that Cucumber and its related gems are only needed in the test and development environments and not the production environment (Section 4.2 introduced the three environments in which Rails apps can run).

<http://pastebin.com/MMZdRmB3>

```
1  # add to end of Gemfile
2  group :test, :development do
3    gem 'cucumber-rails'
4    gem 'cucumber-rails-training-wheels' # some pre-
fabbed step definitions
5
6    gem 'database_cleaner' # to clear Cucumber's test database b
etween runs
7
8    gem 'capybara'          # lets Cucumber pretend to be a web b
rowser
9
10   gem 'launchy'           # a useful debugging aid for user sto
ries
11
12 end
```



Once you've modified Gemfile, run `bundle install --without production`. If all goes well, you'll eventually see "Your bundle is complete." We now have to set up the directories and "boilerplate" files that Cucumber and Capybara need. Like Rails itself, Cucumber comes with a *generator* that does this for you. In the app's root directory, run the following two commands (if they ask whether it's OK to overwrite certain files such as `cucumber.rake`, you can safely say yes):

```
rails generate cucumber:install capybara
rails generate cucumber_rails_training_wheels:install
```

The Cucumber generator gives you commonly used step definitions as a starting point, such as interactions with a web browser. For this app, you will find them in `myrottenpotatoes/features/step_definitions/web_steps.rb`. In addition to these predefined steps, you'll need to create new step definition to match the unique functionality of your app. You will probably want to learn most common predefined step definitions and use them when you write your features so that you can write fewer step definitions.



Before trying to run Cucumber, there's one more step we must take: you must initialize the test database by running `rake db:prepare`. You need to do this before the first time you run tests or whenever the database schema is changed. Section [4.2 in Chapter 3](#) provides a more detailed description.

At this point you're ready to start using Cucumber. You add the features themselves in the `features` directory as files with a `.feature` file extension. Copy the user story in [Figure 5.2](#) and paste it into a file called `AddMovie.feature` in the `features` directory. To see how scenarios and the step definitions interact and how they change color like maple trees in New England when the seasons change, type `cucumber features/AddMovie.feature`. Watch the screencast to see what to do next.

Screencast 5.4.1: [Cucumber Part I](#)

The screencast shows how Cucumber checks to see whether the tests work by coloring the step definitions. Failing steps are red, unimplemented steps are yellow, and passing steps are green. The first step on line 4 is red, so Cucumber skips the rest. It fails because there is no path in `paths.rb` that matches "the Rotten Potatoes home page", as the Cucumber error message explains. The message even suggests how to fix the failure by adding such a path to `paths.rb`. This new path turns this first step as green as a cucumber, but now the third step on line 6 is red. As error message explains, it fails because no path matches "Create New Movie page", and we fix it again by adding the path to `paths.rb`. All steps now are as cool as a cucumber, and the `AddMovie` scenario passes.

Summary: To add features as part of BDD, we need to define acceptance criteria first. Cucumber enables both capturing requirements as user stories and getting integration and acceptance test out of that story. Moreover, we get automatically runnable tests so that we'll have regression tests to help maintain the code as we evolve it further. (We'll see this approach again in Chapter [8](#) with a much larger application than Rotten Potatoes.)

Self-Check 5.4.1. Cucumber colors steps green that pass the test. What is the

difference between steps colored yellow and red?

 Yellow steps have not yet been implemented while red steps have implemented but fail the test.

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 to have a mockup of a page that has the login. Alas, building software prototypes of UI can intimidate stakeholders from suggesting improvements to the UI. That is, software prototypes have just the opposite characteristics of what we need at this early point of the design.

ROTTEN POTATOES!

CREATE NEW MOVIE

MOVIE TITLE

MOVIE RATING

RELEASE DATE

MOVIE DESCRIPTION

SAVE CHANGES

Figure 5.3: Window that appears when adding a movie to Rotten Potatoes.

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 UI](#) and the paper prototypes ***sketches***. For example, Figure 5.3 shows a Lo-Fi sketch of the UI for adding a movie to Rotten Potatoes.

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.

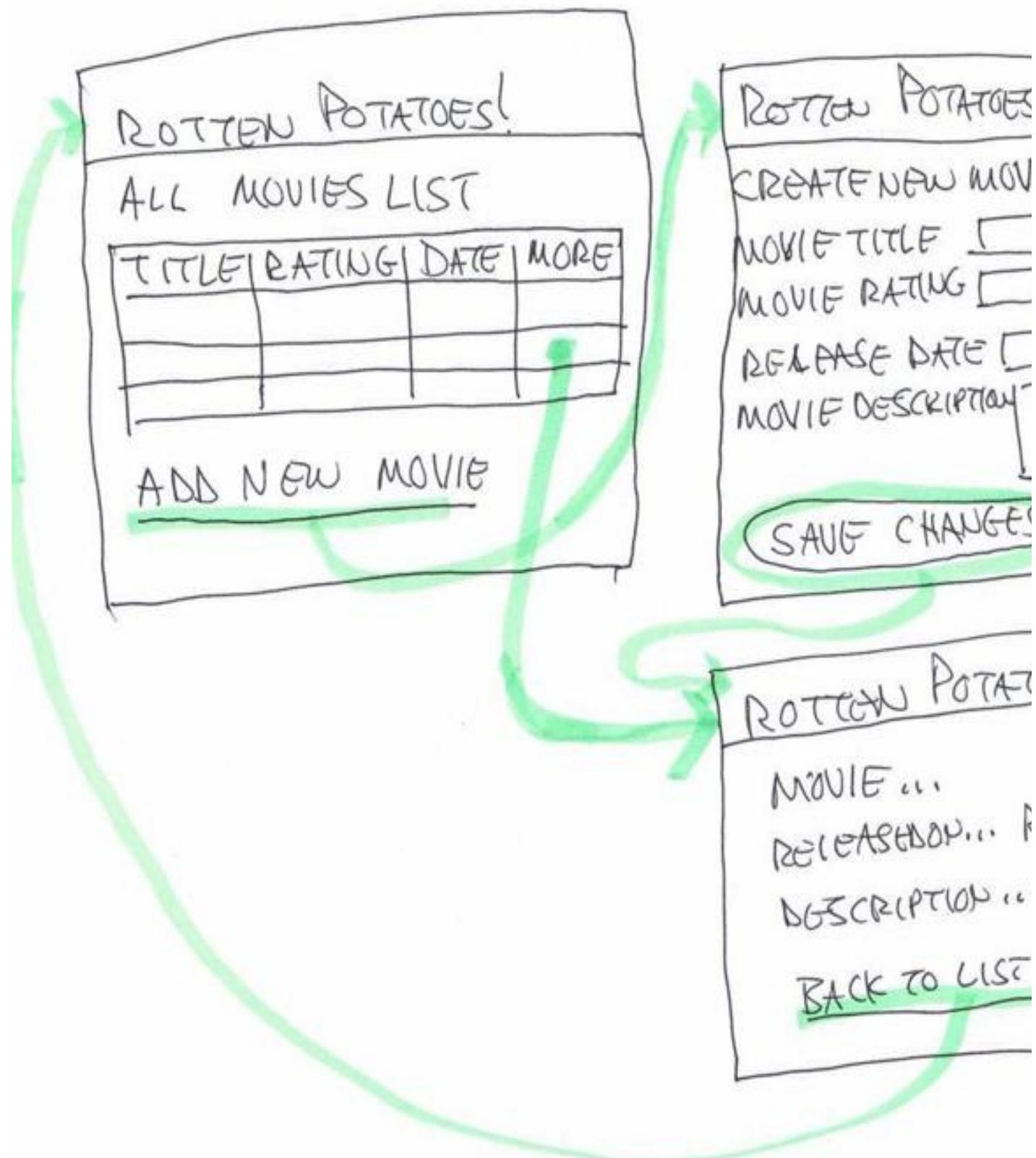


Figure 5.4: Storyboard of images for adding a movie to Rotten Potatoes.

Lo-Fi sketches show what the UI looks like at one instant of 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 like 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.

For a storyboard, you want to think about all the user interactions with a web app:

- Pages or sections of pages,
- Forms and buttons, and
- Popups.

Figure 5.4 shows a sequence of Lo-Fi sketches with 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 2 showed how Haml 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 your sketch. Remember that the common parts of the page layout—banners, structural divs, and so on—can go into `views/layouts/application.html.haml`.

Start the process by looking at the Lo-Fi UI sketches and split them into “blocks” of the layout. Use CSS 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.

Since the example in Section 5.3 involved existing functionality, there is no need to modify the Haml or CSS. The next section adds a new feature to Rotten Potatoes and thus needs Haml changes.

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 media before using Haml and CSS to create the pages you want in HTML.

Self-Check 5.5.1. True or False: The purpose of the Lo-Fi UI and storyboards is to debug the UI before you program it.

 True.

As a second example of user stories and Lo-Fi UIs, suppose we want to search The Open Movie Database (TMDb) to find information about a movie we are interested in adding to Rotten Potatoes. As we'll see in Chapter [6](#), TMDb has an API (application programming interface) designed to allow its information to be accessed in a Service-Oriented Architecture.

In this chapter, we use Cucumber to develop two scenarios and the corresponding Lo-Fi UI sketches to show how we would like Rotten Potatoes to integrate with TMDb, and we'll get one of the scenarios to go green by temporarily "faking out" some of the code. In Chapter [6](#), we'll write the code needed to get the other scenario to go green. Getting the first couple of scenarios working can seem tedious, because you usually have to add a lot of infrastructure, but it goes much faster after that, and in fact you will even be able to re-use your step definitions to create higher-level "declarative" steps, as we will see in Section [5.7](#).

ROTTEN POTATOES!

ALL MOVIES LIST

TITLE	RATING	DATE	MORE
:	:	:	:
:	:	:	:

ADD NEW MOVIE

ADD FROM TMDB

SEARCH

MATCH

NO MATCH

ROTTEN POTATOES!

ADD NEW MOVIE

TITLE [INCEPTION]

RATING [PG-13]

DESC

mm
mm
mm

ADD

CANCEL

ROTTEN POTATOES!

SORRY, NO MATCH FOR "..."

ALL MOVIES LIST

TITLE	RATING	DATE
:	:	
:	:	

ADD NEW MOVIE
ADD FROM T...

Figure 5.5: Storyboard of UI for searching The Movie Database.

The storyboard in Figure 5.5 shows how we envision the feature working. The home page of Rotten Potatoes, which lists all movies, will be augmented with a search box where we can type some title keywords of a movie and a "Search" button that will search TMDb for a movie whose title contains those keywords. If the search does match—the so-called "happy path" of execution—the first movie that matches will be used to "pre-populate" the fields in the Add New Movie page that we already developed in Chapter 3. (In a real app, you'd want to create a separate page showing all matches and letting the user pick one, but we're deliberately keeping the example simple.) If the search doesn't match any movies—the "sad path"—we should be returned to the home page with a message informing us of this fact.

<http://pastebin.com/Qc4rppVL>

```
1 Feature: User can add movie by searching for it in The Movie
Database (TMDb)
2
3 As a movie fan
4 So that I can add new movies without manual tedium
5 I want to add movies by looking up their details in TMDb
6
7 Scenario: Try to add nonexistent movie (sad path)
8
9 Given I am on the RottenPotatoes home page
10 Then I should see "Search TMDb for a movie"
11 When I fill in "Search Terms" with "Movie That Does Not Ex
ist"
12 And I press "Search TMDb"
13 Then I should be on the RottenPotatoes home page
14 And I should see "'Movie That Does Not Exist' was not foun
d in TMDb."
```

Figure 5.6: A sad path scenario associated with adding a feature to search The Movie Database.

Normally you'd complete the happy path first, and when you reach a failing or pending step that requires writing *new* code, you do so via Test Driven Development

(TDD). We'll do that in Chapter [6](#) by writing code that really calls TMDb and integrating it back into this scenario. For now, we'll start with the sad path to illustrate Cucumber features and the BDD process. Figure [5.6](#) shows the sad path scenario for the new feature; create a file `features/search_tmdb.feature` containing this code. When we run the feature with cucumber `features/search_tmdb.feature`, the second step *Then I should see "Search TMDb for a movie"* should fail (red), because we haven't yet added this text to the home page `app/views/movies/index.html.haml`. So our first task is to get this step to go green by making that change.

Technically, a "pure" BDD approach could be to get this step to pass just by adding the text *Search TMDb for a movie* anywhere in that view, and then re-running the scenario. But of course we know that the very next step *When I fill in "Search Terms" with "Movie That Does Not Exist"* will also fail, because we haven't added a form field called "Search Terms" to the view either. So in the interest of efficiency, modify

`index.html.haml` by adding the lines in Figure [5.7](#), which we now explain. 

<http://pastebin.com/viZTPxGN>

```
1 -# add to end of app/views/movies/index.html.haml:
2
3 %h1 Search TMDb for a movie
4
5 = form_tag :action => 'search_tmdb' do
6
7   %label{:for => 'search_terms'} Search Terms
8   = text_field_tag 'search_terms'
9   = submit_tag 'Search TMDb'
```

Figure 5.7: The Haml code for the Search TMDb page.

Line 3 is the text that allows *Then I should see "Search TMDb for a movie"* to pass. The remaining lines create the fill-in form; we introduced these in Chapter [3](#), so some of this markup should be familiar. Two things are worth noting. First, as with any user interaction in a view, we need a controller action that will handle that interaction. In this case the interaction is submitting the form with search keywords. Line 5 says that when the form is submitted, the controller action `search_tmdb` will receive the form submission. That code doesn't exist yet, so we had to choose a descriptive

name for the action.

The second thing to note is the use of the HTML `label` tag. Figure 2.14 in Chapter 2 tells us that lines 7 and 8 will expand to the following HTML markup:

<http://pastebin.com/678dx2gB>

```
1   <label for='search_terms'>Search Terms</label>
2
<input id="search_terms" name="search_terms" type="text" />
```

The key is that the `for` attribute of the `label` tag matches the `id` attribute of the `input` tag, which was determined by the first argument to the `text_field_tag` helper called in line 8 of Figure 5.7. This correspondence allows Cucumber to determine what form field is being referenced by the name “Search Terms” in line 11 of Figure 5.6: *When I fill in “Search Terms”....*

Doing it over and over? `rake cucumber` runs all your features, or more precisely, those selected by the *default profile* in [Cucumber’s configuration file cucumber.yml](#). In the next chapter we’ll meet a tool called autotest that automates re-running tests when you make changes to files.

At this point, re-running `cucumber features/search_tmdb.feature` should show the first three steps passing, but the fourth step *And I press “Search TMDb”* will fail. As you may recall from Section 4.1, we have to make sure there is a route to this controller action. The bottom part of the Figure 5.8 shows the line you must add to `config/routes.rb` to add a form submission (POST) route to that action.

However, even with the new route, this step still will fail with an exception: even though we have a button with the name “Search TMDb”, the `form_tag` specifies that `MoviesController#search_tmdb` is the controller action that should receive the form, yet no such method exists in `movies_controller.rb`. Figure 5.1 says that we should now use Test-Driven Development (TDD) techniques to create that method. But since TDD is the topic of the next chapter, we’re going to cheat a bit in order to get the scenario running. Since this is the “sad path” scenario where no movies are found, we will temporarily create a controller method that *always* behaves as if nothing was found, so we can finish testing the sad path. Also, The top part of Figure 5.8 shows the code you should add to `app/controllers/movies_controller.rb` to create the “fake” hardwired `search_tmdb` action.

<http://pastebin.com/yVNvBKNe>

```
1 # add to movies_controller.rb, anywhere inside
```

```
2 # 'class MoviesController < ApplicationController':  
3  
4 def search_tmdb  
5   # hardwire to simulate failure  
6   flash[:warning] = "'#{params[:search_terms]}' was not found in TMDb."  
7   redirect_to movies_path  
8 end
```

<http://pastebin.com/tCHwFER8>

```
1 # add to routes.rb, just before or just after 'resources :movies' :  
2  
3 # Route that posts 'Search TMDb' form  
4 post '/movies/search_tmdb'
```

Figure 5.8: (Top) This “fake” controller method always behaves as if no matches were found. It retrieves the keywords typed by the user from the `params` hash (as we saw in Chapter 3), stores a message in the `flash[]`, and redirects the user back to the list of movies. Recall from Chapter 3 that we added code to `app/views/layouts/application.html.haml` to display the contents of the flash on every view. (Bottom) A route that triggers this mechanism when a form is POSTed.

If you’re new to BDD, this step might surprise you. Why would we deliberately create a fake controller method that doesn’t actually call TMDb, but just pretends the search failed? In this case, the answer is that it lets us finish the rest of the scenario, making sure that our HTML views match the Lo-Fi sketches and that the sequence of views matches the storyboards. Indeed, once you make the changes in Figure 5.8, the entire sad path should pass. Screencast 5.6.1 summarizes what we’ve done so far.

Screencast 5.6.1: [Cucumber Part II](#)

In this screencast, we do a sad path to illustrate features of Cucumber because it is able to use existing code. The first step on line 5 of Figure 5.6 passes but the step on line 6 fails because we haven’t modified `index.html.haml` to include the name of the new page or to include a form for typing in a movie to search for. We fix this by

adding this form to `index.html.haml`, using the same Rails methods described in Sections [4.4](#) and [4.6](#) of Chapter [3](#). When creating a form, we have to specify which controller action will receive it; we chose the name `search_tmdb` for controller action. (We'll implement this method in the next chapter). Once we have updated `index.html.haml` and named the controller action, Cucumber colors the steps on lines 5 to 7 green. The next step on line 8 fails. Even though we specified the name of the controller action, there is no route that would match an incoming URI to the name. To keep things simple, we will set up a route just for that action in `config/routes.rb`, again using techniques discussed in Section [4.1](#) of Chapter [3](#).

What about the happy path, when we search for an existing movie? Observe that the first two actions on that path—going to the Rotten Potatoes home page and making sure there is a search form there, corresponding to lines 9 and 10 of Figure [5.6](#)—are the same as for the sad path. That should ring a Pavlovian bell in



your head asking how you can DRY out the repetition.

<http://pastebin.com/saRTPLnM>

```
1 Feature: User can add movie by searching for it in The Movie
Database (TMDb)
2
3   As a movie fan
4   So that I can add new movies without manual tedium
5   I want to add movies by looking up their details in TMDb
6
7 Background: Start from the Search form on the home page
8
9   Given I am on the RottenPotatoes home page
10  Then I should see "Search TMDb for a movie"
11
12 Scenario: Try to add nonexistent movie (sad path)
13
14  When I fill in "Search Terms" with "Movie That Does Not Ex
ist"
15  And I press "Search TMDb"
16  Then I should be on the RottenPotatoes home page
17  And I should see "'Movie That Does Not Exist' was not foun
d in TMDb."
```

```
18
19 Scenario: Try to add existing movie (happy path)
20
21 When I fill in "Search Terms" with "Inception"
22 And I press "Search TMDb"
23 Then I should be on the "Search Results" page
24 And I should not see "not found"
25 And I should see "Inception"
```

Figure 5.9: DRYing out the common steps between the happy and sad paths using the `Background` keyword, which groups steps that should be performed before *each* scenario in a feature file.

Figure 5.9 shows the answer. Modify `features/search_tmdb.feature` to match the figure and once again run `cucumber features/search_tmdb.feature`. Unsurprisingly, the step at line 24 will fail, because we have hardwired the controller method to pretend there is never a match in TMDb. At this point we could change the controller method to hardwire success and make the happy path green, but besides the fact that this would cause the sad path to go red, in the next chapter we will see a better way. In particular, we'll develop the *real* controller action using Test-Driven Development (TDD) techniques that "cheat" to set up the inputs and the state of the world to test particular conditions in isolation. Once you learn both BDD and TDD, you'll see that you commonly iterate between these two levels as part of normal software development.

Summary:

- Adding a new feature for a SaaS app normally means you specify a UI for the feature, write new step definitions, and perhaps even write new methods before Cucumber can successfully color steps green.
- Usually, you'd write and complete scenarios for the happy path(s) first; we began with the sad path only because it allowed us to better illustrate some Cucumber features.
- The `Background` keyword can be used to DRY out common steps across related scenarios in a single feature file.
- Usually, system-level tests such as Cucumber scenarios shouldn't "cheat" by hard-wiring fake behavior in methods. BDD and Cucumber are about behavior, not implementation, so we would instead use other techniques such as TDD (which the next chapter describes) to write the actual methods to make all

scenarios pass.

Self-Check 5.6.1. True or False: You need to implement all the code being tested before Cucumber will say that the test passes.

False. A sad path can pass without having the code written need to make a happy path pass.

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 BDUF 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 Rotten Potatoes 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 5.2 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. 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:

<http://pastebin.com/JQkEUVDJ>

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     When I follow "Movie Title"
19     Then I should see "Apocalypse Now" before "Zorro"
```

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.

A declarative version of the above scenario might look like this:

<http://pastebin.com/JZsT3ddv>

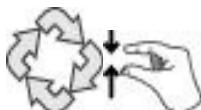
```
1  Feature: movies when added should appear in movie list
2
3
4  Scenario: view movie list after adding movie (declarative and
5      DRY)
6      Given I have added "Zorro" with rating "PG-13"
7      And I have added "Apocalypse Now" with rating "R"
8
9      And I am on the RottenPotatoes home page sorted by title
```

8 Then I should see "Apocalypse Now" before "Zorro"

<http://pastebin.com/rky95ycT>

```
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   step "I am on #{path}"
12   regexp = /#{string1}.*#{string2}/m # /m means match across
newlines
13   page.body.should =~ regexp
14 end
```

Figure 5.10: Adding this code to `movie_steps.rb` creates new step definitions matching lines 5 and 6 of the declarative scenario by reusing your existing steps. `steps` (line 2) reuses a sequence of steps and `step` (line 11) reuses a single step. Recall from Figure 3.1 that `%Q` is an alternative syntax for double-quoting a string, and that `Given`, `When`, `Then` and so on are synonyms provided for readability. (We will learn about `should`, which appears in line 13, in the next chapter.)



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 5.10 shows, you can *reuse* your 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, DSL-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 by making via your step definitions, and they 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.

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 RailsCasts](#) 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 wiki](#) 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 5.7.1. True or False: Explicit requirements are usually defined with imperative scenarios and implicit requirements are usually defined with declarative scenarios.

➊ False. These are two independent classifications; both requirements can use either type of scenarios.

 **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 but 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: 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: 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*. There are many, many outputs that don’t match, so that is not likely to be a good test. Always include positive expectations such as *Then I should see...* to check results.

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

Figure 5.11 shows the relationship of the testing tools introduced in this chapter to the testing tools in the following chapters. Cucumber allows writing user stories as features, scenarios, and steps and matches these steps to step definitions using regular expressions. The step definitions invoke methods in Cucumber and Capybara. We need Capybara because we are writing a SaaS application, and testing requires a tool to act as the user and web browser. If the app was not for SaaS, then we could invoke the methods that test the app directly in Cucumber.

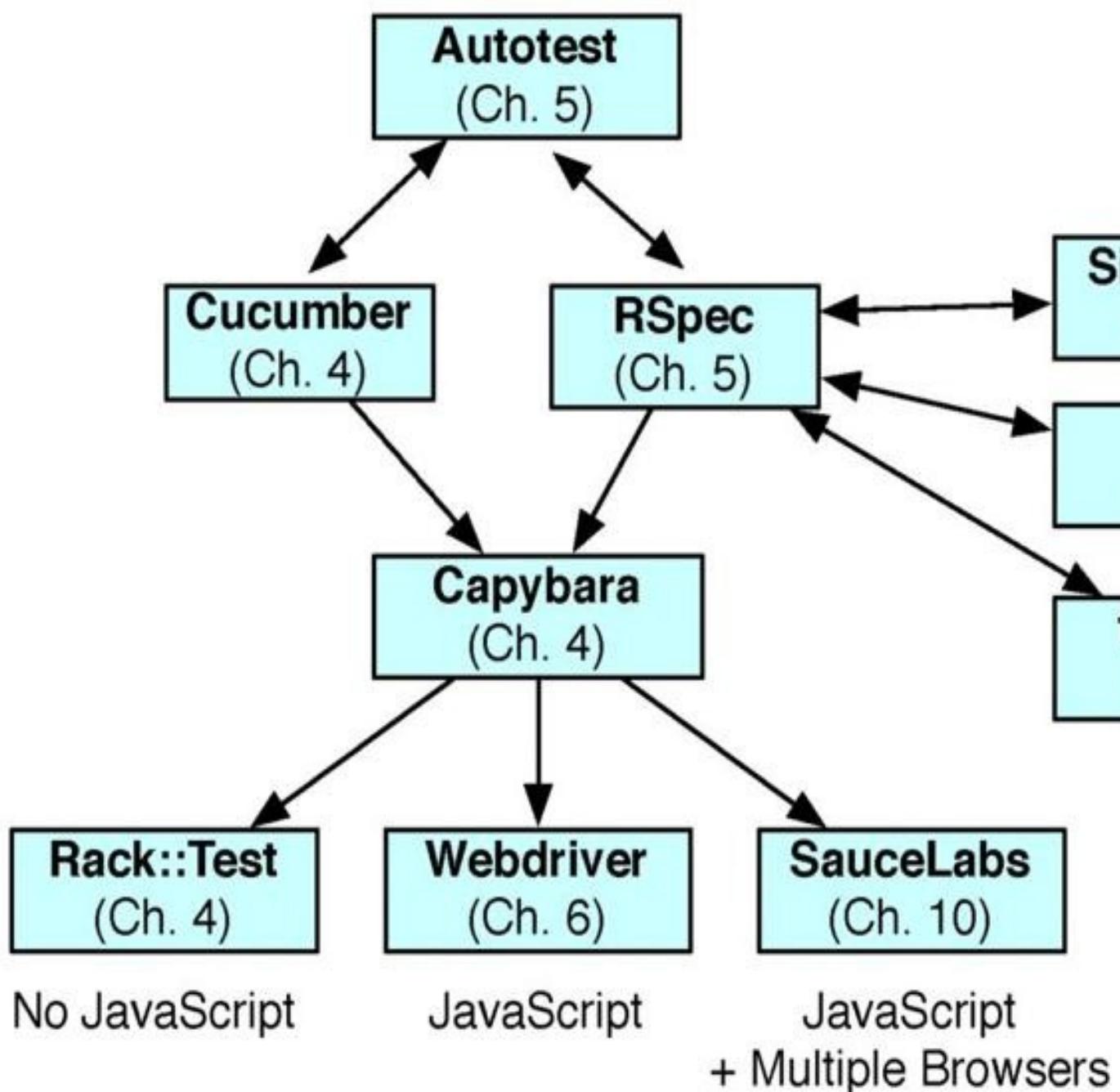


Figure 5.11: The relationship of Cucumber, RSpec, Capybara, and the many other testing tools and services described in this book. This chapter uses Rack::Test, since our application does not yet use JavaScript. If it did, we'd have to use the slower but more complete Webdriver. Chapter 12 shows how we can replace Webdriver with the service from SauceLabs to test your app with many browsers instead of just one.

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 makes 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. Figure [5.12](#) is a poster used at Google that embraces the virtues of testing over debugging.



Figure 5.12: Google places these posters inside restrooms to remind developers of the importance of testing.

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 BDUF 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 10 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 8) 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 wiki](#) 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.

5.10 To Learn More

- The [Cucumber wiki](#) has links to documentation, tutorials, examples, screencasts, best practices, and lots more on Cucumber.
- *The Cucumber Book* ([Wynne and Hellesoy 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 uses Cucumber such as testing RESTful service automation.
- [Ben Mabey](#) (a core Cucumber developer) and [Jonas Nicklas](#), among others, have written eloquently about the benefits of declarative vs. imperative Cucumber scenarios. In fact, the main author of Cucumber, Aslak Hellesøy, deliberately [removed web_steps.rb](#) (which we met in Section 5.4) from Cucumber in October 2011, which is why we had to separately install the `cucumber_rails_training_wheels` gem to get it for our examples.

M. Wynne and A. Hellesoy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012. ISBN 1934356808. URL <http://www.amazon.com/Cucumber-Book-Behaviour-Driven-Development-Developers/dp/1934356808>.

Project 5.1. Suppose in RottenPotatoes, instead of dials to pick the rating and pick the release date, the choice was instead fill in the blank form. First, make the appropriate changes to the scenario in Figure 5.2. List the step definitions from `features/cucumber/web_steps.rb` that Cucumber would now invoke in testing these new steps.

Project 5.2. Add a sad path scenario to the feature in Figure 5.2 of what happens when a user leaves the title field empty.

Project 5.3. Write down a list of background steps that will populate the RottenPotatoes site with a few movies.

Project 5.4. Create a lo-fi mockup showing the current behavior of the RottenPotatoes app.

Project 5.5. Come up with a feature that you would like to add to RottenPotatoes, and draw a storyboards showing how it would be implemented and used.

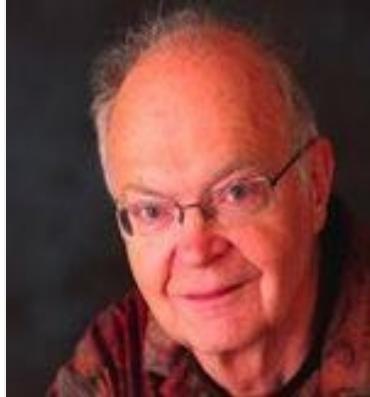
Project 5.6. Create a list of steps such as those in Figure 5.10 that would be used to implement the step:

<http://pastebin.com/6xBUe1EC>

1 When / I delete the movie: "(.*)"/ do |title|

Project 5.7. Use Cucumber and Mechanize to create integration tests or acceptance tests for an existing SaaS application that has no testing harness.

Project 5.8. TBD: exercise that gets you to realize you can store implicit cross-step state in an instance variable of World. For example, generating a report and then referring to “the report” in the step def.



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*. Many consider this series the definitive reference on analysis of algorithms; “bounty checks” from Knuth for finding errors in his books are among the most prized trophies among computer scientists. Knuth also invented the widely-used TeX typesetting system, with which this book was prepared.

One of the most important lessons, perhaps, is the fact that SOFTWARE IS HARD. ...TeX and METAFONT proved to be much more difficult than all the other things I had done (like proving theorems or writing books). The creation of good software demands a significantly higher standard of accuracy than those other things do, and it requires a longer attention span than other intellectual tasks. *Donald Knuth, Keynote address to 11th World Computer Congress, 1989*

- 6.1 [Background: A RESTful API and a Ruby Gem](#)
- 6.2 [FIRST, TDD, and Getting Started With RSpec](#)
- 6.3 [The TDD Cycle: Red–Green–Refactor](#)
- 6.4 [More Controller Specs and Refactoring](#)
- 6.5 [Fixtures and Factories](#)
- 6.6 [TDD for the Model](#)
- 6.7 [Stubbing the Internet](#)
- 6.8 [Coverage Concepts and Unit vs. Integration Tests](#)
- 6.9 [Other Testing Approaches and Terminology](#)
- 6.10 [Fallacies and Pitfalls](#)
- 6.11 [Concluding Remarks: TDD vs. Conventional Debugging](#)
- 6.12 [To Learn More](#)
- 6.13 [Suggested Projects](#)

In test-driven development, you first write failing tests for a small amount of nonexistent code and then fill in the code needed to make them pass, and look for opportunities to refactor (improve the code's structure) before going on to

the next test case. This cycle is sometimes called Red–Green–Refactor, since many testing tools print failed test results in red and passing results in green. To keep tests small and isolate them from the behavior of other classes, we introduce mock objects and stubs as examples of *seams*—places where you can change the behavior of your program at testing time without changing the source code itself.

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 [6.1](#) 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 good code, and vice versa*.

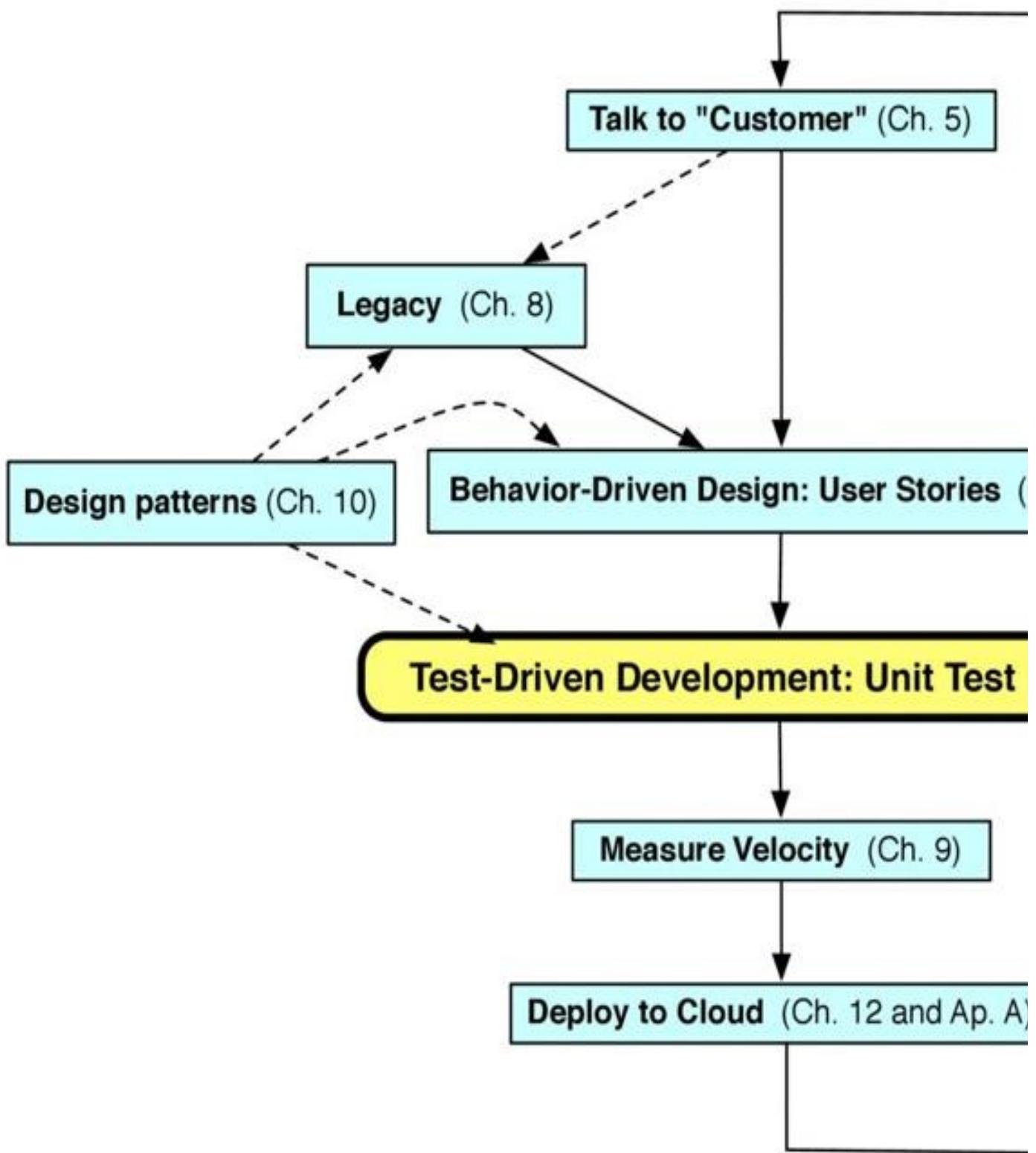


Figure 6.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter emphasizes unit testing as part of Test-Driven Development.

In Chapter 5 we began working on a new feature for Rotten Potatoes to enable information about a movie to be imported automatically from [The Open Movie Database](#). In this chapter, we'll develop the necessary methods to complete this feature.

Method or function? Following the terminology of OOP (object-oriented programming), we use *method* to mean a named piece of code that implements a behavior associated with a class, whether it's more like a function that returns a value or more like a procedure that causes side effects. Additional historical terms for such a piece of code include *function*, *routine*, *subroutine*, and *subprogram*.

Like many SaaS applications, TMDb is designed to be part of a Service-Oriented Architecture: it has an [**API**](#) (application programming interface) that allows external applications, not just human Web surfers, to use its functionality. As Screencast [6.1.1](#) shows, TMDb's API is RESTful, allowing each request made by an external application to be entirely self-contained, as described in Chapter [2](#).

Screencast 6.1.1: [Using the TMDb API](#)

TMDb's API is accessed by constructing a RESTful URI for the appropriate function, such as "search for movies by title" or "retrieve detailed information about a specific movie". To prevent abuse and track each user of the API separately, each developer must first obtain their own **API key** by requesting one via the TMDb website. Request URI's that do not include a valid API key are not honored, returning an error instead. For request URI's containing a valid API key, TMDb returns an XML document containing the result of the request encoded by the URI. This flow—construct a RESTful URI that includes an API key, receive an XML response—is a common pattern for interacting with external services.



Usually, calling such an API from Rotten Potatoes would require us to use the **URI** class in the Ruby standard library to construct the request URI containing our API key, use the **Net::HTTP** class to issue the request to `api.themoviedb.org`, and use an XML parsing library such as [**Nokogiri**](#) to parse the reply. But sometimes we can be more productive by standing on the shoulders of others, as we can in this case. The `ruby-tmdb` gem is one of two user-contributed Ruby "wrappers" around TMDb's RESTful API. It's mentioned on the TMDb API documentation pages, but a Google search for `tmdb ruby library` also yields it as a top result.

Screencast [6.1.2](#) shows how to use it.

Screencast 6.1.2: [Simplified use of the TMDb API with the ruby-tmdb gem](#)

Not every RESTful API has a corresponding Ruby library, but for those that do, such as TMDb, the library can hide the API details behind a few simple Ruby methods. The `ruby-tmdb` gem, which we installed via the Gemfile and Bundler as we learned in Chapter [3](#), manages URI construction and parses the XML result into Ruby objects such as hashes, strings, dates, and so on.

The days of developers “tossing their code over the wall” to [Quality Assurance \(QA\)](#) are largely over, as are the days of QA engineers manually exercising the software and filing bug reports. Indeed, the idea that quality assurance is the responsibility of a separate group rather than the result of a good process is considered antiquated. Today’s developers bear far more responsibility for testing their own code and participating in reviews; the responsibilities of QA have largely shifted to 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 9.

Testing today is also far more automated. Automated testing doesn’t mean that tests are created automatically for you, but that the tests are self-checking: 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. A high degree of automation is key to supporting the five principles for creating good tests, which 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. We

will use a Ruby tool called Autotest to help with this.



- **Independent:** No test should rely on preconditions created by other tests, so that we can prioritize running only a subset of tests that cover recent code changes.
- **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](#).



Y2K bug in action This photo was taken on Jan. 3, 2000. (Wikimedia Commons)

- **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.

Test-driven development (TDD) advocates the use of tests to *drive* the development of code. When TDD is used to create new code, as we do in this chapter, it is sometimes referred to as *test-first development* since the tests come into existence before any of the code being tested. When TDD is used to extend or modify legacy code, as we'll do in Chapter 8, new tests may be created for code that already exists. As we explore TDD in this chapter, we'll show how the Ruby tools support TDD and FIRST. Although TDD may feel strange when you first try it, it tends to result in code that is well tested, more modular, and easier to read than most code developed. While TDD is obviously not the only way to achieve those goals, it is difficult to end up with seriously deficient code if TDD is used correctly.



We will write tests using RSpec, a *domain-specific language* (DSL) for testing Ruby code. A DSL is a small programming language designed to ease tackling problems within a single area (domain) at the expense of generality. You've already seen examples of *external* (standalone) DSLs, such as HTML for describing Web pages. RSpec is an *internal* or *embedded* DSL: RSpec code is just Ruby code, but takes advantage of Ruby's features and syntax so as to make up a "mini-language" focused on the job of testing. Regular expressions are another example of an internal DSL embedded in Ruby.

RSpec can also be used for integration tests, but we prefer Cucumber since it facilitates dialogue with the customer and automates acceptance as well as integration tests.

RSpec's facilities help us capture *expectations* of how our code should behave. Such tests are executable specifications or "specs" written in Ruby, hence the name RSpec. How can we capture expectations in tests before there is any code to be tested? The surprising answer is that we write a test that exercises the *code we wish we had*, which forces us to think not only about what the code will do, but how it will be used by its callers and collaborators (other pieces of code that have to work with it). We did this in Chapter 5 in the Cucumber scenario step *And I click "Search TMDb"*: when we modified the List Movies view (`views/movies/index.html.haml`) to include a "Search TMDb" button, we picked the name `search_tmdb` for the not-yet-existing controller method that would respond to the click. Of course, since no method `MoviesController#search_tmdb` existed, the Cucumber step failed (showing red) when you tried to actually run the scenario. In the rest of this chapter we will use TDD to develop the `search_tmdb` method.

`Bar#foo` is idiomatic Ruby notation denoting the *instance* method `foo` of class `Bar`. The notation `Bar.foo` denotes the *class* method `foo`.

In the MVC architecture, the controller's job is to respond to a user interaction, call the appropriate model method(s) to retrieve or manipulate any necessary data, and generate an appropriate view. We might therefore describe the *desired* behavior of our as-yet-nonexistent controller method as follows:

- It should call a model method to perform the TMDb search, passing it the search terms typed by the user.
- It should select the Search Results HTML view (in Rails parlance, the Search

Results *template*) for rendering.

- It should make the TMDb search results available to that template.

Note that none of the methods or templates in this list of desiderata actually exists yet! That is the essence of TDD: write a concrete and concise list of the desired behaviors (the spec), and use it to drive the creation of the methods and templates.

<http://pastebin.com/Kye9NhS>

```
1 require 'spec_helper'  
2  
3 describe MoviesController do  
4   describe 'searching TMDb' do  
5     it 'should call the model method that performs TMDb sear-  
ch'  
6     it 'should select the Search Results template for render-  
ing'  
7     it 'should make the TMDb search results available to tha-  
t template'  
8   end  
9 end
```

Figure 6.2: Skeleton of RSpec examples for `MoviesController#search_tmdb`. By convention over configuration, the specs for `app/controllers/movies_controller.rb` are expected to be in `spec/controllers/movies_controller_spec.rb`, and so on. (Use Pastebin to copy-and-paste this code.)



You can see what they are in `spec/spec_helper.rb`.

Figure 6.2 shows how we would express these requirements in RSpec. As in Chapter 3, we encourage you learn by doing. Before creating this file, you need to set up RottenPotatoes to use RSpec for testing, which requires three steps:

In the `group :test` block in the Gemfile, add `gem 'rspec-rails'`

As always when modifying the Gemfile, run `bundle install --without production`

In the app root directory of Rotten Potatoes, run `rails generate rspec:install` to set up the files and directories RSpec needs.

You're now ready to create the file `spec/controllers/movies_controller_spec.rb` as shown in Figure 6.2. Line 1 loads some helper methods that will be used by all RSpec tests; in general, for Rails apps this will be the first line of any specfile. Line 3 says that the following specs **describe** the behavior of the `MoviesController` class. Because this class has several methods, line 4 says that this first set of specs describes the behavior of the method that searches TMDb. As you can see, **describe** can be followed by either a class name or a descriptive documentation string.

The next three lines are placeholders for **examples**, the RSpec term for a short piece of code that tests *one* specific behavior of the `search_tmdb` method. We haven't written any test code yet, but the next screencast shows that we can not only execute these test skeletons with the `r spec` command, but more importantly, automate running them with the `autotest` tool. This automation helps productivity since we don't have to shift our attention between writing code and running tests. For the rest of the chapter, we'll assume that `autotest` is running and that whenever you add tests or application code you will get immediate feedback from RSpec. In the next section we'll create our first tests using TDD.



Screencast 6.2.1: [Executing the empty test skeletons and automating execution with autotest](#)

When we run the RSpec command, examples (`it` clauses) containing no code are displayed in yellow as "pending". You can also explicitly mark an example using `pending` and provide a description of why it's pending. Rather than manually running `spec` each time we add or change some code, we can use the `autotest` command, which automatically reruns the appropriate specs whenever you change a specfile or code file.

ruby-debug and autotest To use the interactive debugger introduced in Chapter 3 with autotest, add `require 'ruby-debug'` to `spec/spec_helper.rb` and insert `debugger` calls wherever you want the action to stop.

Summary

- Good tests should be **Fast**, **Independent**, **Repeatable**, **Self-checking**, and **Timely** (FIRST).
- RSpec is a domain-specific language embedded in Ruby for writing tests.

Convention over configuration determines where the specfile corresponding to a given class file should reside.

- Within a specfile, a single **example**, introduced by the **it** method, tests a single behavior of a method. **describe** groups examples hierarchically according to the set of behaviors they test.

Self-Check 6.2.1. A single RSpec test case or *example* is introduced by the keyword _____. A group of related examples is introduced by the keyword _____, which can be nested to organize the examples in a file hierarchically.

 **it; describe**

Self-Check 6.2.2. Since RSpec matches tests to classes using convention over configuration, we would put the tests for app/models/movie.rb in the file _____.

 spec/models/movie_spec.rb

Figure 6.3 captures the basic TDD method. You might think we've violated the TDD methodology by writing down three test cases in Figure 6.2 before completing the code for any of those cases, but in practice, there's nothing wrong with creating **it** blocks for tests you know you will want to write. Now, though, it's time to get down to business and start working on the tests.

1. Before you write any new code, write a test for *one* aspect of the behavior it *should* 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.
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, repetition, or other ugliness that may have arisen as a result of adding the new code. The tests ensure that your refactoring doesn't introduce bugs.
5. Repeat until all behaviors necessary to pass a scenario step are complete.

Figure 6.3: The Test-Driven Development (TDD) loop is also known as Red–Green–Refactor because of its skeleton in steps 2–4. The last step assumes you are developing code in order to complete a scenario, such as the one you started in Chapter 5.

The first example (test case) in Figure 6.2 states that the **search_tmdb** method should call a model method to perform the TMDb search, passing the keywords typed by the user to that method. In Chapter 5, we modified the **index** view of

Rotten Potatoes by adding an HTML form whose submission would be handled by `MoviesController#search_tmdb`; the form contained a single text field called `search_terms` for the user to fill in. Our test case will therefore need to emulate what happens when the user types something into the `search_terms` field and submits the form. As we know, in a Rails app the `params` hash is automatically populated with the data submitted in a form so that the controller method can examine it. Happily, RSpec provides a `post` method that simulates posting a form to a controller action: the first argument is the action name (controller method) that will receive the post, and the second argument is a hash that will become the `params` seen by the controller action. We can now write the first line of our first spec, as Figure 6.4 shows. As the next screencast shows, though, we must overcome a couple of hurdles just to get to the Red phase of Red–Green–Refactor.

<http://pastebin.com/cd4pVa0a>

```
1 require 'spec_helper'  
2  
3 describe MoviesController do  
4   describe 'searching TMDb' do  
5     it 'should call the model method that performs TMDb sear  
ch' do  
6       post :search_tmdb, {:search_terms => 'hardware'}  
7     end  
8     it 'should select the Search Results template for render  
ing'  
9     it 'should make the TMDb search results available to tha  
t template'  
10    end  
11  end
```

Figure 6.4: Filling out the first spec. Whereas a “bare” `it` (line 8) serves as a placeholder for a yet-to-be-written example, an `it` accompanied by a `do...end` block (lines 5–7) is an actual test case.

Screencast 6.3.1: [Developing the first example requires adding an empty controller method and creating an empty view](#)

To get past RSpec’s errors, we first have to create an empty controller method and its corresponding route, so that the action (form submission by the user) would have somewhere to go. Then we need to create an empty view so that the

controller action has something to render. That one line of test code drove us to ensure that our new controller method and the view it will ultimately render have the correct names and have a matching route.

At this point RSpec reports Green for our first example, but that's not really accurate because the example itself is incomplete: we haven't actually checked whether `search_tmdb` calls a model method to search TMDb, as the spec requires. (We did this deliberately in order to illustrate some of the mechanics necessary to get your first specs running. Usually, since each spec tends to be short, you'd complete a spec before re-running your tests.)

The code we wish we had will be a class method, since finding movies in TMDb is a behavior related to movies in general and not to a particular instance of a `Movie`.

How should we check that `search_tmdb` calls a model method, since no model method exists yet? Again, we will write a test for the behavior of the *code we wish we had*, as directed in step 1 of Figure 6.3. Let's pretend we have a model method that does just what we want. In this case, we'd probably want to pass the method a string and get back a collection of `Movie` objects based on TMDb search results matching that string. If that method existed, our controller method might therefore call it like this:

<http://pastebin.com/5uacfzyZ>

```
1 @movies = Movie.find_in_tmdb(params[:search_terms])
```

<http://pastebin.com/vDRTDrkw>

```
1 require 'spec_helper'
2
3 describe MoviesController do
4   describe 'searching TMDb' do
5     it 'should call the model method that performs TMDb search' do
6       fake_results = [mock('movie1'), mock('movie2')]
7       Movie.should_receive(:find_in_tmdb).with('hardware').
8         and_return(fake_results)
9       post :search_tmdb, {:search_terms => 'hardware'}
10      end
11      it 'should select the Search Results template for rendering'
12      it 'should make the TMDb search results available to tha
```

```
t template'  
13 end  
14 end
```

Figure 6.5: Completing the example by asserting that the controller method will call the code we wish we had in the Movie model. Lines 5–10 in this listing replace lines 5–7 in Figure 6.4.

Figure 6.5 shows the code for a test case that asserts such a call will occur. In this case, the code we are testing—the **subject code**—is `search_tmdb`. However, part of the behavior we’re testing appears to depend on `find_in_tmdb`. Since `find_in_tmdb` is code we don’t yet have, the goal of lines 6–8 is to “fake” the behavior it would exhibit if we did have it. Line 6 uses RSpec’s `mock` method to create an array of two “test double” `Movie` objects. In particular, whereas a real `Movie` object would respond to methods like `title` and `rating`, the test double would raise an exception if you called any methods on it. Given this fact, why would we use doubles at all? The reason is to isolate these specs from the behavior of the `Movie` class, which might have bugs of its own. Mocks are like puppets whose behavior we completely control, allowing us to isolate unit tests from their collaborator classes and keep tests Independent (the I in FIRST).

In fact, an alias for `mock` is `double`. For clarity, use `mock` when you’re going to ask the fake object to do things, and `double` when you just need a stand-in.



Returning to Figure 6.5, lines 6–7 express the **expectation** that the `Movie` class should receive a call to the method `find_in_tmdb` and that method should receive the single argument `'hardware'`. RSpec will open the `Movie` class and define a class method called `find_in_tmdb` whose only purpose is to track whether it gets called, and if so, whether the right arguments are passed. Critically, *if a method with the same name already existed in the `Movie` class, it would be temporarily “overwritten” by this `method stub`.* That’s why in our case it doesn’t matter that we haven’t written the “real” `find_in_tmdb`: it wouldn’t get called anyway!

The use of `should_receive` to temporarily replace a “real” method for testing purposes is an example of using a **seam**: “a place where you can alter behavior in your program without editing in that place.” ([Feathers 2004](#)) In this case, `should_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.

Line 8 (which is just a continuation of line 7) specifies that `find_in_tmdb` should return the collection of doubles we set up in line 6. This completes the illusion of “the code we wish we had”: we’re calling a method that doesn’t yet exist, and supplying the result we wish it would give if it existed! If we omit `with`, RSpec will still check that `find_in_tmdb` gets called, but won’t check if the arguments are what we expected. If we omit `and_return`, the fake method call will return `nil` rather than a value chosen by us. In any case, after each example is run, RSpec performs a `teardown` step that restores the classes to their original condition, so if we wanted to perform these same fake-outs in other examples, we’d need to specify them in each one (though we’ll soon see a way to DRY out such repetition). This automatic teardown is another important part of keeping tests Independent.

Technically, in this case it would be OK to omit `and_return`, since this example isn’t checking the return value, but we included it for illustrative purposes.

This new version of the test fails because we established an expectation that `search_tmdb` would call `find_in_tmdb`, but the `search_tmdb` isn’t even written yet. Therefore the last step is to go from Red to Green by adding just enough code to `search_tmdb` to pass this test. We say the test *drives* the creation of the code, because adding to the test results in a failure that must be addressed by adding new code in the model. Since the only thing this particular example is testing is the method call to `find_in_tmdb`, it suffices to add to `search_tmdb` the single line of code we had in mind as “the code we wished we had”:

<http://pastebin.com/MDsSSwU1>

```
1 @movies = Movie.find_in_tmdb(params[:search_terms])
```

If TDD is new to you, this has been a lot to absorb, especially when testing an app using a powerful framework such as Rails. Don’t worry—now that you have been exposed to the main concepts, the next round of specs will go faster. It takes a bit of faith to jump into this system, but we have found that the reward is well worth it. Read the summary below and consider having a sandwich and reviewing the concepts in this section before moving on.

Summary

- The TDD cycle of Red–Green–Refactor begins with writing a test that fails because the **subject code** it’s testing doesn’t exist yet (Red) and then adding the minimum code necessary to pass just that one example (Green).

- Seams let you change the behavior of your application in a particular place without editing in that place. Typical test setup often establishes seams by using `mock` or its alias `double` to create test double objects, or by using `should_receive...and_return` to *stub* (replace and control the return value of) collaborator methods. Mocks and stubs are seams that help with testability by isolating the behavior of the code being tested from the behavior of its collaborators.
 - Each example sets up preconditions, executes the subject code, and asserts something about the results. Assertions such as `should`, `should_not`, `should_receive`, and `with` make tests **Self-checking**, eliminating the need for a human programmer to inspect test results.
 - After each test, an automatic teardown destroys the mocks and stubs and unsets any expectations, so that tests remain **Independent**.
-

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 website](#) 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 6.3.1. In Figure 6.5, why must the `should_receive` expectation in line 7 come *before* the `post` action in line 9?

• The expectation needs to set up a test double for `find_in_tmdb` that can be monitored to make sure it was called. Since the `post` action is eventually going to

result in calling `find_in_tmdb`, the double must be set up before the `post` occurs, otherwise the real `find_in_tmdb` would be called. (In this case, `find_in_tmdb` doesn't even exist yet, so the test would fail for that reason.) Returning to our original specfile skeleton from the listing in Figure 6.2, line 6 says that `search_tmdb` should select the "Search Results" view for rendering. Of course, that view doesn't exist yet, but as in the first example we wrote above, that needn't stop us.



Is this really necessary? Since the default view is determined by convention over configuration, all we're really doing here is testing Rails' built-in functionality. But if we were rendering one view if the action succeeded but a different view for displaying an error, examples like this would verify that the correct view was selected.

Since we know from Chapter 3 that the default behavior of the method `MoviesController#search_tmdb` is to attempt to render the view `app/views/movies/search_tmdb.html.haml` (which we created in Chapter 5), our spec just needs to verify that the controller action will indeed try to render that view template. To do this we will use the `response` method of RSpec: once we have done a `get` or `post` action in a controller spec, the object returned by the `response` method will contain the app server's response to that action, and we can assert an expectation that the response *would have rendered* a particular view. This happens in line 15 of Figure 6.6, which illustrates another kind of RSpec assertion: `object.should match-condition`. In this example, *match-condition* is supplied by `render_template()`, so the assertion is satisfied if the object (in this case the response from the controller action) attempted to render a particular view. We will see the use of `should` with other *match-conditions*. The negative assertion `should_not` can be used to specify that the *match-condition* should not be true.

<http://pastebin.com/3HELV51D>

```
1 require 'spec_helper'  
2  
3 describe MoviesController do  
4   describe 'searching TMDb' do  
5     it 'should call the model method that performs TMDb search' do  
6       fake_results = [mock('movie1'), mock('movie2')]  
7       Movie.should_receive(:find_in_tmdb).with('hardware').  
8         and_return(fake_results)  
9       post :search_tmdb, { :search_terms => 'hardware' }  
10      end
```

```

11      it 'should select the Search Results template for rendering' do
12          fake_results = [mock('Movie'), mock('Movie')]
13          Movie.stub(:find_in_tmdb).and_return(fake_results)
14          post :search_tmdb, {:search_terms => 'hardware'}
15          response.should render_template('search_tmdb')
16      end
17      it 'should make the TMDb search results available to the template'
18  end
19 end

```

Figure 6.6: Filling out the second example. Lines 11–16 replace line 11 from Figure [6.5](#).

There are two things to notice about Figure [6.6](#). First, since each of the two examples (lines 5–10 and 11–16) are self-contained and **Independent**, we have to create the test doubles and perform the `post` command separately in each. Second, whereas the first example uses `should_receive`, the second example uses `stub`, which creates a test double for a method but *doesn't* establish an expectation that that method will necessarily be called. The double springs into action *if* the method is called, but it's not an error if the method is never called. Make the changes so that your specfile looks like Figure [6.6](#); autotest should still be running and report that this second example passes.

In this simple example, you could argue that we're splitting hairs by using `should_receive` in one example and `stub` in another, but the goal is to illustrate that *each example should test a single behavior*. This second example is *only* checking that the correct view is selected for rendering. It's *not* checking that the appropriate model method gets called—that's the job of the first example. In fact, even if the method `Movie.find_in_tmdb` actually *was* implemented already, we'd still stub it out in these examples, because examples should isolate the behaviors under test from the behaviors of other classes with which the subject code collaborates.



Before we write another example, we consider the Refactor step of Red-Green-Refactor. Given that lines 6 and 12 are identical, Figure [6.7](#) shows one way to DRY them out by **factoring out** common setup code into a `before(:each)` block. As the name implies, this code is executed before *each* of the examples within the `describe` example group, similar to the **Background** section of a Cucumber feature, whose steps are performed before each scenario. There is also `before(:all)`, which runs setup code just once for a whole group of tests; but you

risk making your tests dependent on each other by using it, since it's easy for hard-to-debug dependencies to creep in that are only exposed when tests are run in a different order or when only a subset of tests are run.

While the concept of factoring out common setup into a `before` block is straightforward, we had to make one syntactic change to make it work, because of the way RSpec is implemented. Specifically, we had to change `fake_results` into an instance variable `@fake_results`, because local variables occurring inside each test case's `do...end` block disappear once that test case finishes running. In contrast, instance variables of an example group are visible to all examples in that group. Since we are setting the value in the `before :each` block, every test case will see the same initial value of `@fake_results`.

Instance variable of what? `describe` creates and returns a new `Test::Spec::ExampleGroup` object that represents a group of your test cases. `@fake_results` is an instance variable of this `ExampleGroup` object—not of the class under test (`MoviesController`).

<http://pastebin.com/dubDZGTQ>

```
1 require 'spec_helper'
2
3 describe MoviesController do
4   describe 'searching TMDb' do
5     before :each do
6       @fake_results = [mock('movie1'), mock('movie2')]
7     end
8     it 'should call the model method that performs TMDb sear
ch' do
9       Movie.should_receive(:find_in_tmdb).with('hardware').
10         and_return(@fake_results)
11       post :search_tmdb, { :search_terms => 'hardware' }
12     end
13     it 'should select the Search Results template for render
ing' do
14       Movie.stub(:find_in_tmdb).and_return(@fake_results)
15       post :search_tmdb, { :search_terms => 'hardware' }
16       response.should render_template('search_tmdb')
17     end
18     it 'should make the TMDb search results available to tha
t template'
19   end
20 end
```

Figure 6.7: DRYing out the controller examples using a `before` block (lines 5–7).

There's just one example left to write, to check that the TMDb search results will be made available to the response view. Recall that in Chapter 5, we created `views/movies/search_tmdb.html.haml` under the assumption that `@movies` would be set up by the controller action to contain the list of matching movies from TMDb. That's why in `MoviesController#search_tmdb` we assigned the result of calling `find_in_tmdb` to the instance variable `@movies`. (Recall that instance variables set in a controller action are available to the view.) The RSpec `assigns()` method keeps track of what instance variables were assigned in the controller method. Hence `assigns(:movies)` returns whatever value (if any) was assigned to `@movies` in `search_tmdb`, and our spec just has to verify that the controller action correctly sets up this variable. In our case, we've already arranged to return our doubles as the result of the faked-out method call, so the correct behavior for `search_tmdb` would be to set `@movies` to this value, as line 21 of Figure 6.8 asserts.

<http://pastebin.com/ymzLR5ni>

```
1 require 'spec_helper'  
2  
3 describe MoviesController do  
4   describe 'searching TMDb' do  
5     before :each do  
6       @fake_results = [mock('movie1'), mock('movie2')]  
7     end  
8     it 'should call the model method that performs TMDb sear  
ch' do  
9       Movie.should_receive(:find_in_tmdb).with('hardware').  
10      and_return(@fake_results)  
11      post :search_tmdb, {:search_terms => 'hardware'}  
12    end  
13    it 'should select the Search Results template for render  
ing' do  
14      Movie.stub(:find_in_tmdb).and_return(@fake_results)  
15      post :search_tmdb, {:search_terms => 'hardware'}  
16      response.should render_template('search_tmdb')  
17    end
```

```

18     it 'should make the TMDb search results available to the template' do
19       Movie.stub(:find_in_tmdb).and_return(@fake_results)
20       post :search_tmdb, {:search_terms => 'hardware'}
21       assigns(:movies).should == @fake_results
22     end
23   end
24 end

```

Figure 6.8: Asserting that `@movie` is set up correctly by `search_tmdb`. Lines 18–22 in this listing replace line 18 in Figure [6.7](#).

ELABORATION: More than we need?

Strictly speaking, for the purposes of this example the stubbed `find_in_tmdb` could have returned any value at all, such as the string “I am a movie”, because the *only* behavior tested by this example is whether the correct instance variable is being set up and made available to the view. In particular, this example doesn’t care what the *value* of that variable is, or whether `find_in_tmdb` is returning something sensible. But since we already had doubles set up, it was easy enough to use them in this example.

<http://pastebin.com/qg5nXEK0>

```

1 require 'spec_helper'
2
3 describe MoviesController do
4   describe 'searching TMDb' do
5     before :each do
6       @fake_results = [mock('movie1'), mock('movie2')]
7     end
8     it 'should call the model method that performs TMDb search' do
9       Movie.should_receive(:find_in_tmdb).with('hardware').
10         and_return(@fake_results)
11       post :search_tmdb, {:search_terms => 'hardware'}
12     end
13   describe 'after valid search' do
14     before :each do
15       Movie.stub(:find_in_tmdb).and_return(@fake_results)
16       post :search_tmdb, {:search_terms => 'hardware'}
17     end

```

```
18      it 'should select the Search Results template for rend  
ering' do  
19          response.should render_template('search_tmdb')  
20      end  
21      it 'should make the TMDb search results available to t  
hat template' do  
22          assigns(:movies).should == @fake_results  
23      end  
24  end  
25 end  
26 end
```

Figure 6.9: Completed and refactored spec for `search_tmdb`. The nested group starting at line 13 allows DRYing out the duplication between lines 14–15 and 19–20 in Figure 6.8.



Our last task in Red–Green–Refactor is the Refactor step. The second and third examples are identical except for the last line in each one (lines 16 and 21). To DRY them out, Figure 6.9 starts a separate nested example group with `describe`, grouping the common behaviors of the last two examples into their own `before` block. We chose the description string after `valid search` to name this `describe` block because the examples in this subgroup all assume that a valid call to `find_in_tmdb` has occurred. (That assumption itself is tested by the first example.)

When example groups are nested, any `before` blocks associated with the outer nesting are executed prior to those associated with the inner nesting. So, for example, considering the test case in lines 18–20 of Figure 6.9, the setup code in lines 5–7 is run first, followed by the setup code in lines 14–17, and finally the example itself (lines 18–20).

Our next task will be to use TDD to create the model method `find_in_tmdb` that we've so far been stubbing out. Since this method is supposed to call the actual TMDb service, we will again need to use stubbing, this time to avoid having our examples depend on the behavior of a remote Internet service.

Summary

- An example of the Refactor step of Red–Green–Refactor is to move common

setup code into a `before` block, thus DRYing out your specs.

- Like `should_receive`, stubbing with `stub` creates a “test double” method for use in tests, but unlike `should_receive`, `stub` doesn’t require that the method actually be called.
- `assigns()` allows a controller test to inspect the values of instance variables set by a controller action.

Self-Check 6.4.1. Specify whether each of the following RSpec constructs is used to (a) create a seam, (b) determine the behavior of a seam, (c) neither: (1) `assigns()`; (2) `should_receive`; (3) `stub`; (4) `and_return`.

 (1) c, (2) a, (3) a, (4) b

Self-Check 6.4.2. Why is it usually preferable to use `before(:each)` rather than `before(:all)`?

 Code in a `before(:each)` block is run before each spec in that block, setting up identical preconditions for all those specs and thereby keeping them Independent. Mocks and stubs are appropriate when you need a stand-in with a small amount of functionality to express a test case. But suppose you were testing a new method `Movie#name_with_rating` that you know will examine the `title` and `rating` attributes of a `Movie` object. You could create a mock that knows all that information, and pass that mock:

<http://pastebin.com/dyz0DBED>

```
1 fake_movie = mock('Movie')
2 fake_movie.stub(:title).and_return('Casablanca')
3 fake_movie.stub(:rating).and_return('PG')
4 fake_movie.name_with_rating.should == 'Casablanca (PG)'
```

But there are two reasons not to use a mock here. First, this mock object needs almost as much functionality as a real `Movie` object, so you’re probably better off using a real object. Second, since the instance method being tested is part of the `Movie` class itself, it makes sense to use a real object since this isn’t a case of isolating the test code from collaborator classes.

You have two choices for where to get a real `Movie` object to use in such tests. One choice is to set up one or more [fixtures](#)—a fixed state used as a baseline for one or more tests. 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. Like setup and teardown of mocks and stubs, the test database is erased and reloaded with the fixtures before *each spec*, keeping tests **Independent**. Rails looks for fixtures in a file containing **[YAML](#)** (Yet Another Markup Language) objects. As Figure [6.10](#) shows, YAML is a very simple-minded way of representing hierarchies of objects with attributes, similar to XML, which we saw at the beginning of the chapter. 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 [6.10](#) shows.

Strictly speaking, it's not erased, but each spec is run inside a [database transaction](#) that is rolled back when the spec finishes.



<http://pastebin.com/piT1LNGX>

```
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
```

<http://pastebin.com/VRqWKPeQ>

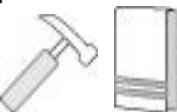
```
1 # spec/models/movie_spec.rb:
2
3 describe Movie do
4   fixtures :movies
5   it 'should include rating and year in full name' do
6     movie = movies(:milk_movie)
7     movie.name_with_rating.should == 'Milk (R)'
8   end
```

```
9 end
```

Figure 6.10: Fixtures declared in YAML files (top) are automatically loaded into the test database before each spec is executed (bottom).

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 each individual test probably relies on only one or two fixtures, the union of fixtures required by all tests can become unwieldy. For this reason, many programmers prefer to use a [factory](#)—a framework designed to allow quick creation of full-featured objects (rather than mocks) at testing time. For example, the popular [FactoryGirl](#) tool for Rails lets you define a factory for Movie objects and create just the objects you need quickly for each test, selectively overriding only certain attributes, as Figure 6.11 shows. (FactoryGirl is part of the bookware.) In our simple app, using a factory doesn't confer much benefit over just calling `Movie.new` to create a new Movie directly. But in more complicated apps in which object creation and initialization involve many steps—for example, objects that have many attributes that must be initialized at creation time—a factory helps DRY out your test preconditions (**before** blocks)

and streamline your test code.



<http://pastebin.com/SnVg4jDz>

```
1 # spec/factories/movie.rb
2
3 FactoryGirl.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
```

<http://pastebin.com/sgs6Etpr>

```

1 # spec/models/movie_spec.rb
2
3 describe Movie do
4   it 'should include rating and year in full name' do
5     movie = FactoryGirl.build(:movie, :title => 'Milk', :rating => 'R')
6     movie.name_with_rating.should == 'Milk (R)'
7   end
8 end
9
10 # or if you mix in FactoryGirl's syntax methods (see FactoryGirl README):
11
12 describe Movie do
13   subject { create(:movie, :title => 'Milk', :rating => 'R') }
14   its(:name_with_rating) { should == 'Milk (R)' }
15 end

```

Figure 6.11: An example of using FactoryGirl instead of fixtures to create real (not mock) objects for use in tests. Factory frameworks streamline the creation of such objects while preserving Independence among tests. Add `gem 'factory_girl_rails'` to your Gemfile to use FactoryGirl.

Before adding more functionality, let's dig a bit more deeply into how RSpec works. RSpec's `should` is a great example of the use of Ruby language features to improve readability and blur the line between tests and documentation. The following screencast explains in a bit more detail how an expression such as `value.should == 5` is actually handled.

Screencast 6.5.1: [How Ruby's dynamic language features make specs more readable](#)
 RSpec mixes a module containing the `should` method into the `Object` class. `should` expects to be passed a *matcher* that can be evaluated to the condition being asserted. RSpec methods such as `be` can be used to construct such a matcher; because of Ruby's flexible syntax and optional parentheses, an assertion such as `value.should be < 5` can be understood by fully parenthesizing and de-sugaring it to `value.should(be.<(5))`. In addition, RSpec uses Ruby's `method_missing` feature (described in Chapter 3) to detect matchers beginning with `be_` or `be_a_`.

allowing you to create assertions such as `cheater.should be_disqualified`.
(Note: The spec shown at the beginning of this Alpha edition screencast doesn't correspond to the example being developed in this section. However, this doesn't affect the main point of the screencast, which is to illustrate in detail how `should` works in RSpec.)

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.
 - 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 6.5.1. 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. We've now created two of the three parts of the new "Search TMDb" feature: we created the view in Chapter 5 and we used TDD to drive the creation of the controller action in the previous sections. All that remains to finish the user story we started in Chapter 5 is the model method `find_in_tmdb`, which actually uses Service-Oriented Architecture technology to communicate with TMDb. Using TDD to drive its implementation will go quickly now that we know the basics.

By convention over configuration, specs for the `Movie` model go in `spec/models/movie_spec.rb`. Figure 6.12 shows the happy path for calling `find_in_tmdb`, which describes what happens when everything works correctly. (Complete specs must also cover the sad paths, as we'll soon see.) Inside the overall `describe Movie`, we've added a nested `describe` block for the keyword-search function. Our first spec says that when `find_in_tmdb` is called with a string

parameter, it should pass that string parameter as the value of the `:title` option to the TMDb gem's `TmdbMovie.find` method. We express this using `hash_including`, a matcher that specifies that a particular argument (in this case, the only argument) must be a hash including the key/value pair `:title => 'Inception'`. This spec should immediately fail because we haven't defined `find_in_tmdb` yet, so we are at the Red stage already. Of course at this stage `find_in_tmdb` is trivial, so the bottom of Figure 6.12 shows its initial



implementation that gets us from Red to Green.

<http://pastebin.com/rQfFZhqJ>

```
1 require 'spec_helper'  
2  
3 describe Movie do  
4   describe 'searching Tmdb by keyword' do  
5     it 'should call Tmdb with title keywords' do  
6       TmdbMovie.should_receive(:find).with(hash_including :  
7         title => 'Inception')  
8       Movie.find_in_tmdb('Inception')  
9     end  
10    end  
11  end
```

<http://pastebin.com/B1wqGuAj>

```
1 class Movie < ActiveRecord::Base  
2  
3   def self.find_in_tmdb(string)  
4     TmdbMovie.find(:title => string)  
5   end  
6  
7   # rest of file elided for brevity  
8 end
```

Figure 6.12: (Top) the happy path spec for calling the Tmdb gem; (bottom) Initial happy path implementation driven by happy path spec.

However, this spec is subtly incomplete! So far, all of our test cases have been based on the ***explicit requirement*** described in the user story of Chapter 5: when the user types in the name of a movie and clicks *Search TMDb*, she should see a page showing matching results. However, Screencast 6.1.1 revealed the additional ***implicit requirement*** that every TMDb request must include a valid API key. Further, Screencast 6.1.2 showed that the ruby-tmdb gem gives a *different* error for a blank API key than for an invalid one, so we must be prepared to handle both kinds of errors. Our specs so far do not cover these implicit requirements, and since we have been stubbing the call to `TmdbMovie.find`, we have been masking the errors that would otherwise occur.

So we must augment our description of the model method to capture these implicit requirements:

- It should raise an exception if an invalid key is provided.
- It should raise an exception if a blank key is provided.

Unfortunately, the way ruby-tmdb is implemented, it raises a `RuntimeError`, the most generic kind of Ruby exception. For Rotten Potatoes, we would prefer that `find_in_tmdb` raise an exception specific to this type of error, as the revised spec in Figure 6.13 demands. Note that we have renamed our first spec to indicate that it applies to the revised case when the API key is valid, and added a new spec to cover the case when the API key hasn't been initialized.

<http://pastebin.com/TJXJk5wQ>

```
1 require 'spec_helper'  
2  
3 describe Movie do  
4   describe 'searching Tmdb by keyword' do  
5     it 'should call Tmdb with title keywords given valid API  
key' do  
6       TmdbMovie.should_receive(:find).  
7         with(hash_including :title => 'Inception')  
8       Movie.find_in_tmdb('Inception')  
9     end  
10    it 'should raise an InvalidKeyError with no API key' do  
11      lambda { Movie.find_in_tmdb('Inception') }.
```

```
12     should raise_error(Movie::InvalidKeyError)
13   end
14 end
15 end
```

Figure 6.13: The code we wish we had would raise a very specific exception to signal a missing API key (lines 11–12), but this spec fails because `find_in_tmdb` raises its own generic exception instead.

Let's take a moment to understand both the code itself and why it fails with Red. Notice first that we had to "wrap" the call to `find_in_tmdb` in a `lambda`. We expect the call to raise an exception, but if a spec raises an exception it stops the testing run! So in order to make the spec Self-checking, we invoke `should` on the callable `lambda` object, which will cause the lambda to be executed in a "controlled environment" where RSpec can catch any exceptions and match them to our expectation.

Nevertheless, the test fails with the message `uninitialized constant Movie::InvalidKeyError`. Recall that a Ruby class is simply a constant that names a class object; this message is telling us that we've referenced an exception class called `Movie::InvalidKeyError` but haven't defined it anywhere. Line 3 of Figure 6.14 fixes this by defining this new class within `Movie`'s namespace (hence the notation `Movie::InvalidKeyError` in line 12 of the spec). When you make this change, the test will still fail, but now it will fail for the right reason: we've asserted that a `Movie::InvalidKeyError` should be raised, but instead got an `ArgumentError` from `TmdbMovie.find`. Figure 6.14 shows the easy fix that makes the test pass Green. Notice in line 9 that we pass the error message from the original `ArgumentError` exception as the descriptive message of our new exception.

<http://pastebin.com/5LhUz8uV>

```
1 class Movie < ActiveRecord::Base
2
3   class Movie::InvalidKeyError < StandardError ; end
4
5   def self.find_in_tmdb(string)
6     begin
7       TmdbMovie.find(:title => string)
8     rescue ArgumentError => tmdb_error
```

```
9      raise Movie::InvalidKeyError, tmdb_error.message
10    end
11  end
12
13 # rest of file elided for brevity
14 end
```

Figure 6.14: Defining our new exception type within class `Movie` (line 3) and getting the behavior we want by rescuing the exception thrown by `TmdbMovie.find` and re-raising our new exception type (lines 6–10).

Summary

- Sometimes explicit requirements lead to additional implicit requirements—additional constraints that are not “visible” like the explicit requirements but must still be satisfied for the explicit requirement to be met. Implicit requirements are just as important as explicit ones and should be tested with the same rigor.
- If we need to check that the subject code raises an exception, we can do so by making a lambda-expression the receiver of an expectation like `should` or `should_not` and using the matcher `raise_error`.
- As an example of how to perform only a partial check on arguments using `with`, we matched the argument using `hash_including`, which checks that the argument quacks like a hash and contains a particular key/value pair but makes no assertions about its other contents.

ELABORATION: Declared vs. undeclared exceptions

In statically-typed languages such as Java, the compiler enforces that a method must declare any exceptions it might throw. If the callee wants to add a new type of exception, the callee’s method signature changes, requiring the callee and all callers to be recompiled.

This approach doesn’t extend well to SaaS apps, which may communicate with other services like TMDb whose evolution and behavior are not under our control. We must rely on the remote service’s API documentation to tell us what could go wrong, but we must also assume that other undocumented failures can also happen. In those cases the best we can do is catch the exception, log detailed information, and (depending on our judgment) notify the maintainers of Rotten Potatoes that an

unexpected error occurred.

Self-Check 6.6.1. Considering line 11 of Figure 6.13, suppose we didn't wrap the call to `find_in_tmdb` in a lambda-expression. What would happen and why?

➊ If `find_in_tmdb` correctly raises the exception, the spec will fail because the exception will stop the run. If `find_in_tmdb` incorrectly fails to raise an exception, the spec will fail because the assertion `should raise_error` expects one. Therefore the test would always fail whether `find_in_tmdb` was correct or not.

Self-Check 6.6.2. Given that failing to initialize a valid API key raises an exception, why doesn't line 8 of Figure 6.13 raise an exception?

➋ Because lines 6–7 replace the `TmdbMovie.find` call with a stub, preventing the "real" method from executing and raising an exception.

Be sure to acquire and use your own [TMDB API key](#) rather than the one hardcoded in the example listing!

The above sad path captures the case of not initializing the API key at all, but what if we specify a TMDB API key but it's an invalid one? This requires us to finally add some code to actually initialize the API key. Figure 6.15 adds an accessor method for the key and sets the key's value in every call to `find_in_tmdb`.

<http://pastebin.com/aGFD3CLb>

```
1 class Movie < ActiveRecord::Base
2
3   class Movie::InvalidKeyError < StandardError ; end
4
5   def self.api_key
6     'cc4b67c52acb514bdf4931f7cedfd12b' # replace with YOUR T
7   mdb key
8
9   def self.find_in_tmdb(string)
10    Tmdb.api_key = self.api_key
11    begin
12      TmdbMovie.find(:title => string)
13      rescue ArgumentError => tmdb_error
14        raise Movie::InvalidKeyError, tmdb_error.message
15    end
16  end
17  # rest of file elided for brevity
18 end
```

Figure 6.15: A simple method to initialize the API key, defined in lines 5–7 and used in line 10. In a real application, the key string would probably be read from a configuration file, but creating this accessor method lets us change the implementation of how the key is retrieved without affecting other code.

Why set the key before every `find` rather than just once? Setting the key is an inexpensive operation, so the extra code required to check if the key has been initialized would hurt conciseness without improving performance. If setting the key were expensive, one-time initialization might make sense. But if your key is valid, this change should result in a failing spec, since the spec in line 12 will no longer raise an exception. We must modify the spec to deliberately set the API key to the empty string (as it would be if we didn't initialize it). Happily, we already know how to do this: we stub out the `Movie.api_key` accessor and return an empty string, as shown in Figure 6.16. This change makes the spec in line 12 revert to Green. Note that setting the key by calling an accessor method creates a seam that we can stub to force a failing value in this test. If the key had been set from a variable, this would have been harder to do. With this seam in place, we can finally test the behavior of having a non-blank but invalid API key. The spec in Figure 6.17 is easy to write since all the pieces are in place. Perhaps surprisingly, the spec doesn't pass:

<http://pastebin.com/3nYhtrcH>

```
1
expected Movie::InvalidKeyError, got #<RuntimeError: Tmdb API
returned
2     status code '404' for URL:
3
'http://api.themoviedb.org/2.1/Movie.search/en/json/INVALID/In
ception'>
```

<http://pastebin.com/RRKJDpbn>

```
1 require 'spec_helper'
2
3 describe Movie do
4   describe 'searching Tmdb by keyword' do
5     # first spec elided for brevity
6     it 'should raise an InvalidKeyError with no API key' do
```

```
7     Movie.stub(:api_key).and_return('')
8     lambda { Movie.find_in_tmdb('Inception') }.
9       should raise_error(Movie::InvalidKeyError)
10    end
11  end
12 end
```

Figure 6.16: Because Figure 6.15 uses an accessor method `Movie.api_key` for getting the key's value rather than assigning it from a variable, it provides us a seam that enables us to control test behavior.

<http://pastebin.com/dgSHRKYk>

```
1 require 'spec_helper'
2
3 describe Movie do
4   describe 'searching Tmdb by keyword' do
5     # first 2 specs elided for brevity
6     it 'should raise an InvalidKeyError with invalid API key
' do
7       Movie.stub(:api_key).and_return('INVALID')
8       lambda { Movie.find_in_tmdb('Inception') }.
9         should raise_error(Movie::InvalidKeyError)
10      end
11    end
12  end
```

Figure 6.17: Testing for a nonblank but invalid key causes a failure because `TmdbMovie.find` raises a different exception for an invalid key than it does for a blank or uninitialized key.

There are two problems here. First, as Screencast 6.1.2 showed, `TmdbMovie.find` raises `ArgumentError` if the key is blank but `RuntimeError` if the key is invalid. (Presumably the reason is that a blank key is obviously invalid, but a nonblank key can only be tested by calling the TMDb service with it.) The problem of raising a different exception can be fixed by adding another `rescue` clause to catch the `RuntimeError` exception, as Figure 6.18 shows.

<http://pastebin.com/1C08dxAu>

```
1 class Movie < ActiveRecord::Base
2
3   class Movie::InvalidKeyError < StandardError ; end
4
5   def self.api_key
6     'cc4b67c52acb514bdf4931f7cedfd12b' # replace with YOUR T
7   mdb key
8   end
9
10  def self.find_in_tmdb(string)
11    Tmdb.api_key = self.api_key
12    begin
13      TmdbMovie.find(:title => string)
14    rescue ArgumentError => tmdb_error
15      raise Movie::InvalidKeyError, tmdb_error.message
16    rescue RuntimeError => tmdb_error
17      if tmdb_error.message =~ /status code '404'/
18        raise Movie::InvalidKeyError, tmdb_error.message
19      else
20        raise RuntimeError, tmdb_error.message
21      end
22    end
23  end
24 # rest of file elided for brevity
25 end
```

Figure 6.18: Rescuing `RuntimeError` as well as `ArgumentError` makes the test pass. Rescuing from `RuntimeError` can be dangerous since it's a generic catch-all for runtime exceptions that aren't otherwise handled, so we specifically check that the exception message matches the TMDb error message we saw in Screencast [6.1.1](#); if it doesn't, we just re-raise the `RuntimeError`.

That leads us to the second problem, which is more serious: this spec actually calls the TMDb service every time it's executed, making it neither **Fast** (it can take a few seconds for each call to complete) nor **Repeatable** (the test will behave differently if TMDb is down or your computer is not connected to the Internet). We already know how to fix this: introduce a seam that isolates the caller from the callee, as we did earlier with `TmdbMovie.find`. Since we now know that

`TmdbMovie.find` raises a `RuntimeError` when given an invalid key, we can mimic that behavior with a stub that does just that. Note that this stub obsoletes the need for stubbing an invalid key value, since we are faking the behavior of the TMDb remote service itself. Figure 6.19 shows the complete `movie_spec.rb` including this change, which keeps the spec Green but “disconnects” it from the real TMDb service.

<http://pastebin.com/aMnxZz5E>

```
1 require 'spec_helper'
2
3 describe Movie do
4   describe 'searching Tmdb by keyword' do
5     it 'should call Tmdb with title keywords given valid API
key' do
6       TmdbMovie.should_receive(:find).
7         with(hash_including :title => 'Inception')
8       Movie.find_in_tmdb('Inception')
9     end
10    it 'should raise an InvalidKeyError with no API key' do
11      Movie.stub(:api_key).and_return('')
12      lambda { Movie.find_in_tmdb('Inception') }.
13        should raise_error(Movie::InvalidKeyError)
14    end
15    it 'should raise an InvalidKeyError with invalid API key
' do
16      TmdbMovie.stub(:find).
17        and_raise(RuntimeError.new("API returned status code
'404'"))
18      lambda { Movie.find_in_tmdb('Inception') }.
19        should raise_error(Movie::InvalidKeyError)
20    end
21  end
22 end
```

Figure 6.19: Mimicking the behavior of the TMDb API when an invalid key is passed (lines 16–17) breaks the dependency between the invalid-API-key spec and TMDb itself.

Figure 6.19 raises a more general question: where should we stub external methods? We chose to test TMDb integration by stubbing `find_in_tmdb` but there are multiple places we could have stubbed. Stubbing the methods of the `ruby-tmdb` gem would allow our tests to catch errors in the interaction between `Movie.find_in_tmdb` and `ruby-tmdb` itself. More radically, we could use a gem like [FakeWeb](#) to stub out the entire Web except for particular URIs that return a canned response when accessed from a Ruby program. (You can think of `FakeWeb` as `stub...with...and_return` for the whole Web.) For example, we could create a canned XML document by cutting and pasting the result of a live call, and use `FakeWeb` to arrange to return that document when the app uses the appropriate RESTful URI to attempt to contact TMDb. From an integration testing standpoint, this is the most realistic way to test, 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, `FakeWeb` is an 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.

Summary

- To test code that communicates with an external service, we create stubs to mimic the service’s behavior in order to keep our tests **Fast** and **Repeatable**.
- We can learn about the behavior we need to mimic both by reading the service’s API documentation and by directly observing the service’s real behavior under a variety of conditions.

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 'should do 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=mock('movie')`
Creates a mock object with no predefined methods
 - `m.stub(: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=mock('movie', :rating=>'R')`
Shortcut that combines the 2 previous examples
 - `Movie.stub(:find).and_return(@fake_movie)`
Forces `@fake_movie` to be returned if `Movie.find` is called, but doesn't require that it be called
-

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.
 - `response.should render_template('show')`
Checks that the controller action renders the `show` template for this controller's model
 - `response.should redirect_to(:controller => 'movies', :action => 'new')`
Checks that the controller action redirects to `MoviesController#new` rather than rendering a view
-

Figure 6.20: Some of the most useful RSpec methods introduced in this chapter. See the [full RSpec documentation](#) for details and additional methods not listed here.

Assertions on method calls: can also negate, e.g. `should_not_receive` ·
`Movie.should_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 ·
`Movie.should_receive(:find).with('Milk','R')`

Checks that `Movie.find` is called with exactly 2 arguments having these values ·
`Movie.should_receive(:find).with(anything())`

Checks that `Movie.find` is called with 1 argument whose value isn't checked ·
`Movie.should_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`' ·
`Movie.should_receive(:find).with(no_args())`

Checks that `Movie.find` is called with zero arguments

Matchers

- `greeting.should == 'bonjour'`
Compares its argument for equality with receiver of assertion
 - `value.should be >= 7`
Compares its argument with the given value; syntactic sugar for
`value.should(be.>=(7))`
 - `result.should be_remarkable`
Calls `remarkable?` (note question mark) on `result`
-

Figure 6.21: Continuation of summary of useful RSpec methods introduced in this chapter.

ELABORATION: Red-Refactor-Green-Refactor?

Sometimes getting from Red to Green is not just a matter of adding new code, but may first require refactoring existing code. For example, if line 10 of Figure 6.15 had set the API key from a variable (`Tmdb.api_key=@api_key`) rather than by calling the accessor method in lines 5–7, we would have refactored to create the accessor method in order to have the testing seam needed in line 7 of Figure 6.16. This refactoring would have been simple, but more complex refactorings may require code changes that temporarily make some Green tests go Red. It's important to change those tests as necessary to make them pass again before continuing with your new test. Remember that the goal is to always have working code.

Self-Check 6.7.1. Name two likely violations of FIRST that arise when unit tests actually call an external service as part of testing.

➊ 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.

Self-Check 6.7.2. How would the spec in Figure 6.19 be simplified if `TmdbMovie` raised the *same* exception for either a blank or an invalid key?

➋ With only one condition to check, either lines 10–14 or 15–19 could be eliminated from the spec.

How much testing is enough? A poor but unfortunately widely-given answer is “As much as you can do before the shipping 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.

A more precise way to approach the question is in terms of **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. Here is a simple fragment of code and the definitions of several commonly-used coverage terms as they apply to the example.

<http://pastebin.com/aNhQjhbt>

```
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 6.22: A simple code example to illustrate basic coverage concepts.

- 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.
| Sometimes written with a subscript, S_0 .
- 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.
- 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 untested or untested parts of the code and show the overall comprehensiveness of your test suite. The next screencast shows how to use the [SimpleCov](#) Ruby gem (included in

the bookware) to quickly check the C0 coverage of your RSpec tests.



Screencast 6.8.1: [Using SimpleCov to check C0 coverage](#)

The SimpleCov tool, provided as a Ruby gem, measures and displays the C0

coverage of your specs. You can zoom in on each file and see which specific lines were covered by your tests.

This chapter, and the above discussion of coverage, have focused on unit tests. Chapter 5 explained how user stories could become automated acceptance tests; those are [**integration tests**](#) or [**system tests**](#) because each test (that is, each scenario) exercises a lot of code in many different parts of the application, rather than relying on fake objects such as mocks and stubs to isolate classes from their collaborators. 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. Their coverage 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, 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.

Somewhere in between these levels are [**functional tests**](#), which exercise a well-defined subset of the code. They rely on mocks and stubs to isolate a set of cooperating classes rather than a single class or method. For example, controller specs such as Figure 6.9 use `get` and `post` methods to submit URIs to the app, which means they rely on the routing subsystem to work correctly in routing those calls to the appropriate controller methods. (To see this for yourself, temporarily remove the line `resources :movies` from `config/routes.rb` and try re-running the controller specs.) However, the controller specs are still isolated from the database by stubbing out the model method `find_in_tmdb` that would normally communicate with the database.

In other words, high assurance requires both good coverage and a mix of all three kinds of tests. Figure 6.23 summarizes the relative strengths and weaknesses of different types of tests.

	Unit	Functional	Integration/System
What is tested	One method/class	Several methods/classes	Large chunks of system
Rails example	Model specs	Controller specs	Cucumber scenarios
Preferred tool	RSpec	RSpec	Cucumber
Running time	Very fast	Fast	Slow
Error localization	Excellent	Moderate	Poor
Coverage	Excellent	Moderate	Poor

Use of mocks & Heavy stubs	Moderate	Little/none
---------------------------------------	----------	-------------

Figure 6.23: Summary of the differences among unit tests, functional tests, and integration or whole-system tests.

Summary

- Static and dynamic measures of coverage, including code-to-test ratio (reported by `rake stats`), C0 coverage (reported by SimpleCov), and C1–C2 coverage, measure the extent to which your test suite exercises different paths in your code.
- Unit, functional, and integration tests differ in terms of their running time, resolution (ability to localize errors), ability to exercise a variety of code paths, and ability to “sanity-check” the whole application. All three are vital to software assurance.

Self-Check 6.8.1. 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 6.8.2. 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.

Self-Check 6.8.3. Why is it usually a bad idea to make extensive use of **mock** or **stub** in Cucumber scenarios such as those described in Chapter 5?

 Cucumber is a tool for full-system testing and acceptance testing. Such testing is specifically intended to exercise the entire system, rather than “faking” certain parts of it as we have done using seams in 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. That topic is the subject of Exercise 6.1.)

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.

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. The Ruby gem Heckle performs mutation testing in conjunction with RSpec, but is unfortunately incompatible with Ruby 1.9, the version of the language we use in this book. As of this writing, no solid tool for Ruby 1.9 mutation testing has yet emerged, though given the importance of testing in the Ruby community, this will likely change soon.

Fuzz testing consists of throwing random data at your application and seeing what breaks. About 1/4 of common Unix utilities can be made to crash by fuzz testing, and Microsoft estimates that 20–25% of their bugs are found this way. *Dumb fuzzing* generates completely random data, while *smart fuzzing* includes knowledge about the app’s structure. For example, smart fuzzing for a Rails app might include randomizing the variables and values occurring in form postings or in URIs embedded in page views, creating URIs that are syntactically valid but might expose a bug. Smart fuzzing for SaaS can also include attacks such as cross-site scripting or SQL injection, which we’ll discuss in Chapter 12. [Tarantula](#) (a fuzzy spider that crawls your site) is a Ruby gem for fuzz-testing Rails applications.

Self-Check 6.9.1. 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 vs. glass-box (see Section 1.4), 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.



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.

STOP 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 code](#) 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).

Even 100% unit test coverage tells you nothing about interactions among classes. You still need to create tests to exercise the interactions between classes (functional or module testing) and to exercise complete paths through the application that touch many classes and change state in many places (integration testing). Conversely, integration tests touch only a tiny fraction of all possible application paths, and therefore exercise only a few behaviors in each method, so they are not a substitute for good unit test coverage to get assurance that your lower-level code is working correctly. A common 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.

⚠️ Pitfall: Undertested 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:

<http://pastebin.com/BvzjrzEr>

1

```
movie = mock('Movie', :pics => [mock('Picture', :format => 'gi
f')])  
2   Movie.count_pics(movie).should == 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 the Chapters 8 and 10 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 autotest prioritizes running tests related to recently-changed code. Therefore it's best to use `before :each` and `after :each` whenever possible.

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 `should` constructs.

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 he doesn't care about or change variable values to force the code to go down the suspected buggy path. For example, he 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, which can even happen automatically using autotest.

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.

6.12 To Learn More

- *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 documentation](#) 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.

P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN 0521880386. URL <http://www.amazon.com/Introduction-to-Software-Testing-Paul-Ammann/dp/0521880386>.

D. Chelimsky, D. Astels, B. Helmkamp, D. North, Z. Dennis, and A. Hellesoy. *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends (The Facets of Ruby Series)*. Pragmatic Bookshelf, 2010. ISBN 1934356379. URL <http://www.amazon.com/RSpec-Book-Behaviour-Development-Cucumber/dp/1934356379>.

M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004. ISBN 9780131177055. URL <http://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052>.

J. A. Whittaker, J. Arbon, and J. Carollo. *How Google Tests Software*. Addison-Wesley Professional, 2012. ISBN 0321803027. URL <http://www.amazon.com/Google-Tests-Software-James-Whittaker/dp/0321803027>.

Project 6.1. Complete the happy path of the Cucumber scenario started in Chapter [5](#) for retrieving movie info from TMDb. To keep the scenario Independent of the real TMDb service, you'll need to download and use the FakeWeb gem to "stub out" calls to the TMDb service.

Project 6.2. Our current controller specs only cover the happy path of testing `search_tmdb`: since `find_in_tmdb` can now raise an `InvalidKeyError`, the controller specs are incomplete because we don't test what the controller method does in that case. Use TDD to create specs and controller code to handle this case.

Project 6.3. In Section [6.3](#), we stubbed the method `find_in_tmdb` both to isolate the testing of the controller from other classes and because the method did not yet exist. How would such stubbing be handled in Java?

Project 6.4. Based on the specfile below, to what method(s) must instances of `Foo` respond in order for the tests to pass?

<http://pastebin.com/rxK7eucu>

```

1  require 'foo'
2  describe Foo do
3    describe "a new foo" do
4      before :each do ; @foo = Foo.new ; end
5      it "should be a pain in the butt" do
6        @foo.should be_a_pain_in_the_butt
7      end
8      it "should be awesome" do

```

```
9      @foo.should be_awesome
10     end
11     it "should not be nil" do
12       @foo.should_not be_nil
13     end
14     it "should not be the empty string" do
15       @foo.should_not == ""
16     end
17   end
18 end
```

Project 6.5. (Hard) In the previous exercise, explain why incorrect test behavior would result from rewriting line 15 as:

<http://pastebin.com/iyu1LEi1>

```
1   @foo.should != ""
```

Project 6.6. In Chapter 5, we created a “Find in TMDb” button on the index page of Rotten Potatoes that would post to `search_tmdb`, but we never wrote a spec that verifies that the button routes to the correct action. Write this spec using RSpec’s `route_to` assertion matcher and add it to the controller spec. (Hint: since this route doesn’t correspond to a basic CRUD action, you won’t be able to use the built-in RESTful URI helpers to specify the route, but you can use the `:controller` and `:action` arguments to `route_to` to specify the action explicitly.)

Project 6.7. Increase the C0 coverage of `movies_controller.rb` to 100% by creating additional specs in `movies_controller_spec.rb`.

Project 6.8. Write a sequence of test cases that will ultimately qualify emails as valid or not. Start with the simplest possible thing, but keep regressions as you add more features.

Project 6.9.

In 1999, the \$165 million Mars Climate Orbiter spacecraft burned up while entering the Martian atmosphere because one team working on thruster software had expressed thrust in metric (SI) units while another team working on a different part of the thruster software had expressed them in Imperial units. What kind(s) of correctness tests—unit, functional, or integration—would have been necessary to catch this bug?



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 OO language.

Programming is understanding. *Kristen Nygaard*

- 7.1 [DRYing Out MVC: Partials, Validations and Filters](#)
- 7.2 [Single Sign-On and Third-Party Authentication](#)
- 7.3 [Associations and Foreign Keys](#)
- 7.4 [Through-Associations](#)
- 7.5 [RESTful Routes for Associations](#)
- 7.6 [Composing Queries With Reusable Scopes](#)
- 7.7 [Fallacies and Pitfalls](#)
- 7.8 [Concluding Remarks: Languages, Productivity, and Beauty](#)
- 7.9 [To Learn More](#)
- 7.10 [Suggested Projects](#)

This chapter explores three sets of mechanisms for DRYing out your code, thereby making it more concise, beautiful and maintainable. Model validations and controller filters centralize what invariants must hold in order for a model object to be valid (for example, a movie must have a nonblank title) or for a controller action to proceed (for example, the user must be logged in as an admin). ActiveRecord Associations use Ruby language features to represent and manipulate relationships among different types of ActiveRecord models, while using relational-database functionality to represent these relationships as foreign-key associations. Finally, scopes let you encapsulate different ActiveRecord queries into composable “building blocks” that you can easily reuse to add new query functionality to your app. In each case, tastefully-chosen language features and framework architecture support DRY and concise app code.

We’ll focus our discussion of DRYness on the three elements of MVC, starting with Views.

As Section [11.6](#) explains, the partial is also the basic unit of view updating for JavaScript-enabled pages.



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. For example, in Chapter [3](#), Figure [4.13](#) noted the similarities between the `new` and `edit` actions, both of whose views display the same form for entering movie information—not very DRY. Figure [7.1](#) captures this common view code in a partial and shows how to modify the existing `new.html.haml` and `edit.html.haml` to use it.

<http://pastebin.com/G1c32mQW>

```
1 -# in _movie_form.html.haml (the partial)
2
3 = label :movie, :title, 'Title'
4 = text_field :movie, 'title', @movie.title
5
6 = label :movie, :rating, 'Rating'
7 = select :movie, :rating, ['G','PG','PG-13','R','NC-17'], @movie.rating
8
9 = label :movie, :release_date, 'Released On'
10 = date_select :movie, :release_date, @movie.release_date
```

<http://pastebin.com/ZNKAws0T>

```
1 -# new.html.haml using partial
2
3 %h2 Create New Movie
4
5 = form_tag '/movies', :method => :post do
6   = render :partial => 'movie_form'
7   = submit_tag 'Save Changes'
```

<http://pastebin.com/pRWLLbNk>

```
1 -# edit.html.haml using partial
2
3 %h2 Edit Existing Movie
4
5 = form_tag movie_path(@movie), :method => :put do
6   = render :partial => 'movie_form'
7   = submit_tag 'Update Movie Info'
```

Figure 7.1: A partial (top) that captures the form elements common to both the `new` and `edit` templates. The modified `new` and `edit` templates both use the partial (line 6 in both snippets), but line 5 is different between the two templates because `new` and `edit` submit to different actions. (Use the Pastebin service to copy-and-paste this code.)



Partials rely heavily on convention over configuration. Their names must begin with an underscore (we used `_movie_form.html.haml`) which is *absent from* the code that references the partial. If a partial is not in the same directory as the view that uses it, '`layouts/footer`' would cause Rails to look for `app/views/layouts/_footer.html.haml`. A partial can access all the same instance variables as the view that includes it. A particularly nice use of a partial is to render a table or other collection in which all elements are the same, as Figure 7.2 demonstrates.

<http://pastebin.com/4CLiWA2t>

```
1 -# A single row of the All Movies table
2 %tr
3   %td= movie.title
4   %td= movie.rating
5   %td= movie.release_date
6   %td= link_to "More about #{movie.title}", movie_path(movie)
)
```

Figure 7.2: If this partial is saved as `views/movies/_movie.html.haml`, lines 12–17 of our original

`index.html.haml` view (Figure 4.5) can be replaced with the single line `render :partial=>'movie', :collection=>@movies`. By convention over configuration, the singularized name of the `:collection` argument is available as a local variable in the partial, with Rails using `each` to retrieve `@movie`'s elements in turn.

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. As another example, perhaps we want to allow any user to add new movies, but only allow special "admin" users to delete movies.

Didn't the user choose the rating from a menu? Yes, 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.

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.

<http://pastebin.com/2GtWshSb>

```
1 class Movie < ActiveRecord::Base
2   RATINGS = %w[G PG PG-13 R NC-17] # %w[] shortcut for array of strings
3   validates :title, :presence => true
4   validates :release_date, :presence => true
5   validate :released_1930_or_later # uses custom validator below
6   validates :rating, :inclusion => { :in => RATINGS }, :unless => :grandfathered?
7   def released_1930_or_later
8     errors.add(:release_date, 'must be 1930 or later') if
9       self.release_date < Date.parse('1 Jan 1930')
10    end
11    def grandfathered? ; self.release_date >= @@grandfathered_date ; end
12
13 end
14 # try in console:
```

```
15 m = Movie.new(:title => '', :rating => 'RG', :release_date => '1929-01-01')
16 # force validation checks to be performed:
17 m.valid? # => false
18 m.errors[:title] # => ["can't be blank"]
19 m.errors[:rating] # => ["is not included in the list"]
20 m.errors[:release_date] # => ["must be 1930 or later"]
21 m.errors.full_messages # => ["Title can't be blank", "Rating is not included in the list", "Release date must be 1930 or later"]
```

Figure 7.3: Lines 3–5 use predefined validation behaviors in [ActiveModel::Validations::ClassMethods](#). Lines 6–10 show how you can create your own validation method, which receives the object to be validated as an argument and adds error messages describing any problems. Note that we first validate the presence of `release_date`, otherwise the comparison in line 9 could fail if `release_date` is nil. (We could also have checked this inside the `released_1930_or_later` method, but having two validations emphasizes that both conditions need to be true.)

Model validations, like migrations, are expressed in a mini-DSL embedded in Ruby, as Figure 7.3 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](#) object associated with each model; this object is returned by the the instance method `errors`. As Figure 7.3 shows, you can inquire about errors on individual attributes (lines 18–20) or use `full_messages` to get all the errors as an array of strings. When writing your own custom validations, as in lines 7–10, you can use `errors.add` to add an error message associated with either a specific invalid attribute of the object or the object in general (by passing `:base` rather than an attribute name).

The example also demonstrates that validations can be conditional. For example, line 6 in Figure 7.3 ensures that the movie's rating is a valid one, *unless* the movie was released before the ratings system went into effect (in the USA, 1 November 1968), in which case we don't need to validate the rating.

 We can make use of validation to replace the dangerous `save!` and `update_attributes!` in our controller actions with their safer versions, `save` and `update_attributes`, which fail if validation fails. Figure 7.4 shows how to modify our controller methods to be more idiomatic by taking advantage of this. Modify your `create` and `update` controller actions to match the figure, modify `app/models/movie.rb` to include the validations in Figure 7.3, and then try adding a movie that violates one or more of the validations.

<http://pastebin.com/CM6ntZzK>

```
1 # replaces the create method in controller:  
2 def create  
3   @movie = Movie.new(params[:movie])  
4   if @movie.save  
5     flash[:notice] = "#{@movie.title} was successfully created."  
6   redirect_to movies_path  
7 else  
8   render 'new' # note, 'new' template can access @movie's  
9 field values!  
10 end  
11 # replaces the update method in controller:  
12 def update  
13   @movie = Movie.find params[:id]  
14   if @movie.update_attributes(params[:movie])  
15     flash[:notice] = "#{@movie.title} was successfully updated."  
16   redirect_to movie_path(@movie)  
17 else  
18   render 'edit' # note, 'edit' template can access @movie'  
s field values!  
19 end  
20 end
```

Figure 7.4: If `create` or `update` fails, we use an explicit `render` to re-render the form so the user can fill it in again. Conveniently, `@movie` will be made available to the view and will hold the values the user entered the first time, so the form will be prepopulated with those values, since (by convention over configuration) the form tag helpers used in `_movie_form.html.haml` will use `@movie` to populate the form fields as long as it's a valid `Movie` object.

Of course, it would be nice to provide the user an informative message specifying what went wrong and what she should do. This is easy: within a view we *can* get the name of the controller action that called for the view to be rendered, so we can include it in the error message, as line 5 of the following code shows.

<http://pastebin.com/YGqvV8pg>

```
1  -# insert at top of _movie_form.html.haml
2
3  - unless @movie.errors.empty?
4      #warning
5
6      Errors prevented this movie from being #{controller.action
7      _name}d:
8      %ul
9          - @movie.errors.full_messages.each do |error|
10             %li= error
```

Screencast [7.1.1](#) digs deeper and explains how Rails views take advantage of being able to query each model attribute about its validity separately, to give the user better visual feedback about which specific fields caused the problem.

Screencast 7.1.1: [How model validations interact with controllers and views](#)

The form helpers in our views use the `errors` object to discover which fields caused validation errors and apply the special CSS class `field-with-errors` to any such fields. By including selectors for that class in `app/assets/stylesheets/application.css`, we can visually highlight the affected fields for the user's benefit.

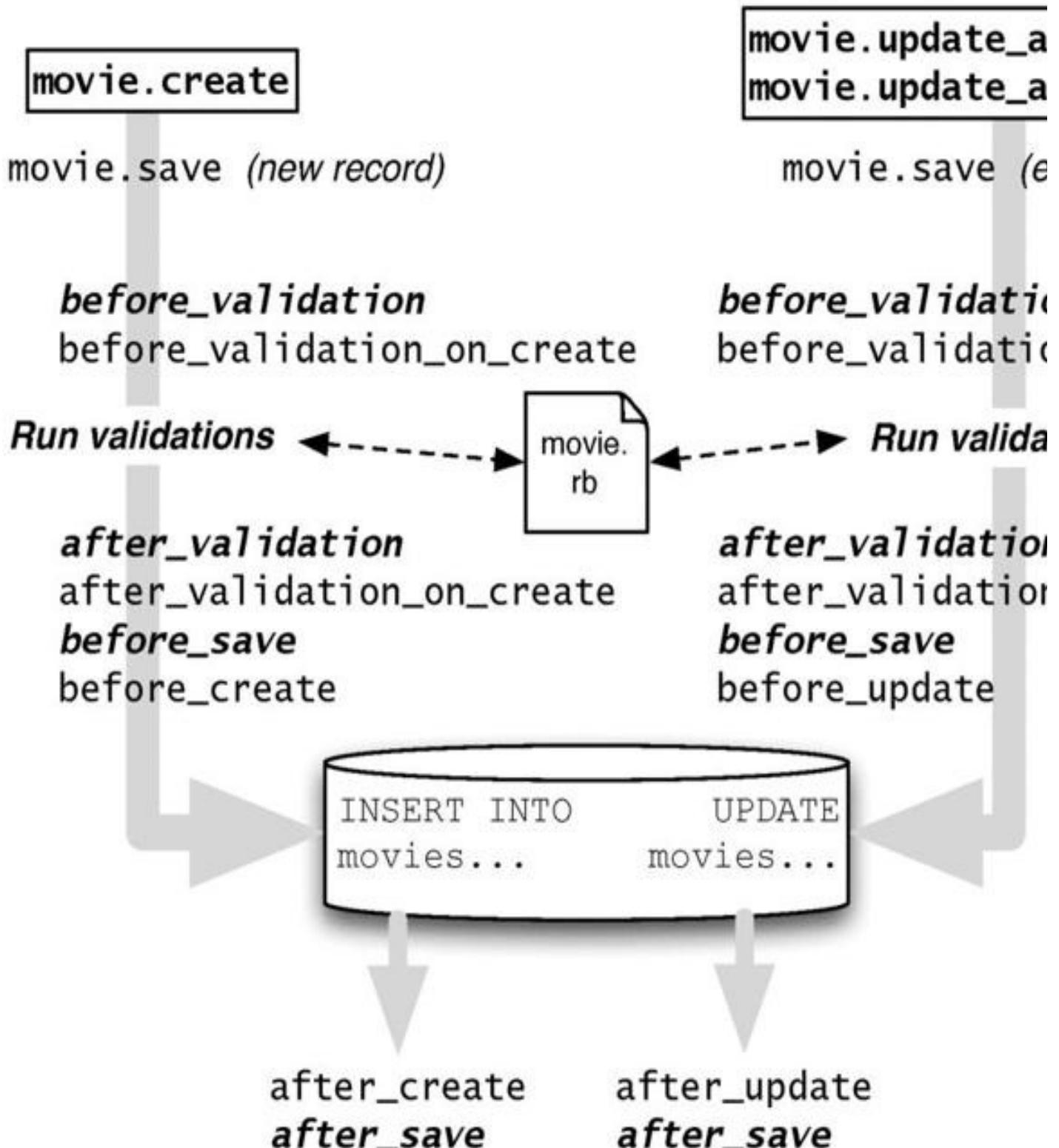


Figure 7.5: The various points at which you can “hook into” the lifecycle of an ActiveRecord model object. All ActiveRecord operations that modify the database (`update`, `create`, and so on) all eventually call `save`, so a `before_save` callback can intercept every change to the database. See [this Rails Guide](#) for additional details and examples.

<http://pastebin.com/w4c1nm1q>

```
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 m.title # => "Star Wars"

```

Figure 7.6: 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` (not necessarily the right behavior for movie titles, but useful for illustration).

In fact, validations are just a special case of a more general mechanism, [Active Record lifecycle callbacks](#), which allow you to provide methods that “intercept” a model object at various relevant points in its lifecycle. Figure 7.5 shows what callbacks are available; Figure 7.6 illustrates how to use this mechanism to “canonicalize” (standardize the format of) certain model fields before the model is saved. We will see another use of lifecycle callbacks when we discuss the Observer design pattern in Chapter 10 and caching in Chapter 12.

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 7.7 shows a filter that enforces that a valid user is logged in. In Section 7.2 we will show how to combine the before-filter with the other “moving parts” involved in dealing with logged-in users.

<http://pastebin.com/MHuGdTUT>

```

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 find_by_id(nil) returns nil
6     @current_user ||
= Moviegoer.find_by_id(session[:user_id])
7     redirect_to login_path and return unless @current_user
8   end
9 end

```

Figure 7.7: 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, about which more in Section 7.2. (`and` is just like `&&` but has lower precedence, thus `((redirect_to login_path) and (return)) unless...`)

Filters normally apply to all actions in the controller, but `:only` can be used to specify that the filter only guards certain actions, while `:except` can be used to specify that some actions are exempt. Each takes an array of action names. 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 specify actions to run before and after, as you might do for auditing or timing. Around-filters use `yield` to actually do the controller action:

<http://pastebin.com/VMBKsbBs>

```

1   # somewhat contrived example of an around-filter
2
3   around_filter :only => ['withdraw_money', 'transfer_money'] do
4     # log who is trying to move money around
5     start = Time.now
6     yield    # do the action
7     # note how long it took
8     logger.info params
9     logger.info (Time.now - start)
9 end

```

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. View form helpers can use this information to apply special CSS styles to fields whose values failed validation, cleanly separating the responsibility for detecting an error from the responsibility for displaying that error to the user.
- Controller filters let you collect conditions affecting many controller actions in a single place by defining a method that always runs before those actions. A filter declared in a controller affects all actions in that controller, and a filter declared in `ApplicationController` affects all actions in all controllers, unless `:only` or `:except` are specified.

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***. Ruby doesn't support full AOP, which would allow you to specify arbitrary pointcuts along with what advice applies to each. But Rails uses Ruby's dynamic language features to define convenient pointcuts such as the AR model lifecycle, which supports validations and other lifecycle callbacks, and join points around controller actions, which support the use of before- and after-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 7.1.1. Why didn't the Rails designers choose to trigger validation when you first instantiate one using `Movie#new`, rather than waiting until you try to persist the object?

❷ 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 7.1.2. 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?

❸ 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. 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 her 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 **provenance** can be verified using cryptography.

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.

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.



Facebook was an early example of SSO.



Figure 7.8: Third-party authentication enables SSO by allowing a SaaS app to request that the user authenticate himself via a third-party provider. Once he has done so, the provider sends a token to the requesting app proving that the user authenticated himself 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.

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 7.8 shows how **third-party authentication** solves this problem using Rotten Potatoes and Twitter as an example. First, the app requesting authentication (Rotten Potatoes) 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.

The SSO process usually begins with a link or button the user must click. That link takes the user to a login page served securely *by the provider*; depending on implementation, the login page may be a popup, an [HTML frame or iframe element](#), or a regular page served by the provider's site. The user is then given the chance to login to the provider and decide whether to grant the app the requested privileges. Critically, this interaction takes place entirely between the user and the provider: the requesting app has no access to any part of this interaction. Once authentication succeeds, the provider generates a **callback** to the requesting app including an [access token](#)—a string created using cryptographic techniques, which allows the provider to verify that it 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:

It can trust that the user has proven her identity to the provider, and optionally record the provider's persistent unique ID for that user (usually part of the access token) for future reference. For example, Armando Fox's unique user-id (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.

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 7.9.

<http://pastebin.com/UXQZiBY0>

```
1 rails generate model Moviegoer name:string provider:string u  
id:string
```

<http://pastebin.com/LyD2qxRA>

```
1 # Edit app/models/moviegoer.rb to look like this:  
2 class Moviegoer < ActiveRecord::Base  
3   include ActiveModel::MassAssignmentSecurity  
4   attr_protected :uid, :provider, :name # see text for explanation  
5   def self.create_with_omniauth(auth)  
6     Moviegoer.create!(  
7       :provider => auth["provider"],  
8       :uid => auth["uid"],  
9       :name => auth["info"]["name"])  
10    end  
11  end
```

Figure 7.9: 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.

<http://pastebin.com/pBuULmVY>

```
1 match 'auth/:provider/callback' => 'sessions#create', :as =>  
'login'  
2 match 'logout' => 'sessions#destroy'
```

<http://pastebin.com/QfYMbdKX>

```
1 class SessionsController < ApplicationController  
2   # user shouldn't have to be logged in before logging in!  
3   skip_before_filter :set_current_user  
4   def create  
5     auth = request.env["omniauth.auth"]  
6     user = Moviegoer.find_by_provider_and_uid(auth["provider"],  
"uid") ||  
7     Moviegoer.create_with_omniauth(auth)  
8     session[:user_id] = user.id
```

```
9     redirect_to movies_path
10    end
11    def destroy
12      session.delete(:user_id)
13      flash[:notice] = 'Logged out successfully.'
14      redirect_to movies_path
15    end
16 end
```

<http://pastebin.com/sGpXM3kW>

```
1 #login
2   - if @current_user
3     %p.welcome Welcome, #{@current_user.name}!
4     = link_to 'Log Out', logout_path
5   - else
6     %p.login= link_to 'Log in with your Twitter account', login_path
```

Figure 7.10: The moving parts in a typical Rails app authentication flow. Top (a): Two routes that follow the OmniAuth gem's convention for mapping the `create` and `destroy` actions in a separate `SessionsController`; for convenience we ask for `login_path` and `login_url` helpers for the first route. Middle (b): Line 3 skips the `before_filter` that we added to `ApplicationController` in Figure 7.7, to avoid an infinite loop when the user first tries to login. (Another strategy would be to delete line 7 in Figure 7.7 and let each controller specify its own filters for what to do if `@current_user` is nil.) Upon successful login of a given user, the `create` action remembers that user's primary key (ID) in the session until the `destroy` action is called to forget it. Bottom (c): The `@current_user` variable (set in line 6 of `ApplicationController`, Figure 7.7) can be used by a login partial to display an appropriate message. The partial could be included from `application.html.haml` with `render :partial=>'sessions/login'`.



There are three aspects to managing third-party authentication in Rails. The first is how to actually authenticate the user via a third party. We will use the excellent [OmniAuth](#) gem, which abstracts away the entire process in Figure 7.8 by allowing developers to create a *strategy* for each third-party auth provider. A strategy handles all the interactions with the authentication provider (steps 2–4 in Figure 7.8) and ultimately performs an HTTP POST to the URI

/auth/provider/callback in your app. The data included with the POST indicate the success or failure of the authentication process, and if successful, may include authentication token(s) that your app can use to get additional information about the logged-in user. As of this writing, strategies are available for Facebook, Twitter, Google Apps, and many others, each available as a gem named `omniauth-provider`. We will use Twitter as an example, so add `gem 'omniauth-twitter'` to your Gemfile and run `bundle install --without production` as usual.

Figure 7.10(a) shows how to add the route that the OmniAuth strategy expects to use when it completes the authentication with Twitter.

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 7.10(b) shows how we can “create” a session when a user successfully authenticates herself (lines 3–9) and “destroy” it when she logs out (lines 11–15). 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 7.10(c) 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 `moviegoer` 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 7.9(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 `find_by_provider_and_uid` query in line 6 of the sessions controller (Figure 7.10(b)) will return `nil`, so `Moviegoer.create_with_omniauth` (Figure 7.9(b), lines 5–10) 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 7.2.1 shows the user experience associated with this code.

Screencast 7.2.1: [Logging into RottenPotatoes with Twitter](#)

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 have just created a security vulnerability. So far we've exploited the convenience of "mass assignment" from the `params[]` hash to an ActiveRecord object, as when we write `@movie.update_attributes(params[:movie])` in `MoviesController#update`. But 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 `params[:moviegoer][:uid]` and so on? Figure 7.9(b), line 3, shows how to "protect" such sensitive attributes from being mass-assigned from `params`. The more restrictive `attr_accessible` arranges for *only* the named attributes to be modifiable through mass assignment from `params[]` (or for that matter from any hash). We will return to this topic in Section 12.8 when discussing how to defend customer data in more detail.

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 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 [OmniAuth](#)), a session is created by storing the authenticated user's `id` (primary key) in the `session[]`. When she signs out, the session is destroyed by deleting that information from the `session[]`.
- Use `attr_protected` and `attr_accessible` to identify model attributes that are "sensitive" and should be excluded from mass assignment via a hash, such as user ID information used for session management or authentication.

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) Bob 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 7.2.1. Briefly describe how RottenPotatoes could let you log in with your Twitter ID without you having to reveal your Twitter password to RottenPotatoes.

Green circle icon: 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 7.2.2. True or false: If you log in to RottenPotatoes using your Twitter ID, RottenPotatoes becomes capable of tweeting using your Twitter ID.

Green circle icon: 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.

An *association* is a logical relationship between two types of entities in a software architecture. For example, we might add **Review** and **Moviegoer** classes to Rotten Potatoes to allow individual users to write reviews of their favorite movies; we could do this by establishing a one-to-many association from reviews to movies (each review is about exactly one movie) and from reviews to moviegoers (each review is authored by exactly one moviegoer). Figure 7.11 shows these associations using one type of [Unified Modeling Language \(UML\)](#) diagram. We will see more examples of UML in Chapter 10.



Figure 7.11: 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.)

In Rails parlance, Figure 7.11 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 7.12 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.

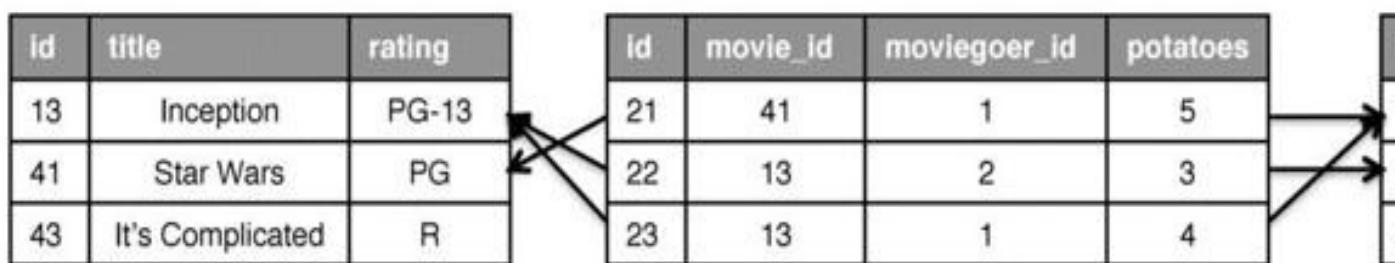


Figure 7.12: In this figure, Alice has given 5 potatoes to Star Wars and 4 potatoes to Inception, Bob has given 3 potatoes to Inception, Carol hasn’t provided any reviews, and no one has reviewed It’s Complicated. For brevity and clarity, the other fields of the movies and reviews tables are not shown.

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: <http://pastebin.com/Qd334nG8>

```
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 [7.13](#).

<http://pastebin.com/419hfNTY>

```
1 # it would be nice if we could do this:  
2 inception = Movie.find_by_title('Inception')  
3 alice,bob = Moviegoer.find(alice_id, bob_id)  
4 # alice likes Inception, bob hates it  
5 alice_review = Review.new(:potatoes => 5)  
6 bob_review = Review.new(:potatoes => 2)  
7 # a movie has many reviews:  
8 inception.reviews = [alice_review, bob_review]  
9 inception.save!  
10 # a moviegoer has many reviews:  
11 alice.reviews << alice_review  
12 alice.save!  
13 # can we find out who wrote each review?  
14 inception.reviews.map { |  
r| r.moviegoer.name } # => ['alice','bob']
```

Figure 7.13: A straightforward implementation of associations would allow us to refer directly to associated objects, even though they're stored in different database tables.

Rails' [ActiveRecord::Associations](#) module supports exactly this design, as we'll learn by doing. Apply the code changes in Figure [7.14](#) as directed in the caption, and you should then be able to start rails console and successfully

execute the examples in Figure 7.13.



<http://pastebin.com/Ch2FTEzA>

```
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 up
5     create_table 'reviews' do |t|
6       t.integer    'potatoes'
7       t.text       'comments'
8       t.references 'moviegoers'
9       t.references 'movies'
10      end
11    end
12    def down ; drop_table 'reviews' ; end
13  end
```

<http://pastebin.com/GBTCH47f>

```
1 class Review < ActiveRecord::Base
2   belongs_to :movie
3   belongs_to :moviegoer
4   attr_protected :moviegoer_id # see text
5 end
```

<http://pastebin.com/WHuX2LcV>

```
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 7.14: 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.

How does this work? Since everything in Ruby is a method call, we know that Line 8 in Figure 7.13 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, `inception.save!` 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*. (Of course, if we had actually modified any of *Inception*'s attributes, `inception.save!` would try to persist them; but because `save!` is transactional—that is, it's all-or-nothing—if the `save!` fails then *every* aspect of it fails, so neither the changes to *Inception* nor its associated Reviews would be saved.)

<code>m.reviews</code>	returns an Enumerable 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<<r2</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>r = m.reviews.create(:potatoes=>5)</code>	Review object whose movie_id is preset to indicate that it belongs to <code>m</code> . Arguments are the same as for <code>Review.new</code> .
	Like <code>build</code> but saves the object immediately (analogous to the difference between <code>new</code> and <code>save</code>).
<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 7.15: 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 documentation](#) for a full list. Method names of association methods follow convention over configuration based on the name of the associated model.

Figure 7.15 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 an collection. That is, you can use all the collection idioms of Figure 3.7 on it—iterate over its elements with `each`, use functional idioms like `sort`, `search` and `map`, and so on, as in lines 8, 11 and 14 of Figure 7.13.

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.

`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.

Associations 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 ActiveRecord vs. Data Mapper

The concept of associations is architectural; the use of foreign keys to represent them is an implementation choice, and the Active Record architectural pattern, which Rails' ActiveRecord library implements, can bridge the two by providing methods for automatic traversal when a relational database is used. But foreign-key-based associations can become so complex that the overhead of managing them limits the scalability of relational databases, which are already the first bottleneck in the 3-tier architecture of Figure 2.7. One way to avoid this pitfall is to use the Data Mapper architectural pattern (see Figure 7.16), in which a Mapper class defines how each model and its associations are represented in the storage system and provides its own code to traverse them. Since DataMapper doesn't rely on foreign key support in the underlying storage system, it can use so-called "NoSQL" storage systems, such as [Cassandra](#), [MongoDB](#), and [CouchDB](#), which omit all but the simplest foreign key support in order to achieve superior horizontal scalability far beyond most RDBMSs. Indeed, you can deploy Ruby-based SaaS apps on [Google AppEngine](#) if they use a Google-provided DataMapper library rather than ActiveRecord.

Active Record

Moviegoer
firstname
lastname
age
<i>create()</i>
<i>read()</i>
<i>update()</i>
<i>delete()</i>

Movie
title
rating
created_on
description
<i>create()</i>
<i>read()</i>
<i>update()</i>
<i>delete()</i>

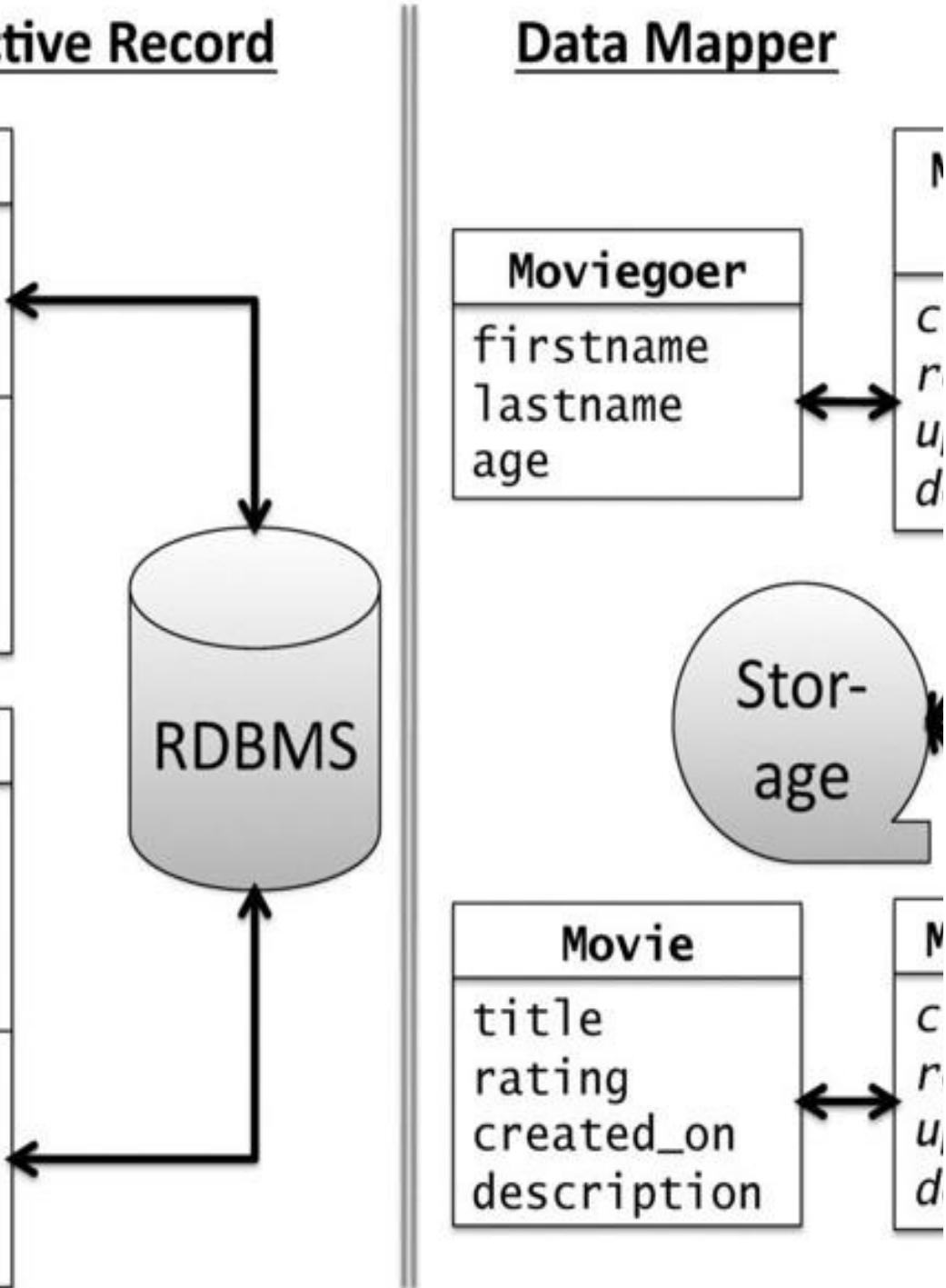


Figure 7.16: In the Active Record design pattern (left), used by Rails, the model object itself knows how it's stored in the persistence tier, and how its relationship to other types of models is represented there. In the Data Mapper pattern (right), used by Google AppEngine, PHP and Sinatra, a separate 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 10.

Self-Check 7.3.1. In Figure 7.14(a), why did we add foreign keys ([references](#)) only to the reviews table and not to the moviegoers or movies tables?

- ➊ 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 7.3.2. In Figure 7.15, 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.

Referring back to Figure 7.11, 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 14 in Figure 7.13 essentially answers the second question.

<http://pastebin.com/LwQAgaw7>

```
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.find_by_name('Alice')
8 alice_movies = alice.movies
9 # MAY work, but a bad idea - see caption:
10 alice.movies << Movie.find_by_name('Inception') # Don't do this!
```

Figure 7.17: 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 lines 9 and 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.

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 7.17 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 7.12, 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 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 7.3, the SQL query might look like this:

<http://pastebin.com/VpG61Cnn>

```
1   SELECT movies.*  
2  
3   FROM movies JOIN reviews ON movies.id = reviews.movie_id  
4       JOIN moviegoers ON moviegoers.id = reviews.moviegoer_id  
5   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 these two sections, though, is not just to explain how to use associations, but to help you appreciate the elegant use of duck typing and metaprogramming that makes them possible. In Figure 7.14(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 7.13. 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. Just like `attr_accessor`, `has_many` is not a declaration, but an actual method call that does all of this work at runtime, adding a slew of new model instance methods to

help manage the association.



Associations are one of the most feature-rich aspects of Rails, so take a good look at the full [documentation](#) 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, for example, to ensure that

a review is only saved after it has been associated with some movie:
<http://pastebin.com/Q4R9hXGb>

```
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
5   # valid
6   # in order for the review to be valid:
7   validates_associated :movie
8 end
```

-
- 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 Reviews 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 7.4.1. 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.

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 Section [7.2](#). But what about the movie?

Let’s think about this problem from the BDD point of 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 control, we 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 Detail 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, 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: <http://pastebin.com/cmB747Ta>

```
1 # in routes.rb, change the line 'resources :movies' to:  
2 resources :movies do
```

```
3     resources :reviews  
4   end
```



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*. Make the above changes to `routes.rb` and try `rake routes`, comparing the output to the simple routes introduced in Chapter 4. Figure 7.18 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).



Helper method	RESTful Route and action	
<code>movie_reviews_path(m)</code>	GET	index
	<code>/movies/:movie_id/reviews</code>	
<code>movie_review_path(m)</code>	POST	create
	<code>/movies/:movie_id/reviews</code>	
<code>new_movie_review_path(m)</code>	GET	new
	<code>/movies/:movie_id/reviews</code>	
	<code>/new</code>	
<code>edit_movie_review_path(m,r)</code>	GET	edit
)	<code>/movies/:movie_id/reviews</code>	
	<code>/:id/edit</code>	
<code>movie_review_path(m,r)</code>	GET	show
	<code>/movies/:movie_id/reviews</code>	
	<code>/:id</code>	
<code>movie_review_path(m,r)</code>	PUT	update
	<code>/movies/:movie_id/reviews</code>	
	<code>/:id</code>	
<code>movie_review_path(m,r)</code>	DELETE	destroy
	<code>/movies/:movie_id/reviews</code>	

/:id

Figure 7.18: Specifying nested routes in `routes.rb` also provides nested URI helpers, analogous to the simpler ones provided for regular resources. Compare this table with Figure 4.6 in Chapter 4.

<http://pastebin.com/qwzV1tZb>

```
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.find_by_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
20    # get
21    # assigned by the mass-
22    # assignment from params[:review], we set it
23    # by using the << method on the association. We could also
24    # set it manually with review.moviegoer = @current_user.
25    @current_user.reviews << @movie.reviews.build(params[:review])
26  end
27 end
```

<http://pastebin.com/YSQsXt47>

```
1 %h1 New Review for #{@movie.name}
2
3 = form_tag movie_review_path(@movie) do
4   %label How many potatoes:
5   = select_tag 'review[potatoes]', options_for_select(1..5)
6   = submit_tag 'Create Review'
7
```

Figure 7.19: 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 URI. Bottom (b): A possible Haml view template for creating a new review, that is, app/views/reviews/new.html.haml.

Figure 7.19 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, **@current_user** is set (that is, someone is logged in and will “own” the new review) and that the movie captured from the route (Figure 7.18) as **params[:movie_id]** exists in 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 7.18 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. Finally, since in Figure 7.14(b) we declared **moviegoer_id** as a protected attribute, it cannot be assigned to the new review via the mass assignment from **params** (line 23 in the **create** action), so instead we set it by building the review from its other “owner” **@current_user**.

We could link to the page for creating a new review using something like the following on a movie’s Details page:

<http://pastebin.com/RfYpnTKx>

```
1   = link_to 'Add Review', new_movie_review_path(@movie)
```

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 was 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 7.5.1. 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.



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.



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.

`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.

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 `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.

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.

7.9 To Learn More

- The ActiveRelation part of Rails, which manipulates ActiveRecord associations and generates SQL queries, was completely redesigned for Rails 3 and is amazingly powerful. This [guide](#) has many examples beyond those introduced in this chapter that will help you use the database effectively, which as we'll see in Chapter 12 is critical to successful operations.
- The [Guides section of the Rails website](#) includes useful guides on a variety of Rails topics including debugging, managing the configuration of your app, and more.
- A concise [review of associations basics](#) is in the Guides section of the Rails website.
- *The Rails 3 Way* ([Fernandez 2010](#)) is an encyclopedic reference to all aspects of Rails 3, including the extremely powerful mechanisms that support associations.
- *jQuery: Novice to Ninja* ([Castledine and Sharkey 2010](#)) is a more thorough treatment of the jQuery framework's extensive features with many examples of how to use them.

E. Castledine and C. Sharkey. *jQuery: Novice to Ninja*. SitePoint Books, 2010.

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>.

O. Fernandez. *Rails 3 Way, The (2nd Edition) (Addison-Wesley Professional Ruby Series)*. Addison-Wesley Professional, 2010. ISBN 0321601661. URL <http://www.amazon.com/Rails-Way-Addison-Wesley-Professional-Ruby/dp/0321601661>.

Filters and authentication:

Project 7.1. Extend the example in Section [7.2](#) to allow authentication via Facebook Connect.

Project 7.2. Extend your solution to Exercise [7.1](#) to allow an authenticated user to “link” two accounts. That is, if Alice has previously logged in with Twitter and subsequently logs in with Facebook, she should be able to “link” the two accounts so that in the future, logging in with either one will authenticate her as the same principal. **Hint:** Consider creating an additional model **Identity** that has a many-to-one relationship to **Moviegoer**.

Project 7.3. In the README for the OmniAuth plugin, the author gives the following example code showing how to integrate OmniAuth into a Rails app:

<http://pastebin.com/G3iFdCFn>

```
1  class SessionsController < ApplicationController
2    def create
3      @user = User.find_or_create_from_auth_hash(auth_hash)
4      self.current_user = @user
```

```
5      redirect_to '//'
6    end
7  protected
8    def auth_hash
9      request.env['omniauth.auth']
10   end
11 end
```

In line 3, why do you think the author calls

The **auth_hash** method (lines 8–10) has the trivial task of returning whatever OmniAuth returned as the result of trying to authenticate a user. Why do you think the author placed this functionality in its own method rather than just referencing **request.env['omniauth.auth']** directly in line 3?

Associations and RESTful application architecture:

Project 7.4. Extend the controller code in Figure 7.19 with **edit** and **update** methods for reviews. Use a controller filter to ensure that a user can only edit or update her own reviews.



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*

- 8.1 [What Makes Code "Legacy" and How Can Agile Help?](#)
- 8.2 [Exploring a Legacy Codebase](#)
- 8.3 [Establishing Ground Truth With Characterization Tests](#)
- 8.4 [Metrics, Code Smells, and SOFA](#)
- 8.5 [Method-Level Refactoring: Replacing Dependencies With Seams](#)
- 8.6 [Fallacies and Pitfalls](#)
- 8.7 [Concluding Remarks: Continuous Refactoring](#)
- 8.8 [To Learn More](#)
- 8.9 [Suggested Projects](#)

Out of every dollar spent on software, 36% is spent on enhancements, 10% on fixing bugs, 11% on adapting to environmental changes such as new library versions or API changes, and 3% on *refactoring* to make the software more maintainable. In total, therefore, about 60% of software expenses is devoted to software maintenance, so your first job is more likely to involve improving existing code than creating a brand-new system from a clean slate. In Chapters 5 and 6 we looked at disciplined ways to evolve new code. Although thorough formal documentation of legacy systems may be lacking or inaccurate, the Agile techniques we already know can be pressed into service to

help understand the structure of legacy software and create a foundation for extending and modifying it with confidence. We will describe what good code looks like and why, and show how to apply refactoring techniques to legacy code both to make it more testable (and therefore modifiable with confidence) and to leave it in better shape than we found it for the next developers.



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 how to apply Agile techniques to enhance and modify legacy code. Figure 8.1 highlights this topic in the context of the overall Agile lifecycle.

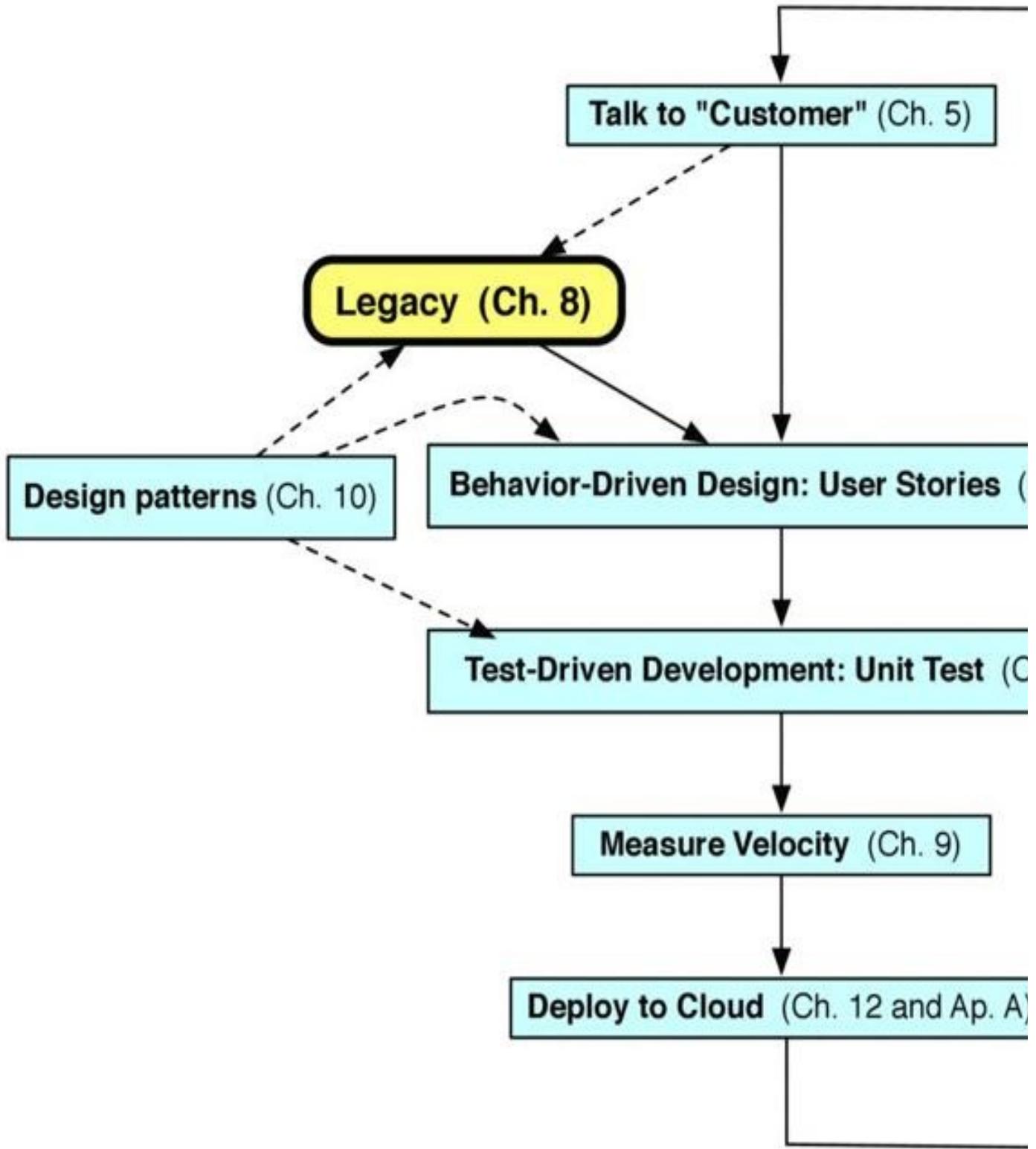


Figure 8.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter covers how Agile techniques can be helpful when enhancing legacy apps.

Maintainability is the ease with which a product can be improved. In software engineering, maintenance consists of four categories:

- 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.

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, because 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 8.2 lists.

Highly-readable unit, functional and integration tests ([Chapter 6](#))

Lo-fi UI mockups and Cucumber-style user stories ([Chapter 5](#))

Photos of whiteboard sketches about the application architecture, class relationships, etc. ([Section 8.2](#))

Git commit log messages ([Chapter 9](#))

Comments and RDoc-style documentation embedded in the code ([Section 8.1](#))

Archived email, wiki/blog, notes, or video recordings of code and design reviews, for example in [Campfire](#) or [Basecamp](#) ([Chapter 9](#))

Figure 8.2: 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.

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 undertested code, we need to "bootstrap" ourselves into the desired situation in four steps:

Identify the **change points**, or places where you will need to make changes in the legacy system. Section [8.2](#) 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. If necessary, add **characterization tests** that capture how the code works now, to establish a baseline "ground truth" before making any changes.

Section [8.3](#) explains how to do this using tools you're already familiar with. Determine whether the change points require **refactoring** to make the existing code more testable or accommodate the required changes, for example, by breaking dependencies that make the code hard to test.

Section [8.5](#) introduces a few of the most widely-used techniques from the many catalogs of refactorings that have evolved as part of the Agile movement.

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 [5](#) and [6](#).

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 wikibook](#). The default HTML output from RDoc can be seen, for example, in the [Rails documentation](#). 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 8.1.1.

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.

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.

Just as we explored SaaS architecture in Chapter 2 using height as an analogy, we can follow some "outside-in" steps to understand the structure of a legacy app at various levels:

Check out a scratch branch to run the app in a development environment

Learn and replicate the user stories, working with other stakeholders if necessary

Examine the database schema and the relationships among the most important classes

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 you never intend to check back in 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.
- Cloning the database eliminates possible differences in behavior between production and development resulting from differences in database implementations, difference 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 database](#) using the steps in Figure 8.3.

```
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  
9
```

Figure 8.3: You can create an empty development database that has the same schema as the production database and then populate it with fixtures. Although Chapter [6](#) 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.

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 [8.3](#) 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 [8.4](#) shows a simplified Unified Modeling Language (UML) class diagram generated by the `railroady` 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 [8.4](#), 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.

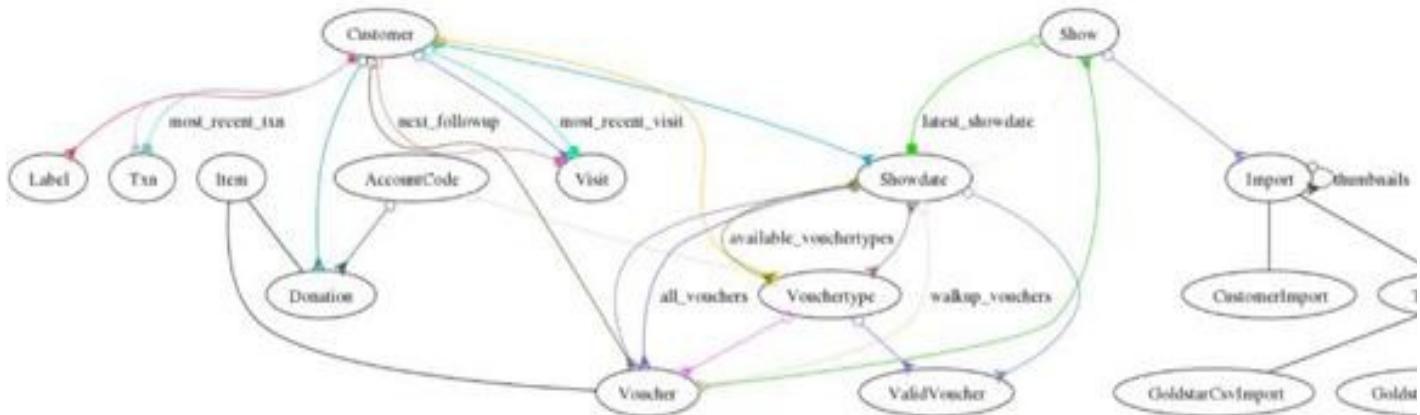


Figure 8.4: This simplified Unified Modeling Language (UML) class diagram, produced automatically by the railroady gem, shows the models in a Rails app that manages ticket sales, donations, and performance attendance for a small theater. Edges with arrowheads or circles show relationships between classes: a `Customer` has many `Visits` and `Vouchers` (open circle to arrowhead), has one `most_recent_visit` (solid circle to arrowhead), and has many `Labels` (arrowhead to open circle). Plain edges show inheritance: `Donation` and `Voucher` are subclasses of `Item`. (All of the important classes here inherit from `ActiveRecord::Base`, but railroady draws only the app's classes.) We will see other types of UML diagrams in Chapter [10](#).

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 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
Showdo
Custo
Vouche

Figure 8.5: A 3-by-5 inch (76 mm by 127 mm) Class–Responsibility–Collaborator (CRC) card representing the `Voucher` class from Figure 8.4. 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 8.5) were [proposed in 1989](#) 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 screencast](#) shows, a team designing new code selects a user story (Section 5.1). 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 8.2.1.

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:

For SaaS, the existing tests may need access to a test database, which may not be accessible in production.

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.

For part of your exploration you might want to modify data in the database, which you can't do with live customer data.

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 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 5.7), 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 test seems to require us to understand the implementation. For example, consider the code in Figure 8.6. 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. How would we create unit tests without understanding the method’s logic in detail?

<http://pastebin.com/jj6mCU65>

```
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 8.6: 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. Screencast [8.3.1](#) shows the bug and fix.

<http://pastebin.com/R6r5Y1nF>

```
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     TimeSetter.convert(arg).should == result  
8   end  
9 end  
10 end
```

Figure 8.7: This simple spec, resulting from the reverse-engineering technique shown in Screencast [8.3.1](#), achieves 100% C0 coverage and helps us find a bug in Figure [8.6](#).

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. Screencast [8.3.1](#) shows how we do this for `convert`, resulting in the specs in Figure [8.7](#) and even finding a bug in the process!

Screencast 8.3.1: [Creating characterization specs for TimeSetter](#)

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.

- 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 adding coverage, and don’t get distracted trying to fix bugs along the way.



A key theme of this book is that engineering long-lasting 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 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 8.6. [**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.

BDUF (Big Design Up Front) software projects sometimes include specific contractual requirements based on software metrics.

Figure 8.8 shows the control flow graph corresponding to Figure 8.6, which we can use to compute two widely-used indicators of method-level complexity:

Cyclomatic complexity measures the number of linearly-independent paths through a piece of code.

ABC score is a weighted sum of the number of **A**ssignments, **B**ranches and **C**onditionals in a piece of code.

The cyclomatic complexity metric was invented by software engineer Frank McCabe Sr. in 1976.

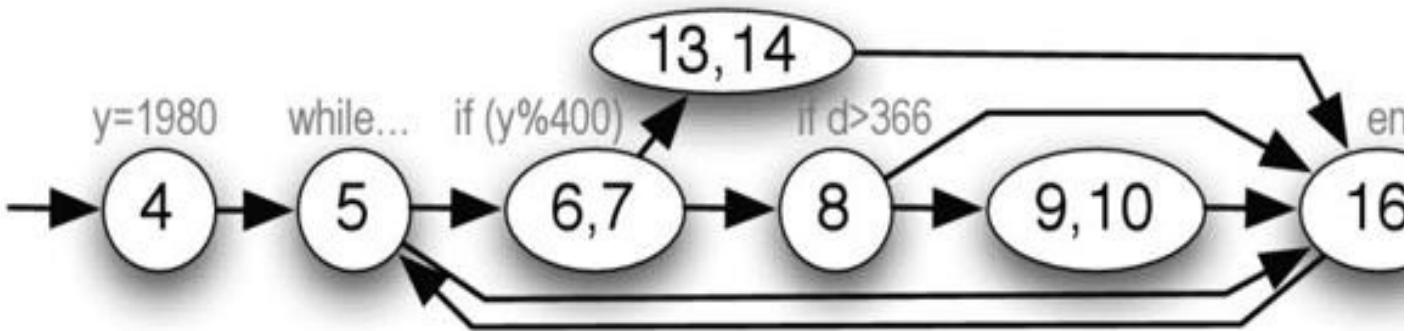


Figure 8.8: The node numbers in this control flow graph correspond to line numbers in Figure 8.6.

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 flog and an ABC score (Assignments, Branches, Conditionals) of 23 as measured by saikuro. Figure 8.9 puts these scores in context.

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. Figure 8.9 summarizes useful metrics we've seen so far that

speak to testability and therefore to code beauty.



Metric	Tool	Target score	Sec.
Code-to-test ratio	<code>rake stats</code>	$\leq 1 : 2$	6.8
C0 coverage	<code>SimpleCov</code>	$\geq 90\%$	6.8
ABC score	<code>flog (rake metrics)</code>	$< 20/\text{method}$	8.4

Figure 8.9: 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 [10](#).

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 [8.10](#), 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.

Design smells (see Chapter [10](#)) tell us when something's wrong in the way classes interact, rather than within the methods of a specific class.

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 yield to

extract the “enclosing” code and having it yield back to the non-repetitive code.

Figure 8.10: 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.

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 **S**hort, so that its main purpose is quickly grasped;
- do only **O**ne thing, so testing can focus on thoroughly exercising that one thing;
- take **F**ew arguments, so that all important combinations of argument values can be tested;
- maintain a consistent level of **A**bstraction, so that it doesn’t jump back and forth between saying *what to do* and saying *how to do it*.



Figure 8.6 violates at least the first and last of these, and exhibits other smells as well, as we can see by running reek on it:

<http://pastebin.com/jAQeaUV6>

```
1   time_setter.rb -- 5 warnings:
2
3   TimeSetter#self.convert calls (y + 1) twice (Duplication)
4
5   TimeSetter#self.convert has approx 6 statements (LongMethod)
6
7   TimeSetter#self.convert has the parameter name 'd' (UncommunicativeName)
8
9   TimeSetter#self.convert has the variable name 'd' (UncommunicativeName)
10
11  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 8.11 provides suggestions.

What	Guideline	Example
Variable or class name	Noun phrase	<code>PopularMovie, top_movies</code>
Method with side effects	Verb phrase	<code>pay_for_order, charge_credit_card!</code>
Method that returns a value	Noun phrase	<code>movie.producers, actor_list</code>
Boolean variable or method	Adjective phrase	<code>alreadyRated?, @is_oscar_winner</code>

Figure 8.11: 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 retying the full name, your colleagues will thank you for writing `@is_oscar_winner` instead of `Oswin`.

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 8.12 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.

```

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
8
9

```

Figure 8.12: 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.

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, following the simple steps in Figure 8.12. In the next section, we will make the Ruby code as transparent as the pseudocode by [refactoring](#) it—applying transformations that improve its structure without changing its behavior.

- 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 **Short**, do **One** thing, have **Few** arguments, and maintain a single level of **Abstraction**.
-

Self-Check 8.4.1.

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 more subtle example is a method such as `before_filter` (Section 7.1), which essentially adds a new method to a list of methods to be called before a controller action.

Self-Check 8.4.2.

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.

With the characterization specs developed in Section 8.3, we have a solid foundation on which to base our refactoring to repair the problems identified in Section 8.4. 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 10.

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 is changed by refactoring. 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, but 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 8.13 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 8.14 shows the first few steps for applying the Extract Method refactoring. With these examples in mind, we can refactor Figure 8.6.

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	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	You have a literal number with a particular meaning.	Create a constant, name it after the meaning, and replace the number with it.

Figure 8.13: 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 8.14 shows.

1. Create a new method, and name it after the intention of the method (name it by what how it does it). If the code you want to extract is very simple, such as a single message call, you should extract it if the name of the new method reveals the intention of the code. 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 as temporary variables.
5. Look to see whether any of these local-scope variables are modified by the extracted code. If the variable is modified, see whether you can treat the extracted code as a query and as a function of the variable concerned. If this is awkward, or if there is more than one such variable, extract the method as it stands. You may need to use *Split Temporary Variable* and try *Eliminate temporary variables with Replace Temp with Query* (see the discussion in the next section).
6. Pass into the target method as parameters local-scope variables that are read from the source method.
7. ...

Figure 8.14: 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.

Long method is the most obvious code smell in Figure 8.6, 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` statement in lines 6–7 is difficult to understand, and the conditional is nested two-deep. As Figure 8.13 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 8.15 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 6.7. 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.

<http://pastebin.com/SnT7DyTi>

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 8.15: Applying the Extract Method refactoring to lines 3–4 of Figure 8.6 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?`. **Note:** Line 7 reflects the bug fix described in Screencast [8.3.1](#).

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 8.12, 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 8.16 shows the result.

<http://pastebin.com/26dKjvKH>

```
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 peel_off_leap_year!
18    if (@@days_remaining >= 366)
19      @@days_remaining -= 366 ; @@year += 1
20    end
21  end
22  def 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 8.16: 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 8.12, sticking to a single level of Abstraction while delegating details to the extracted helper methods.

As long as we're cleaning up, the code in Figure 8.16 also fixes two minor code smells. The first is uncommunicative variable names: `convert` doesn't describe very well what 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 `STARTING_YEAR`. What about the other constants such as 365 and 366? In this example, they're probably familiar enough to most programmers to leave as-is, but if you saw 351 rather than 365, and if line 26 (in `leap_year?`) used the constant 19 rather than 4, you might not recognize the leap year calculation for 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 8.16 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 10. 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 more clumsy 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.

<http://pastebin.com/YkkbyBXi>

```
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 self.leap_year?
31    @year % 400 == 0 ||
32    (@year % 4 == 0 && @year % 100 != 0)
33  end
34 end

```

Figure 8.17: 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.

Figure 8.17 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, description of when to use the refactoring and what it does, and detailed sequence of mechanical steps to apply the refactoring. Effective refactorings should 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*.

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 3.4 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 10, 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 8.5.1. 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.

 **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. *Resist the temptation to fix it!* 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. 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 probably will underestimate the effort vastly, 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. business decision to do so). 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. 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 6 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.

A ship in port is safe, but that's not what ships are built for.

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 which 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 [8.2–8.3](#). 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. 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 cataloged. 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 8.18 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 8.19 lists more. 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.

Category	Refactorings		
Composing	<i>Extract method</i>	Replace temp with method	Introduce explaining variable
Methods	<i>Replace method with method object</i>	Inline temp	Split temp variable
	Remove parameter assignments	Substitute algorithm	
Organizing	self-encapsulate field	replace data value with object	change value to reference
Data	replace array/hash with Object	<i>Replace magic number with symbolic constant</i>	
Simplifying	<i>Decompose Conditional</i>	Consolidate Conditional	Introduce Assertion
Conditionals	Replace Conditional with Polymorphism	Replace Type Code with Polymorphism	Replace Nested Conditional with Guard Clauses
	Consolidate Duplicate Conditional Fragments	Remove Control Flag	Introduce Null Object
Simplifying	Rename Method	Add Parameter	Separate Query from Modifier
Method Calls	Replace Parameter with Explicit Methods	Preserve Whole Object	Replace Error Code with Exception

Figure 8.18: 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 8.19: Several of Fowler's and Martin's code smells, with the ones introduced in this chapter in italics.

M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004. ISBN 9780131177055. URL <http://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052>.

J. Fields, S. Harvie, M. Fowler, and K. Beck. *Refactoring: Ruby Edition*. Addison-Wesley Professional, 2009. ISBN 0321603508. URL <http://www.amazon.com/Refactoring-Ruby-Jay-Fields>.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN 0201485672. URL <http://www.amazon.com/Refactoring-Improving-Design-Existing-Code>.

R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002. ISBN 0321117425. URL <http://www.amazon.com/Facts-Fallacies-Software-Engineering-Robert/dp/0321117425>.

R. Green and H. Ledgard. Coding guidelines: Finding the art in the science. *Communications of the ACM*, 54(12):57–63, Dec 2011.

R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN 9780132350884.

O. Nierstrasz, S. Ducasse, and S. Demeyer. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2009. ISBN 395233412X. URL <http://www.amazon.com/Object-Oriented-Reengineering-Patterns-Oscar-Nierstrasz/dp/395233412X>.

Project 8.1. You're tasked with designing a RESTful API for a hypothetical legacy enrollment system (Berkeley's is called TeleBears). An outside-in description of the system (that is, without examining its database schema) is as follows:

- A course has a department, course number, and description, for example, “Computer Science 169—Software Engineering”.
- An offering of a course specifies additional information:

- the semester (Fall, Spring or Summer) and year that the course is taught,
- the building and room number,
- the day(s) and time(s) of lectures each week,
- the day(s) and time(s) of small-section meetings each week (not all courses have these),
- the instructor,
- the limit on how many students may enroll.

Each offering has a unique ID called the control number.

Provide a UML diagram describing the design of such a system, and a set of RESTful routes (in the form of either a simplified routes .rb file or a table similar to that in Figure [7.18](#)) that support at least the following operations:

- Search for course offerings by any combination of department name, instructor name (partial match OK), semester being offered
- Get meeting times for a course offering (lecture sections, rec sections, etc, each with unique timeslot-ID, day and time)
- Enroll a student in a course offering



Fred Brooks, Jr. (1931–) is the author of the classic software engineering book *The Mythical Man-Month*, based on his years leading the IBM OS/360 operating system effort after managing the System/360 project and reporting directly to IBM Chairman T.J. Watson Jr. The System/360 was the first family of computers that had an instruction-set-compatible architecture across a product family, so many have argued that it is 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*

- 9.1 [It Takes a Team: Two-Pizza and Scrum](#)
- 9.2 [Points, Velocity, and Pivotal Tracker](#)
- 9.3 [Pair Programming](#)
- 9.4 [Design Reviews and Code Reviews](#)
- 9.5 [Version Control for the Two-Pizza Team: Merge Conflicts](#)
- 9.6 [Using Branches Effectively](#)
- 9.7 [Reporting and Fixing Bugs: The Five R's](#)
- 9.8 [Fallacies and Pitfalls](#)
- 9.9 [Concluding Remarks: Teams, Collaboration, and Four Decades of Version Control](#)
- 9.10 [To Learn More](#)
- 9.11 [Suggested Projects](#)

Programming is now primarily a team sport, and this chapter covers techniques that can help teams succeed. Everyone on the team must agree on dividing up work, how to estimate the difficulty of the work to produce a schedule, how to correct the schedule when actual progress differs from predicted progress, and know where and how to check in code. Velocity-based iteration planning, supported by tools such as *Pivotal Tracker*, address the managing and scheduling tasks. Pair programming, design reviews, and code reviews can

improve software quality. Good version control practices, supported by tools such as Git, address code management.

The Six Phases of a Project:

Enthusiasm

Disillusionment

Panic

Search for the Guilty

Punishment of the Innocent

Praise for non-participants

Dutch Holland([Holland 2004](#))

As we've said many times in this book, the Agile lifecycle uses iterations of typically one to two weeks that start and end with a working prototype. Each iteration implements some user stories, which act as acceptance tests or integration tests. The stakeholders examine the product after the iteration to see if this is what everyone wants, and then prioritize the remaining user stories. Figure [9.1](#) highlights where the topics of this chapter fit in the Agile lifecycle.

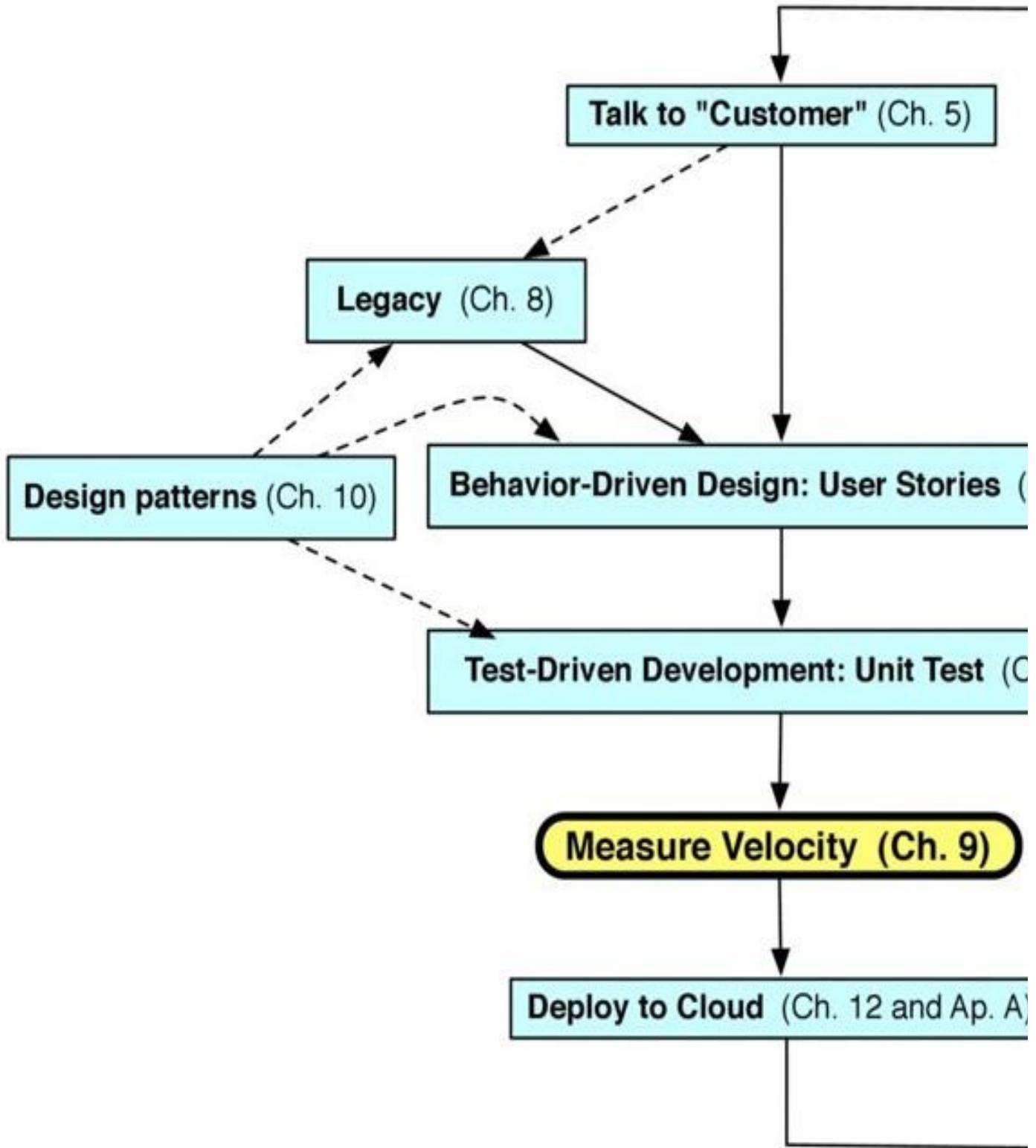


Figure 9.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter emphasizes evaluating the productivity of the team so as to come up with more accurate schedules by tracking velocity.

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 now is 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 make your team succeed. As the opening quote from Fred Brooks states, you cannot win if your team loses, and you cannot fail if your team wins.

Jeff Bezos, the CEO of Amazon who received his college degree in computer science, coined the two-pizza characterization of team size.

Hence, the first step of a software development project is to form and organize a team. As to its size, “two-pizza” teams—the group that can be fed by two pizzas in a meeting—typically develop SaaS projects. Our discussions with senior software engineers suggest the typical team size range varies by company, but the inclusive range of the typical ranges is from four to nine people.

While there are many ways to organize a two-pizza software development, a popular one today is **Scrum** ([Schwaber and Beedle 2001](#)). The name is inspired by its frequent short meetings—15 minutes every day at the same place and time—where each team member answers three questions:

What have you done since yesterday?

What are you planning to do today?

Are there any impediments or stumbling blocks?

The benefit of these daily scrums is that by understanding what each team member was doing, the team can identify work that would help others make more rapid progress.



A scrum is held on every minor infraction in rugby. The game stops to bring the players together for a quick “meeting” in order to restart the game.

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:

Team—A two-pizza size team that delivers the software.

ScrumMaster—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.

Product Owner—A team member (not the ScrumMaster) who represents the voice of the customer and prioritizes user stories.

Scrum relies on self-organization, and team members often rotate through different roles. For example, we recommend that each member rotate through the Product Owner role, changing on every iteration or sprint.

Summary: SaaS is a good match to two-pizza teams and Scrum, a self-organized small team that meets daily. Two team members take on the additional roles of ScrumMaster, who removes impediments and keeps the team focused, and Product Owner, who speaks for the customer.

Self-Check 9.1.1. True or False: Scrum is at its best 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.

Given a size and an organization of a team, we can introduce a metric and tool to measure its productivity.

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 number of stories per week. 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 rate each user story in advance on a simple integer scale. We recommend starting with a three-point scale: 1 for straightforward stories, 2 for medium stories, and 3 for very complex stories. (As you get experience with rating and completing stories, you can use a wider range.) The average is now of the number of **points** per iteration a team completes, which is called **velocity**. The **backlog** is the name of the collection of stories that have not yet been completed in this iteration.

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.

Note that 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.

Pivotal Tracker is a service that tracks user stories and velocity.  Figure 9.2 shows Tracker's UI. You start by entering user stories, which requires filling in the difficulty rating. You then prioritize the stories by placing them either into the Current panel or into the Backlog panel. The order of the stories within each panel defines their relative priority. When stories are completed, they are placed into a Done panel and Tracker suggests stories from the backlog in priority order. Another

category is the 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 intro** Pivotal Labs has produced an [excellent 3-minute video intro](#) to using Tracker.

Tracker also allows the insertion of markers of Release points in the prioritized lists of user stories. As it calculates velocity based on points completed, it also supplies the current estimate of when these software releases will actually occur. This approach is in sharp contrast to management by schedule, where a manager picks a release date and the team is expected to work hard to meet the deadline.

The screenshot shows the Pivotal Tracker interface for the 'Visit Day Meeting Scheduler' project. The top navigation bar includes links for 'PROJECTS', 'DASHBOARD', and 'REPORTS'. Below the header, there are tabs for 'CURRENT', 'BACKLOG', 'ICEBOX', 'DONE', and 'MORE'. The 'STORIES' tab is currently active, displaying a list of 12 iterations (stories) from 41 to 62, each with a due date and completion percentage (e.g., '41 3 Oct Pts: 0 %'). The 'CURRENT' panel on the right shows three stories: 'Replace relevant parts of view layer with spine.js (VC)', 'Standardize "data field" module interface (VC)', and 'Administrator can invite people to sign up as Visitors.' The 'ICEBOX' panel on the far right lists several stories, many of which are marked with yellow stars and icons indicating specific requirements or features like 'algorithm' and 'Faculty can cap'.

Figure 9.2: Screen image of the UI of the Pivotal Tracker service.

Tracker recently added a new concept to combine a collection of related user stories into a group called an ***Epic***. Epics have their own panel and their own progress bar in Tracker, and can be ordered independent of the user stories 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.

Developers do not decide when the user stories are completed. The developer pushes the Deliver button, which sends it to the Product Owner—exactly the same role as in the Scrum organization. The Product Owner tries out the user story and then either hits the Accept button that marks user story as done or hits the Reject button, which marks the story as needing to be Restarted by the developer.

Teams need a virtual commons in which to share information, and Tracker allows you to attach documents to user stories, which seems a perfect place for Lo-Fi sketches and design documents. Here are other good cyberspace locales your team could use to communicate and share information such as meeting notes, software architecture, and so on:

- Every GitHub repository (see Section A.7) offers a Wiki, which allows team members to jointly edit a document and add files.
- [Google Docs](#) allows joint creation and viewing of drawings, presentations, spreadsheets, and text documents.
- [Campfire](#) is a web-based service for password-protected online chat rooms.

Summary: 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.

Pivotal Tracker provides a service that helps prioritize and keep track user stories and their status, calculates velocity, and predicts software development time based on the team's history.

Self-Check 9.2.1. 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 9.2.2. True or False: When you don't know how to approach a given user story, just give it 3 points.

False: User stories should not be so complex that you don't know 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.

Given a team organization and way to keep track and measure progress, we are now ready to start programming.

Two heads are better than one. *English proverb*

Dilbert on Pair Programming The comic strip Dilbert comments humorously on pair programming in these [two strips](#).

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 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. As the name suggests, two software 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 9.3, shows engineers at Pivotal Labs—makers of Pivotal Tracker—who spend most of the day doing pair programming.([Moore 2011](#))



Figure 9.3: 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 courtesy Pivotal Labs.

Pair programming is cooperative, and should involve a lot of talking. It focuses effort to the task at hand, and two people working together increases the likelihood of following good development practices. If one partner is silent or checking email, then it's not pair programming, just two people sitting near each other.

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. ([Hannay et al. 2009](#)), in a subsequent study of pair programming studies, conclude 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 effort than do solo programmers.

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 in 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 9.3.1. 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 tasks—than solo programming.

Self-Check 9.3.2. 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.

Design reviews and **code reviews** originated in the Big Design Up Front methodologies. 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 development.

A design review is a meeting where the authors of program present its design with the goal of improving software quality by benefiting from the experience of the people attending the meeting. A code review is held once the design has been implemented. In the Agile/Scrum context, since design and implementation occur together, they might be held every few iterations or sprints.

While the primary goal is increasing software quality, this peer-oriented feedback also helps with knowledge exchange within the organization and offers coaching that can help the careers of the presenters.

([Shalloway 2002](#)) suggests that formal design code reviews are often too late in the process to make a big impact on the result. 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, he 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.

To have a good review, you first need to have a good meeting! Below is our digest of meeting advice from the many meeting guidelines found on the Web. We use the acronym SAMOSAS as a memory device; surely including them will make for an effective meeting!

- 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.



Samosas are a popular stuffed deep-fried snack from India.

The minutes and action items record the feedback from the design review on how to improve the software.

The formal review itself should start with a high-level description of what the customers want. The Agile lifecycle helps with the rest of the presentation. Present a few storyboards of user stories that capture acceptance tests, and show how they run on the current prototype. Then give the architecture of the software, showing the APIs of the components of your Software-Oriented Architecture. It will be important to highlight the design patterns used at different levels of abstraction. You should expect to explain *why* you made the decisions, and whether you considered plausible alternatives. Depending on the amount of time and interests of those at the meeting, the final phase would be to go through the code of the implemented methods. Towards the end of the meeting, show the information recorded in Pivotal Tracker to show the status of your project: user stories completed, what you intend to work on next, and what stories have not yet been prioritized, and the team's velocity. 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.

Interestingly, most companies using Agile methods do not perform design or code reviews. For example, the conventional wisdom at Pivotal Labs is that pair programming makes such reviews superfluous, as the code is continuously being reviewed during development. At GitHub, formal code reviews are replaced by **pull requests**, in which a developer requests that her latest code changes be integrated into the main codebase, as we describe in the next section. All developers on the team see each such request and determine how it might affect their own code. If anyone has a concern, an online discussion coalesces around the pull request, perhaps facilitated by tools such as Campfire or GitHub Issue tracking (discussed later in this chapter), which might result in changes to the code in question before

the pull request is completed. Since many pull requests occur each day, these “mini-reviews” are occurring continuously, so there is no need for special meetings.

Summary: Design and code reviews let outsiders give feedback on the current design and future plans, letting the team benefit from the experience of others. While not part of the Agile development process, some companies still find them useful.

Self-Check 9.4.1. 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.

For a team to work together one a common code base with many revisions and interdependent pieces, you need tools to help manage the effort.

This section and the next one assume familiarity with basic version control practices in Git. Section [A.6](#) summarizes the basics and Section [9.10](#) suggests in-depth resources.

Good version control practices are even more critical for a team than for individuals. How is the 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 were they changed? How does one developer work on a new feature without introducing problems into stable code? 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.

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 (the *origin*) is designated as authoritative, and all developers agree to push their changes to the origin and periodically pull from the origin to get others’ changes. Famously, Git itself doesn’t care which copy 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, for example on GitHub or ProjectLocker.

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 means something quite different in Git.

Team members **clone** the repo onto their development machines, do their work, make their commits, and push the commits to origin.

<http://pastebin.com/Vk3xm0Jm>

```
1 Roses are red,  
2 Violets are blue.  
3 <<<<< HEAD:poem.txt  
4 I love GitHub,  
5 =====  
6 ProjectLocker rocks,  
7 >>>>> 77976da35a11db4580b80ae27e8d65caf5208086:poem.txt  
8 and so do you.
```

Figure 9.4: 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. (Use Pastebin to copy-and-paste this code.)

We already know that `git push` and `git pull` can be used to back up your copy of the repo to the cloud, but in the context of a team, these operations acquire additional important meanings: if Amy commits her changes to her 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:

Amy and Bob each have a current copy of the origin repo.

Amy makes and commits a set of changes to file A.

Amy makes and commits a separate set of changes to file B.

Amy pushes her commits to origin.

Bob makes and commits his own changes to file A, but doesn't touch file B.

Bob tries to push his commits, but is prevented from doing so because additional commits have occurred in the origin repo since Bob last pulled. Bob must bring his copy of the repo up-to-date with respect to the origin before he can push his changes.

`git pull` actually combines the separate commands `git fetch`, which copies new commits from the origin, and `git merge`, which tries to reconcile them with the local repo.

Note that the presence of additional commits doesn't necessarily mean that conflicting changes were made—in particular, Bob is unaffected by Amy's commit to file B (step 3). In our example, though, steps 2 and 5 do represent conflicting changes to the same file. When Bob performs `git pull`, Git will try to **merge** Bob's and Amy's changes to file A. If Amy and Bob had edited different parts of file A, Git would automatically incorporate both sets of changes into file A and commit the

merged file in Bob's repo. However, if Amy and Bob edited parts of file A that are within a few lines of each other, as in Figure 9.4, Git will conclude that there is no safe way to automatically create a version of the file that reflects both sets of changes, and it will leave a conflicted *and uncommitted* version of the file in Bob's repo, which Bob must manually edit and commit.

In either case, once Bob has committed file A, he can now push his changes, after which the origin repo successfully reflects both Amy's and Bob's changes. The next time Amy pulls, she will get the merge of her changes and Bob's.

This example shows an important rule of thumb: *always commit before merging* (and therefore before pulling, which implicitly performs a merge). Committing ensures that you have a stable snapshot of your own work before attempting to merge changes made by others. Git warns you if you try to merge or pull with uncommitted files, and even provides mechanisms to recover from this scenario if you choose to proceed anyway (see Figure 9.5); but your life will be easier if you *commit early and often*, making it easier to undo or recover from mistakes.

Git's advanced features like *rebasing* and *commit squashing* can merge many small commits into a few larger ones to keep the publicly-visible change history clean.

Since working in a team means that multiple developers are changing the contents of files, Figure 9.6 lists some useful Git commands to help keep track of who did what and when. Figure 9.7 shows some convenient notational alternatives to the cumbersome 40-digit Git commit-IDs.

```
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 9.7 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 9.5: When a merge goes awry, these commands can help you recover by undoing all or part of the merge.

```
git blame [file]
```

Annotate each line of a file to show who changed it last and when.

```
git diff [file]
```

Show differences between current working version of *file* and last committed version.

```
git diff branch [file]
```

Show differences between current version of *file* and the way it appears in the most recent commit on *branch* (see Section 9.6).

```
git log [ref..ref] [files]
```

Show log entries affecting all *files* between the two commits specified by the *refs* (which must be separated by exactly two dots), or if omitted, entire log history affecting those files.

```
git log --since="date" files
```

Show the log entries affecting all *files* since the given date (examples: "25-Dec-2011", "2 weeks ago").

Figure 9.6: 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`.

HEAD	the most recently committed version on the current branch.
HEAD [~]	the prior commit on the current branch (HEAD ^{~n} refers to the <i>n</i> 'th previous commit).
ORIG_HEAD	When a merge is performed, HEAD is updated to the newly-merged version, and ORIG_HEAD 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.
1dfb2c ^{~2}	2 commits prior to the commit whose ID has 1dfb2c as a unique prefix.
"branch@{date}"	The last commit prior to <i>date</i> (see Figure 9.6 for date format) on <i>branch</i> , where HEAD refers to the current branch.

Figure 9.7: 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:

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 referred to as the *origin* repo and is often stored in the cloud on GitHub or on an internal company server.

Before you start changing files, *commit* your own changes locally and then *merge* the changes made by others while you were away. In Git, the easiest way to merge changes from the origin repo is `git pull`.

If changes cannot be automatically merged, you must manually edit the conflicted file by looking for the *conflict markers* in the merged file, and then commit and push the fixed version. With Git, a conflict is considered resolved when the conflicted file is re-committed.

ELABORATION: Remote collaboration: fork-and-pull for public repos

Git was designed to support very-large-scale projects with many developers, such as the Linux kernel. The **fork-and-pull** management model allows subgroups to work on independent and possibly divergent sets of changes without interfering with each others’ efforts. A remote subgroup can [fork your repo](#), which creates their own copy of it on GitHub to receive their pushes. When they are ready to contribute stable code back to your repo, the subgroup [creates a pull request](#) asking you to merge selected commits from their forked repo back into your origin repo. Pull requests therefore allow selective merging of two repos that otherwise remain separate.

Self-Check 9.5.1. 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.

 False. 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.

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 *graph* of commits: a branch is started by creating a logical copy of the code tree as it exists at some particular commit. Unlike a real tree branch, a repo branch not only diverges from the “trunk” but can also be merged back into it.

From that point on, the new 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.

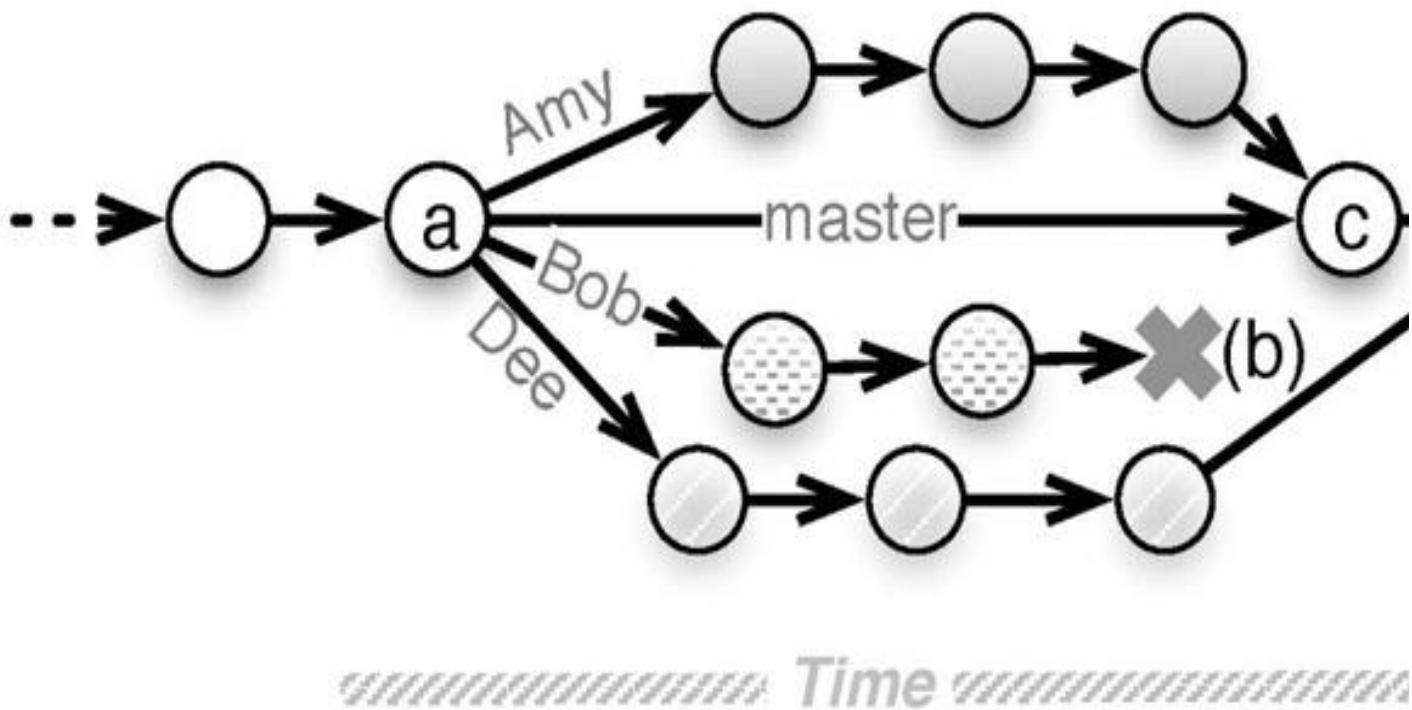


Figure 9.8: Each circle represents a commit. Amy, Bob and Dee each start branches based on the same commit (a) to work on different RottenPotatoes features. After several commits, Bob decides his feature won't work out, so he deletes his branch (b); meanwhile Amy completes her tests and code and merges her feature branch back into the master branch, creating the merge-commit (c). Finally, Dee completes her feature, but since the master branch has changed due to Amy's merge-commit (c), Dee has to do some manual conflict resolution to complete her merge-commit (d).

We highlight two common branch management strategies that can be used together or separately, both of which strive to ensure that the master branch always contains a stable working version of the code. Figure 9.8 shows a **feature branch**, which allows a developer or sub-team to make the changes necessary to implement a particular feature without affecting the master branch until the changes are complete and tested. If the feature is merged into the master 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 master that depend on the new feature.

| **Flickr** developers now [use](#) a repository with no feature branches at all—commits always go to the master branch!

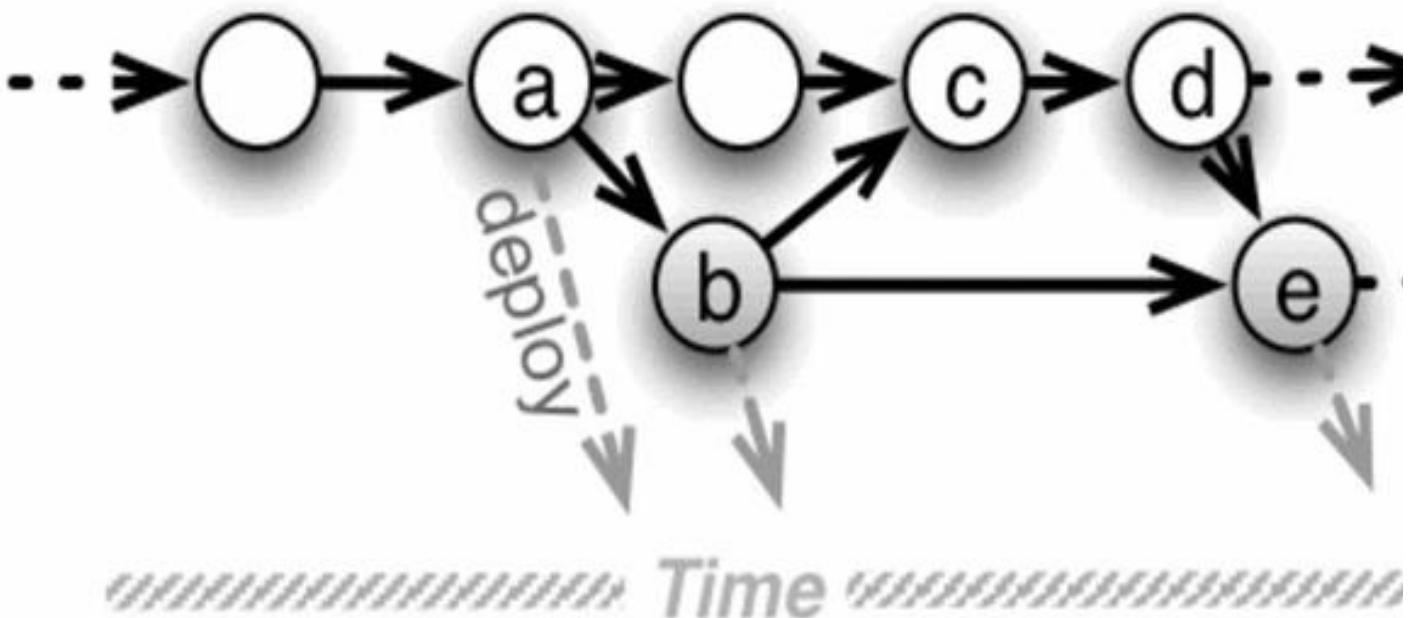


Figure 9.9: (a) A new release branch is created to “snapshot” version 1.3 of RottenPotatoes. A bug is found in the release and the fix is committed in the release branch (b); the app is redeployed from the release branch. The commit(s) containing the fix are merged into the master branch (c), but the code in the master 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 master 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 master branch changes to the release branch except for this fix.

Figure 9.9 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 master 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 master 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 master branch, so you might as well deploy directly from the master branch. We discuss continuous deployment further in Chapter 12.

| [GitFlow](#), a branch management strategy that captures many best practices, may be useful for larger projects with long-lived branches.

Figure 9.10 shows some commands for manipulating Git branches. Newly-created Git repos start out with a single branch called `master`. 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 Screencast 9.6.1 for an

important caveat about doing so).

`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, putting [this code](#) in the file `~/.profile` will make the shell prompt display the current Git branch when you're in a Git repo directory.

`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 `master`.

`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 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 9.10: Common Git commands for handling branches and merging. Branch management involves merging; Figure 9.5 tells how to undo merges gone awry.

When multiple branches are present, how do you specify which one should receive pushes or pulls? As Figure 9.10 shows, the `git push` and `git pull` commands we've been using so far are actually abbreviated special cases—these commands handle pushes and pulls using branches as well.

Screencast 9.6.1: [Using Branches with Git](#)

This screencast shows how to create and manage a new branch in Git (for example, to develop a new feature), how to merge the branch's changes back into the trunk from which it was split, and how to undo the merge of the branch if something goes wrong (for example, if it turns out the feature had bugs and needs to be backed out). It also emphasizes an important caveat and shows why you should always commit your changes in the current branch before switching to a different branch. Lastly, we use `gitk` to view the branching and merging history on our local repo.

Summary of branching:

- 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 master branch) may result in conflict merges for certain files, so always commit before you merge and before switching to a different branch to work on.

- With Agile + SaaS, feature branches are usually short-lived and release branches uncommon.
-

ELABORATION: Long-lived Branches and Rebasing

While you're working on a feature branch, its commits will diverge from the trunk; if you work on it for too long, the "big merge" when you're done may be very painful with many conflicts. The pain can be mitigated by frequent *rebasing*, an operation in which you incrementally merge some recent changes from another branch, then tell Git to rearrange things to look as if your branch had originated from a later commit. While rebasing is useful for long-lived branches such as release branches or long-lived experimental branches, if you're breaking down your user stories into manageable sizes (Section 9.2) and doing frequent deployments (Section 12.3), rebasing should rarely be necessary in agile SaaS development.

Self-Check 9.6.1. Describe a scenario in which merges could go in both directions—changes in a feature branch merged back into the master branch, and changes in the master branch merged into a feature branch. (In Git this is called a *cross-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 master branch. Because the bug is in a part of the code that interacts with Diana's feature, she merges the fix from master into her own feature branch. Finally, when she finishes the feature, her feature branch is merged back into master.

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:

Reporting a bug

Reproducing the problem, or else Reclassifying it as "not a bug" or "won't be fixed"

creating a Regression test that demonstrates the bug

Repairing the bug

Releasing 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 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.

"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! ([Bodík et al. 2006](#))

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 hooks](#), 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.

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 9.7.1. Why do you think “bug fix” stories are worth zero points in Tracker even though they follow the same lifecycle as regular user stories?

Fixing bugs doesn’t add customer value, so the team’s velocity should not be artificially inflated because they are fixing bugs.

Self-Check 9.7.2. 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 [FakeWeb gem](#) or the techniques described in Section [6.7](#), can almost always be used to mimic the external conditions necessary to reproduce the bug in an automated test.

Self-Check 9.7.3. 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 ??) can be used to develop such tests.

Pitfall: Always watching the master 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: 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.

Pitfall: Accidentally stomping on changes after merging or switching branches.

If you do a pull or 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`, as described in “Gitster” Scott Chacon's informative and detailed [blog post](#).

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 9.6) will be painful. You should always `git pull` before starting to work, and `git push` as soon as your locally-committed changes are stable enough to inflict on your teammates. If you're working on a long-lived feature branch that is at risk of getting out of sync with the master, see the documentation for `git rebase` to periodically “re-sync” your branch without merging it back into master until it's ready.

Fallacy: It's fine to make simple changes on the master 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 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.

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)

allowed only one developer at a time to “lock” a particular file for editing—others could only read but not edit the file until the first developer checked the file back in, releasing the lock. SCCS and RCS also required all developers to use the same (then timeshared) computer, whose filesystem held the repo. In a project with many files, this locking mechanism quickly became a bottleneck, so in 1986 the [Concurrent Versions System](#) (CVS) finally allowed simultaneous editing of the same file with automatic merging, and allowed the master repo to be on a different computer than the developer’s copy, facilitating distributed development. [Subversion](#), introduced in 2001, had much better support for branches, allowing developers to independently work on different versions of a project, but still assumed all developers working on a particular code tree would push their changes to a single “master” copy of the 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. This new two-pizza team size makes it easier for a team to stay organized than the giant programming teams of Big Design Up Front.

Despite these improvements, software projects are still infamous for being late and over budget. The techniques in this chapter can help an agile team avoid those pitfalls. Rating user stories on difficulty and recording the points actually completed per iteration increases the chances of more realistic estimates. Checking in with all stakeholders on 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. The Scrum team organization fits well with Agile lifecycle and the challenges of developing SaaS. Disciplined use of 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, once the project is complete, it is important to take the time to think about on what you learned on this project before leaping head first into your next one. 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.

9.10 To Learn More

- 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 Book](#), 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.

- 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. Large projects with wide software distribution use considerably more sophisticated (and complex) bug tracking systems such as the open-source [Bugzilla](#).

P. Bodík, A. Fox, M. I. Jordan, D. Patterson, A. Banerjee, R. Jagannathan, T. Su, S. Tenginakai, 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.

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

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.

D. Holland. *Red Zone Management*. WinHope Press, 2004. ISBN 0967140188. URL <http://www.amazon.com/Red-Zone-Management-Dutch-Holland/dp/0967140188>.

J. Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, 2009. ISBN 0596520123. URL <http://www.amazon.com/Version-Control-Git-Collaborative-Development/dp/0596520123>.

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

K. Schwaber and M. Beedle. *Agile Software Development with Scrum (Series in Agile Software Development)*. Prentice Hall, 2001. ISBN 0130676349. URL <http://www.amazon.com/Agile-Software-Development-Scrum/dp/0130676349>.

A. Shalloway. *Agile Design and Code Reviews*. 2002. URL <http://www.netobjectives.com/download/designreviews.pdf>.

Project 9.1. Select several exercises from the book, assign points to them and then measure your velocity as you work through them.

Project 9.2. Think about a website that you frequently visit, or a web app that you often use; list some user stories that would guide you to create a similar application from the ground up.

Project 9.3. (Discussion) Think about the last project you worked on in a group. How many of the ideas and practices discussed in the chapter did you and your group use? Of those that were used, which did you find the most useful? Which unused methods do you think would have been the most helpful?

Project 9.4. (Discussion) A suggested project in Chapter 1 was to make a list of the Top 10 most important applications. Given such a list, which would best be developed and maintained using a two-pizza sized team with a Scrum organization? Which ones would not? List your reasons for each choice.

Project 9.5. (Discussion) Why do you think did Pivotal added Epics to Tracker? What

was the problem that they were solving with this new feature?

Project 9.6. (Discussion) Find a nearby programming colleague to try pair programming for a few days. Several of the suggested projects in the early chapters are good candidates for pair programming. How hard was it to find a place where you could sit side-by-side? Do you find it forces you both to concentrate more to create higher quality code, in part setting distractions aside, or does it seem less productive since essentially only one person is doing any work?

Project 9.7. With a programming partner, come up with a simple website or application that you could build. Ideally, it should be no more complicated than RottenPotatoes! Using the methods and tools described in this chapter—pair programming, velocity, version control—work on and complete your project. Which of the tools or methods were the most helpful?

Project 9.8. (Discussion) Next time you go to a meeting, keep track how many of SAMOSA guidelines are being violated. If there are several, suggest as an experiment that you try following SAMOSA. What did you notice about the differences between the two meetings? Did SAMOSA help or hurt the meeting?

Project 9.9. (Discussion) Subversion (svn) was a popular version control system developed by CollabNet five years before Linus Torvalds created git. What were the problems with svn that Torvalds was trying to solve with git? How well did he succeed? What do you think are the pros and cons of svn versus git?



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*

10.1 [Patterns, Antipatterns, and SOLID Class Architecture](#)

10.2 [Just Enough UML](#)

10.3 [Single Responsibility Principle](#)

10.4 [Open/Closed Principle](#)

10.5 [Liskov Substitution Principle](#)

10.6 [Dependency Injection Principle](#)

10.7 [Demeter Principle](#)

10.8 [Fallacies and Pitfalls](#)

10.9 [Concluding Remarks: Is Agile Design an Oxymoron?](#)

10.10 [To Learn More](#)

10.11 [Suggested Projects](#)

Besides reusability, programmer productivity requires concise, readable code with minimal clutter. In this chapter, we describe some concrete guidelines for making your class architecture DRY and maintainable: the SOLID principles of object-oriented design—Single Responsibility, Open/Closed, Liskov Substitution, Injection of Dependencies, and Demeter—and some design patterns supporting them. We will learn about design smells and metrics that may warn you of violations of SOLID, and explore some refactorings to fix those problems. In some cases, those refactorings will lead us to one of a collection a design patterns—proven “templates” for class interaction that capture successful structural solutions to common software problems.

10.1 Patterns, Antipatterns, and SOLID Class Architecture

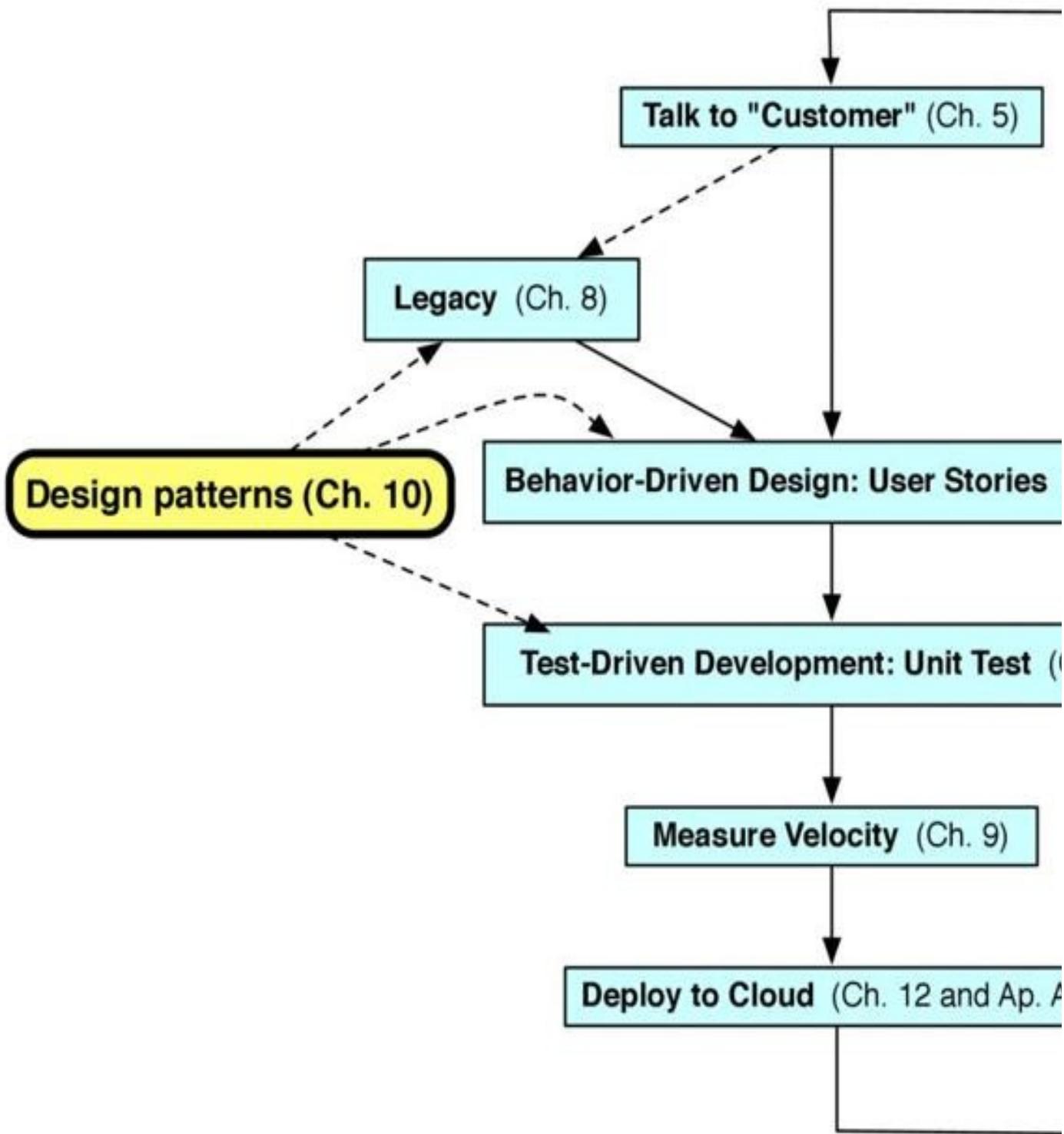


Figure 10.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter covers design patterns, which influence BDD and TDD for new apps *and* for enhancing legacy code.

In Chapter 2, 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 10.1 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 10.2 shows, we will follow a similar approach as we did in Chapter 8. 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.

Chapter 8

Code smells warn of problems in methods of a class

Many catalogs of code smells and refactorings; we use Fowler's as definitive

ABC, Cyclomatic Complexity metrics complement code smells with quantitative warnings

Refactoring by extracting methods and moving code within a class

SOFA guidelines for good methods (**S**hort, do **O**ne thing, **F**ew arguments, single **A**bstraction level)

Some code smells don't apply in Ruby

Chapter 10

Design smells warn of problems in relationships among classes

Many catalogs of design smells and design patterns; we use Ruby-specific versions of the Gang of Four (GoF) design patterns as definitive

LCOM (Lack of Cohesion of Methods) metric complements design smells with quantitative warnings

Refactoring by extracting classes and moving code between classes

SOLID guidelines for good class architecture (**S**ingle responsibility, **O**pen/Closed, **L**iskov substitution, dependency **I**njection, **D**emeter)

Some design smells don't apply in Ruby or SaaS

Figure 10.2: The parallels between the warning symptoms and remedies introduced for individual classes and methods in Chapter 8 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.

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**](#).” The 23 GoF design patterns are divided into Creational, Structural, and Behavioral design patterns, as Figure [10.3](#) 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](#)).

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.

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.

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’ll see in Chapter [11](#), 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 10.3: 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.

In an ideal world, all programmers would use design patterns tastefully, continuously refactoring their code as Chapter 8 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 8, 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 8, the mnemonic SOLID (credited to Robert C. Martin) stands for a set of five design principles that clean code should respect. As in Chapter 8, 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.

"**Uncle Bob**" Martin, an American software engineer and [consultant](#) 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.

Figure 10.4 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:

Viscosity: it's easier to fix a problem using a quick hack, even though you know that's not the right thing to do.

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.

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**.

Needless complexity: the app's design reflects generality that was inserted before it was needed.

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 which 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 (Section 6.10)	Delegate behaviors and call the delegate methods instead

Figure 10.4: 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.

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 [10.10](#) 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.

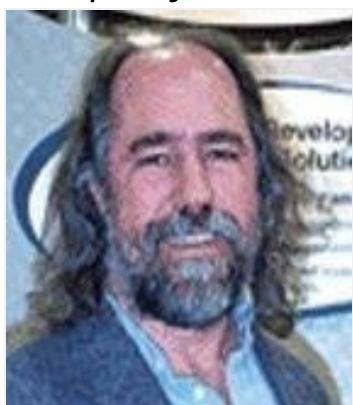
ELABORATION: Other types of patterns

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 applications](#) (we met some in Chapter 2), [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 patterns](#).

Self-Check 10.1.1.

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.



Grady Booch (1955–), internationally recognized for his work in software engineering and collaborative development environments, developed

UML with Ivar Jacobson and James Rumbaugh.

The [**Unified Modeling Language**](#) or UML is not a textual language, but a set of graphical notation techniques to “[specify, visualize, modify, construct, and document the artifacts of an object-oriented software-intensive system under development](#).” UML evolved from 1995 to the present through the unification of previously-distinct modeling language standards and diagram types, which Figure 10.5 lists.

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 diagrams	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 10.5: 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 UML](#), 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.

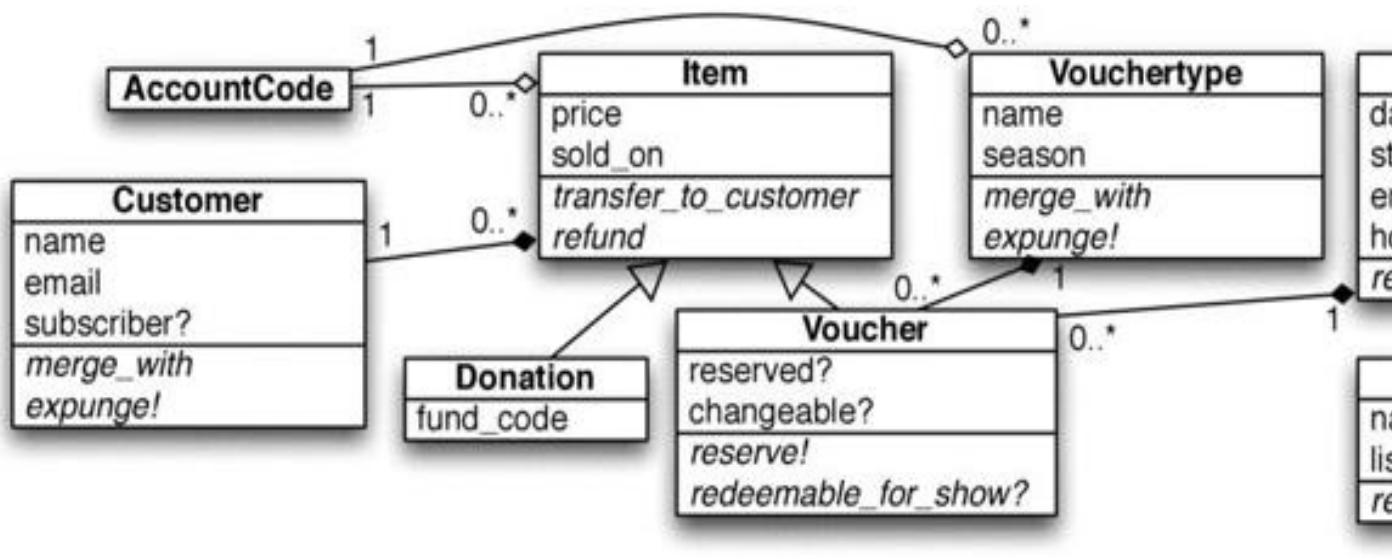


Figure 10.6: This UML [class diagram](#) shows a subset of the classes in the theater-ticketing app consistent with Figures 8.4 and 8.5. Each box represents a class with its most important methods and attributes (responsibilities). Inheritance is represented by an arrow. Classes with associations are connected by lines whose endpoints are annotated with a multiplicity and optionally a diamond—open for aggregations, filled for compositions, absent otherwise.

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 10.6 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 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”, $0\dots*$ 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 [UMPLE](#), a domain-specific language developed at the University of Ottawa for expressing class relationships. The [Try Umple](#) web site can generate UML class diagrams from UMPL code, generate UMPL code from diagrams you draw yourself, or generate executable code in various programming languages corresponding to your UMPL 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 10.2.1. 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 10.2.2. 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.

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 [7.2](#), 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 [7](#), 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 he or she signs 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 [10.7](#) shows two of the most commonly used variants of the LCOM metric.

In Section [8.5](#), 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.

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 in your class as number of connected components in a graph in related methods' nodes are connected by an edge. A score $n > 1$ suggests that up to $n - 1$ responsibilities could be extracted into their own classes.

Figure 10.7: The “recommended” LCOM score depends heavily on which LCOM variant is used. The table shows two of the most widely-used variants.

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 into a new `Address` class, as Figure 10.8 shows.

<http://pastebin.com/RwPzDREv>

```

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 verify_zipcode ; ... ; end
7   def format_address(street, phone_number, zipcode) # data c
lump
8     # do formatting, calling format_phone_number and verify_
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 o
n 'self'
22   private # no need to expose these now:
23   def format_phone_number ; ... ; end
24   def verify_zipcode ; ... ; end
25 end

```

Figure 10.8: 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.

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 clients. 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 10.3.1. Draw the UML class diagrams showing class architecture before and after the refactoring in Figure [10.8](#).

Figure [10.9](#) shows the UML diagrams.

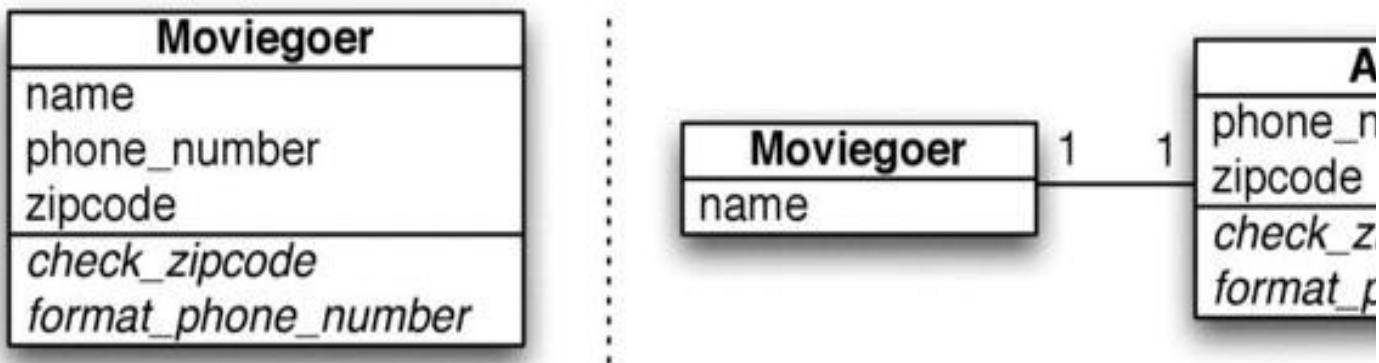


Figure 10.9: UML class diagrams before (left) and after (right) extracting the `Address` class from `Moviegoer`.

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 [10.10](#) shows why the presence of `case`-based dispatching logic—one variant of the *Case Statement* design smell—suggests a possible OCP violation.

<http://pastebin.com/r1KPQA0t>

```
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     # ...
10    end
11  end
12 end
```

Figure 10.10: 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](#).

<http://pastebin.com/3MxUNNSD>

```
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 10.11: 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.

Depending on the specific case, various design patterns can help. One problem that the smelly code in Figure 10.10 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 10.11 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; hence it is usually implemented using composition. Figure 10.12 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 7.2): 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."

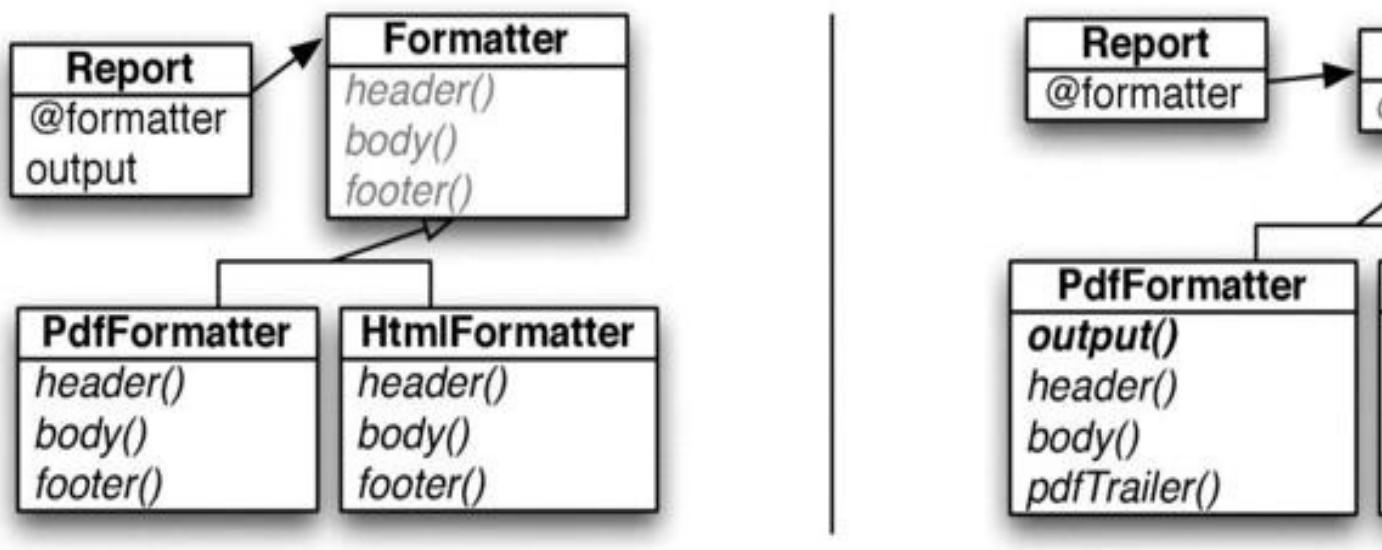


Figure 10.12: In Template Method (left), the extension points are `header`, `body`, and `footer`, since the `Report#output` method calls `@formatter.header`, `@formatter.body`, and so on, each of which delegates to a specialized counterpart in the appropriate subclass. (Light gray type indicates methods 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**](#).

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 10.13 (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 10.14 shows the code corresponding to the more elegant decorator-based design of the PDF formatter shown in Figure 10.13 (right).

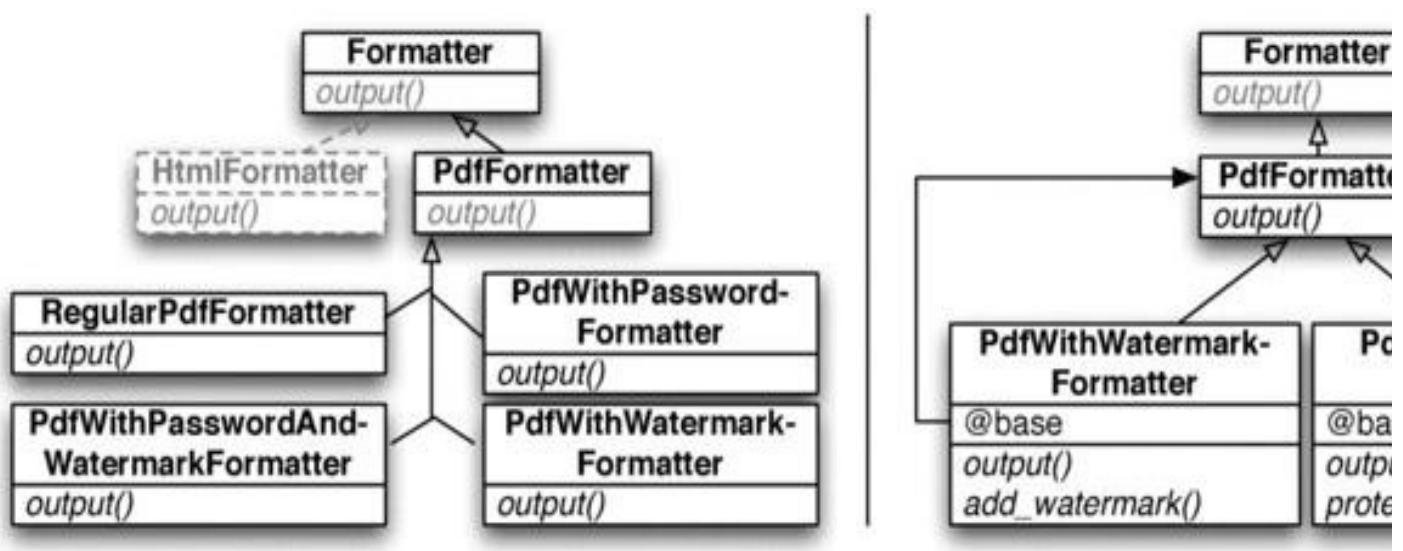


Figure 10.13: (Left) The multiplication of subclasses resulting from trying to solve the Formatter problem using inheritance shows why your class designs should “prefer composition over inheritance.” (Right) A more elegant solution uses the Decorator design pattern.

<http://pastebin.com/siYYms6T>

```

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 ; ... ; end
8   def output
9     @base.output.protect_with_password
10  end

```

```

11 end
12 class PdfWithWatermarkFormatter < PdfFormatter
13   def initialize(base) ; @base = base ; end
14   def add_watermark ; ... ; end
15   def output
16     @base.output.protect_with_password
17   end
18 end
19 # If we just want a plain PDF
20 formatter = PdfFormatter.new
21 # If we want a "draft" watermark
22 formatter = PdfWithWatermarkFormatter.new(PdfFormatter.new)
23 # Both password protection and watermark
24 formatter = PdfWithWatermarkFormatter.new(
25   PdfWithPasswordFormatter.new(PdfFormatter.new))

```

Figure 10.14: 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 [10.5](#)). 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.

<http://pastebin.com/xghNdUW0>

```

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 m
ethod
5     send_email_without_cc(recipient, body) # will call origin
al 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 10.15: 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.

| **Python's "decorators"** are, unfortunately, completely unrelated to the Decorator design pattern.

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 10.15 shows. A more interesting example of Decorator in the wild is the Rack application server we've been using since Chapter 2. 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.

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 for 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.

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."



Barbara Liskov was one of the first women in the USA to receive a Ph.D. in computer science, in 1968.

<http://pastebin.com/aGLbJzrA>

```
1 class Rectangle
2   attr_accessor :width, :height, :top_left_corner
3   def new(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 10.16: 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`.

This may seem like common sense, but it's subtly easy to get wrong. Consider the code in Figure 10.16, which suffers from an LSP violation. You might think a `Square` is just a 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.

<http://pastebin.com/2gZCgrYz>

```
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 10.17: As with some OCP violations, the problem arises from a misuse of inheritance. As Figure 10.17 shows, preferring composition and delegation over 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.

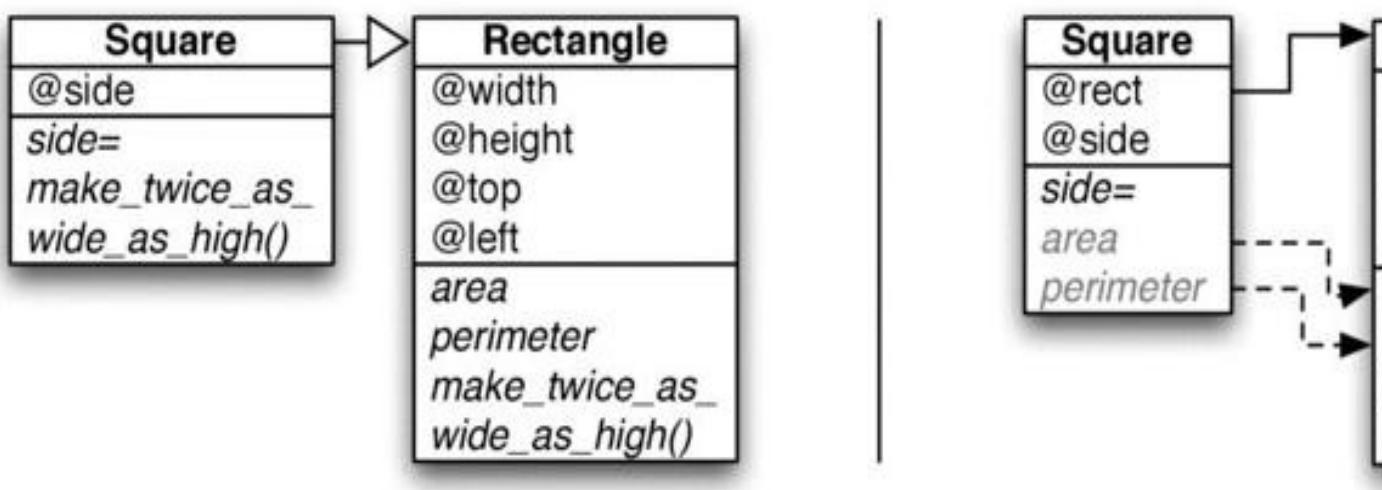


Figure 10.18: Left: The UML class diagram representing the original LSP-violating code in Figure 10.16, which destructively overrides `Rectangle#make_twice_as_wide_as_high`. Right: the class diagram for the refactored LSP-compliant code in Figure 10.17.

The fix, therefore, is to again use composition and delegation rather than inheritance, as Figure 10.17 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`.

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 10.5.1. 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 over inheritance!

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 which 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:

<http://pastebin.com/5zbPHjMq>

```
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
13
14
```

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:

<http://pastebin.com/0JBj79Dh>

```
1  class EmailList
2    attr_reader :mailer
3    delegate :send_email, :to => :mailer
4    def initialize(mail_type)
5      @mailer = mail_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
12   mailer = if Config.has_amiko? then Amiko else MailerMonkey e
nd
12   EmailList.new(mailer).send_email_to moviegoers
13 end
14
15
16
17
```

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.

`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 6 shows that Ruby’s open classes and metaprogramming let you insert test seams wherever needed.

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:

<http://pastebin.com/EAvjBwBL>

```
1  class AmikoAdapter
2    def initialize ; @amiko = Amiko.new(...) ; end
3    def send_email
4      @amiko.authenticate(...)
5      @amiko.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
12    mailer = if Config.has_amiko? then AmikoAdapter else MailerMonkey end
13    EmailList.new(mailer).send_email_to moviegoers
14  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 [Facade pattern](#).

<http://pastebin.com/wmTcSJFy>

```
1 class Config
2   def self.email_enabled? ; ... ; end
```

```
3 def self.emailer ; if has_amiko? then Amiko else MailerMonkey
key 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
11
```

<http://pastebin.com/Ge131bKB>

```
1 class Config
2   def self.emailer
3     if email_disabled? then NullMailer else
4       if has_amiko? then AmikoAdapter else MailerMonkey end
5     end
6   end
7 end
8 class NullMailer
9   def initialize ; end
10  def send_email ; 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
17
```

Figure 10.19: 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.

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 10.19 (top) shows a naive approach: we have moved the logic for determining which emailer 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 10.19 (bottom) applies the Null Object pattern to this example, avoiding the proliferation of conditionals throughout the code. Figure 10.20 shows the UML class diagrams corresponding to the various versions of our DIP example.

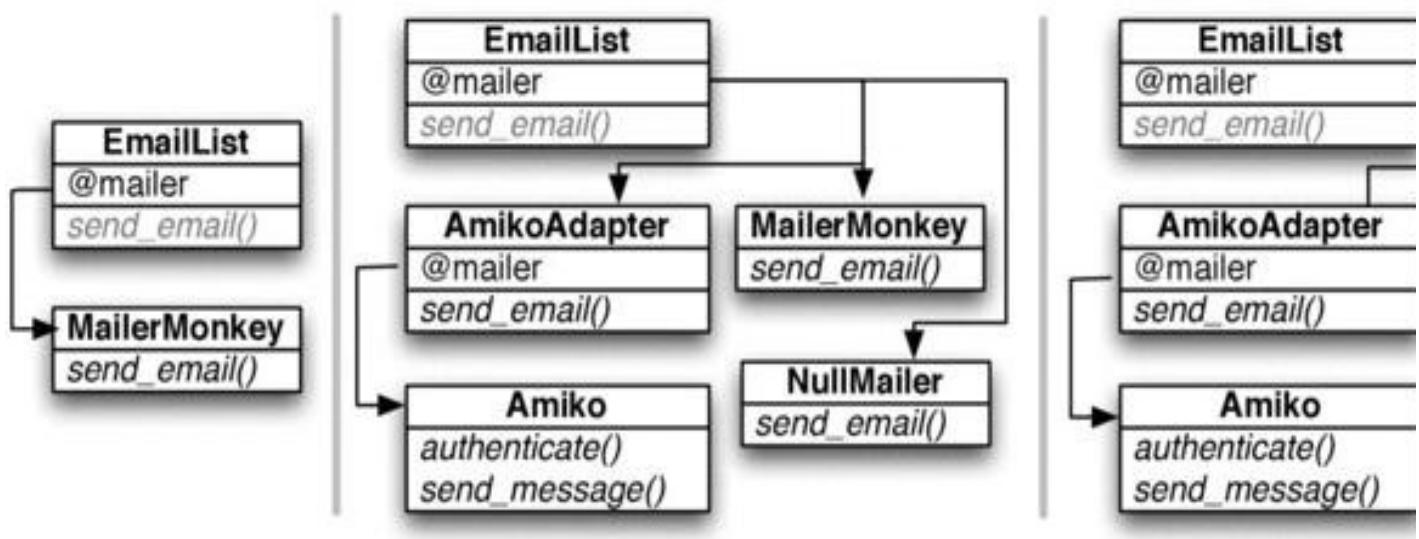


Figure 10.20: Left: Without dependency injection, **EmailList** depends directly on **MailerMonkey**. Center: With dependency injection, `@mailer` can be set at runtime to use any of **MailerMonkey**, **NullMailer** (which implements the Null Object pattern to disable email), or **AmikoAdapter** (which 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.

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. A classic example of this pattern is ActiveRecord associations (Section 7.3). Recall that under the relationship *Movie has many Reviews*, we could write `r=@movie.reviews`. What kind of object is `r`? Although we’ve seen that we can treat `r` as an enumerable collection, it’s actually a proxy object that responds to all the collection methods (`length`, `<<`, and so on), but 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, thereby shielding the client (email GUI) from having to change its behavior when the user isn't connected.

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 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 [10.4](#). 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 10.6.1. 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 injecting the dependency.

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 10.4 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).

<http://pastebin.com/gX2Xs8Ws>

```
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, amount)
11    # VIOLATION OF DEMETER (see text)
12    if moviegoer.wallet.credit_balance < 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 violatio
```

```

n
25     wallet = mock('wallet', :credit_balance => 5.00)
26     moviegoer = mock('moviegoer', :wallet => wallet)
27     lambda { @theater.collect_money(moviegoer, 10.00) }.
28         should raise_error(...)
29     end
30   end
31 end

```

Figure 10.21: 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`.

Figure 10.21 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 ***inappropriate 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 10.21: 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 10.22 (top) shows. But the best delegation is that in Figure 10.22 (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.

<http://pastebin.com/5u2BLrC9>

1 # Better: delegate credit_balance so MovieTheater only acces

```
ses 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,amount)
9     if moviegoer.credit_balance >= amount
10      moviegoer.credit_balance -= due_amount
11      @collected_amount += due_amount
12    else
13      raise InsufficientFundsError
14    end
15  end
16 end
```

<http://pastebin.com/CanQdVHv>

```
1 class Wallet
2   attr_reader :credit_balance # no longer attr_accessor!
3   def withdraw(amount)
4     raise InsufficientFundsError if amount > @credit_balanc
e
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 10.22: (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.

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 violations. 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 [Nokogiri](#) 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.

Observer was first implemented in the MVC framework of [Smalltalk](#), from which Ruby inherits its object model.

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.

```

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 10.23: 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.

While the Ruby standard library [includes a mixin 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 [7.1](#). Figure [10.23](#) shows how easy it is to add an `EmailList` class to RottenPotatoes that “subscribes” to two kinds of state changes:

When a new review is added, it emails all moviegoers who have already reviewed that same movie.

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 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:

<http://pastebin.com/5Ht0Hi8G>

```
1 %p Review of: #{@review.movie.title}  
2 %p Written by: #{@review.moviegoer.name}
```

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 10.23 showed: from the programmer's point of view, all of the logic is in the observing class, not the subjects.

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: Using initialize to implement factory patterns. In Section 10.4, 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:

<http://pastebin.com/zd2MKWNn>

```
1   class A
2
3     def self.create(subclass, *args) # subclass must be either 'A1' or 'A2'
4       return Object.const_get(subclass).send(:new, *args)
5     end
6   end
```

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

As Martin Fowler points out in his article [Is Design Dead?](#), 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 Big Design Up Front (BDUF) is 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, especially if 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 [2](#) 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, because if design decisions early in the project cripple scalability, 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. But if you don't have previous experience with that design constraint or element, 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. In this sense, 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.

10.10 To Learn More

- *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 talks about patterns made unnecessary



by Ruby language features.

- *Rails Antipatterns* ([Pytel and Saleh 2010](#)) gives great examples of how real-life code that starts from a good design can become cluttered, and how to beautify and streamline it by applying refactorings (many drawn from [Fields et al. 2009](#)) to use one or more of the appropriate design patterns.
- *Clean Code* ([Martin 2008](#)) has a more thorough exposition of the SOFA and SOLID guidelines.
- Martin Fowler's [catalog of refactorings](#) is based on his original book *Refactoring* ([Fowler et al. 1999](#)); a recent variant of the book focuses specifically on refactoring in Ruby ([Fields et al. 2009](#)).
- M.V. Mäntyllä and C. Lassenius have created a taxonomy of code and design smells based on various existing sources and grouped into descriptively-named categories such as "The Bloaters", "The Change Preventers", and so on. Their [web page](#) shows the overall taxonomy and links to the full [journal article](#).

J. Fields, S. Harvie, M. Fowler, and K. Beck. *Refactoring: Ruby Edition*. Addison-Wesley Professional, 2009. ISBN 0321603508. URL <http://www.amazon.com/Refactoring-Ruby-Jay-Fields>.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN 0201485672. URL <http://www.amazon.com/Refactoring-Improving- Design-Existing-Code>.

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. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN 9780132350884.

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.

(Suggested Projects will be added in a future revision of the Beta Edition.)



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 eponymous [**Backus-Naur form**](#) (named for Turing award winner Jim Backus and his colleague Peter Naur).

A language that doesn't affect the way you think about programming is not worth knowing. *Alan Perlis*

- 11.1 [JavaScript: The Big Picture](#)
- 11.2 [Client-Side JavaScript for Ruby Programmers](#)
- 11.3 [Functions and Prototype Inheritance](#)
- 11.4 [The Document Object Model and jQuery](#)
- 11.5 [Events and Callbacks](#)
- 11.6 [AJAX: Asynchronous JavaScript And XML](#)
- 11.7 [Fallacies and Pitfalls](#)
- 11.8 [Concluding Remarks: JavaScript Past, Present and Future](#)
- 11.9 [To Learn More](#)
- 11.10 [Suggested Projects](#)

Proper use of JavaScript enhances the user experience for newer browsers without excluding older browsers or those in which JavaScript is disabled. The Web's client-side programming language has a bad reputation because most people who use it lack the programming experience to use its unusual features to write beautiful code. Fortunately, your Ruby knowledge will let you grasp JavaScript's unusual features easily, your SaaS knowledge will let you quickly understand frameworks like jQuery, and your TDD and BDD experience will apply directly to using Jasmine for test-driven JavaScript development.

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*

Brendan Eich proposed embedding the Scheme language in the browser ([Seibel 2009](#)). Although pressure to create a Java-like syntax prevailed, many Scheme ideas survive in 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. In fact, as a language JavaScript inherits almost nothing from Java except superficial syntax. Its object inheritance model comes from Self, its functional and higher-order programming model comes from the Scheme dialect of Lisp, and 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 Chapter [3](#) and are comfortable using CSS selectors as you did in Chapters [2](#) and [5](#), learning and using JavaScript productively will be easy.

There are three major uses of JavaScript in today's SaaS ecosystem:

Creating 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 must be built on some underlying architecture; for example, the [Angular](#) framework for JavaScript supports Model–View–Controller.

Creating server-side apps similar to those we've been building using Rails, but using JavaScript frameworks such as Node.js.

Using JavaScript to enhance the user experience of server-centric SaaS apps that follow the Model–View–Controller architectural pattern. In this case, JavaScript is combined with HTML and CSS to form the “three-legged stool” of client-side SaaS programming. This case is called ***client-side JavaScript*** to clarify that JavaScript is “embedded” in the browser in a way that lets it interact with Web pages.

We focus on case 3, emphasizing two best practices:

- **Graceful degradation:** A site's user experience should be usable, if less flashy, in the absence of JavaScript. (Displaying the message “JavaScript is required for this site” doesn't count.) JavaScript might be absent or incompatibly implemented: [QuirksMode](#) reports that 36% of Chinese Internet users—about 180 million people—use Internet Explorer 6, which has serious JavaScript compatibility problems. JavaScript may be disabled for security reasons, especially in enterprise environments where users cannot change their own configuration settings. We refer to all these cases as legacy browsers, and we insist that our app remain usable in the absence of JavaScript.
- **Unobtrusive:** To make progressive enhancement easier to implement, support legacy browsers, and separate concerns as we did with Model–View–Controller, JavaScript code should be kept completely separate from page markup.

Figure [11.1](#) compares our exposition of server-side and client-side programming.

Screencast [11.1.1](#) demonstrates the two JavaScript features we will add to RottenPotatoes in this chapter.

	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 JSON object, which is used by JavaScript code running in browser to modify current page in place
Debugging	Ruby debugger, rails console	Firebug
TDD	RSpec	Jasmine

Figure 11.1: 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.

Screencast 11.1.1: [Adding JavaScript features to RottenPotatoes](#)

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 [11.4](#) and [11.5](#). 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 [11.6](#) introduces AJAX programming. Both behaviors will be implemented with graceful degradation so that legacy browsers still have a good experience.

JavaScript has a bad reputation that isn’t entirely deserved. It began as a language that would 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. In fact, JavaScript is a powerful and expressive language that incorporates great ideas such as closures and higher-order functions that enable reuse and DRYness, but people without programming experience rarely use these tools properly.

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.

That said, because of JavaScript’s turbulent birth, its syntax and semantics has quirks ranging from the idiosyncratic to the regrettable, with almost as many special-case exceptions as there are rules. In some cases, these may lead you to inadvertently write legal code that doesn’t do what you intended (if you’ve written in Perl, you may know the feeling). As well, there are minor incompatibilities among different versions of JavaScript from different vendors and across different versions. We will avoid compatibility problems in three ways:

Restricting our use to language features in the ECMAScript 3 standard, which all browsers support

Limiting ourselves to the subset of JavaScript that Douglas Crockford recommends in his excellent *JavaScript: The Good Parts* ([Crockford 2008](#))

Introducing and using code quality tools such as [JSLint](#) to identify both potential pitfalls and opportunities to beautify our code.

Section [11.2](#) introduces the language and how code is connected to Web pages and Section [11.3](#) describes its unusual inheritance mechanism, an understanding of



which is the basis of writing clean and unobtrusive JavaScript code. Since JavaScript is the browser’s scripting language, browser functionality must be made accessible to JavaScript code. The JavaScript Application Programming Interface (JSAPI) refers broadly to both the browser functionality exposed to JavaScript and the way that JavaScript code can manipulate the content of the current HTML page. Unfortunately, as is all too often the case with desktop vs. SaaS software (Chapter [1](#)), major incompatibilities across the JSAPIs of different browsers and even different versions of the same browser cause many developer headaches. Fortunately, Section [11.4](#) introduces the powerful [jQuery](#) library, which helps avoid this pitfall by overlaying the separate browsers’ incompatible JSAPIs with a single API that works across all browsers, and Section [11.5](#) describes how jQuery’s features make it easy to program interactions between page elements and JavaScript code.

[jQuery](#) can be viewed as an enhanced Adapter (Section ??) to the various browsers’ JSAPIs.

Section [11.6](#) introduces AJAX programming. 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.

Ironically, modern AJAX programming involves much less XML than originally, as we’ll see.

Finally, testing and debugging client-side JavaScript is challenging because browsers will fail silently when an error occurs rather than displaying JavaScript error

messages to unsuspecting users. Fortunately, debugging and TDD tools are available to help you correct your code, as Section ?? describes.

Summary of JavaScript background:

- JavaScript resembles Java in name and syntax only; despite nontrivial flaws, it embodies great ideas found in Scheme and Ruby.
 - 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. We strive for graceful degradation (user experience without JavaScript should be usable, if impoverished) and unobtrusiveness (JavaScript code should be completely separate from page markup).
-

Self-Check 11.1.1. True or false: one early advantage of JavaScript for form validation (preventing a user from submitting a form with invalid data) was the ability to remove validation code from the server and move it to the client instead.

 False; there is no guarantee the submission actually came from that page (anyone with a command line tool can construct an HTTP request), and even if it did, the user might be working with a legacy browser. As we point out repeatedly in SaaS, the server cannot trust anyone and must always validate its inputs.

Stop me if you think you've heard this before. *variously attributed*

Despite its name and syntax, JavaScript has much in common with Ruby:

- Almost everything is an object (JavaScript objects are basically hashes).
 - Typing is dynamic: variables don't have types, but the objects they refer to do.
 - Classes and types don't matter much (in fact, JavaScript has no classes), only whether an object responds to a function call.
 - 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.
 - JavaScript is interpreted and makes heavy use of metaprogramming.
-

Objects `movie={title: 'The Godfather', 'releaseInfo': {'year': 1972, rating: 'PG'}}`

Quotes optional around property name if it's a legal variable name; objects can be nested.

Access an object's properties with `movie.title`, or

	<code>movie['title']</code> if property name isn't a legal variable name or isn't known until runtime. See the Elaboration in Section ?? for more about the so-called JavaScript Object Notation or JSON.
	<code>for (var in obj) { . . . }</code> iterates over <code>obj</code> 's property names in arbitrary order.
Types	<code>typeof x</code> returns <code>x</code> 's primitive type: one of "object", "string", "array", "number", "boolean", "function", "undefined". All numbers are doubles.
Strings	<code>'string', 'also a string', 'joining'+'strings'</code> <code>'mad, mad world'.split(/[,]+/) ==</code> <code>["mad", "mad", "world"]</code> <code>'mad, mad world'.slice(3,4)==", mad"; '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>
Regexps	<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, unless 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==15, '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</code> (like Ruby <code>nil</code>), <code>undefined</code> (undefined value, different

from `null`), 0, the empty string ' ', and `NaN` (not-a-number) are *falsy* (Boolean false); `true` and all other expressions are *truthy*.

Naming	<code>localVar</code> , <code>local_var</code> , <code>ConstructorFunction</code> , <code>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 11.3 describes). Variables don't have types, but the objects they refer to do.
Control flow	<code>while</code> , <code>for(;;)</code> , <code>if/else if/else</code> , <code>?:</code> (ternary operator), <code>switch/case</code> , <code>try/catch/throw</code> , <code>return</code> , <code>break</code> Statements separated by semicolons; interpreter tries to auto-insert "missing" ones, but this is perilous (see Fallacies & Pitfalls)

Figure 11.2: Analogous to Figure [3.1](#), this table summarizes basic constructs of JavaScript. See the text for important pitfalls.

Figure [11.2](#) shows JavaScript's basic syntax and constructs, which should look familiar to Java and Ruby programmers. As the first row shows, JavaScript's basic object type is an unordered collection of key/value pairs, like a Ruby hash; indeed, Ruby 1.9's alternate hash notation `key: 'value'` mimics the so-called JavaScript Object Notation or JSON. From now on, when we refer to a JavaScript "object," this is what we mean, as opposed to, for example, a `String` or `Array`. In JavaScript it's idiomatic to refer to the hash keys as *slots* or *properties*.

Unlike Ruby, functions in JavaScript are [*first-class objects*](#)—you can pass them around, assign them to variables, and so on—so it's common in JavaScript for an object's properties to be functions. Like Ruby, functions are closures that carry their environment around with them. Figure [11.3](#) illustrates both properties, which are used extensively in AJAX programming.



Try these examples in [Firebug](#) or your browser's built-in JavaScript console.

The Fallacies & Pitfalls section describes several JavaScript pitfalls associated with the figure; read them carefully, or you may find yourself banging your head against a bug that turns out to be the result of one of JavaScript's unfortunate misfeatures. For the moment we continue our whirlwind tour of the language's key unusual features.

<http://pastebin.com/X7nVhdxA>

```
1 // This way looks more familiar:
```

```

2 function logToBase (number, base) { ...; }
3 // But it's syntactic sugar for this:
4 var logToBase = function (number, base) {
5   return( Math.log(number)/Math.log(base) );
6 };
7 logToBase;      // => function (number, base) {...}
8 logToBase(16,2); // => 4
9 // A higher-order function that returns a function:
10 var makeLogToBase = function (logbase) {
11   return( function (number) {
12     return( logToBase(number, logbase) );
13   }); // value of logbase "captured" by closure
14 };
15 var log2 = makeLogToBase(2);
16 log2(16);      // => 4

```

Figure 11.3: While line 2 may look comfortably familiar, lines 4–6 are more idiomatic, clarifying that `log_to_base` is just a variable whose value is a function. Lines 7–8 show that unlike Ruby's flexible syntax, parentheses in JavaScript are necessary if you want to invoke the function named by a variable. The higher-order function defined in lines 10–13 returns a function to compute a logarithm to a given base; this works because functions are closures, so the `log2` function "remembers" the value of `logbase` passed in line 15 to `make_log_to_base`. (Use Pastebin to copy-and-paste this 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 view 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:
<http://pastebin.com/MLSjvnke>

```

1
<script src="/public/javascripts/application.js"></script>

```



The Rails view helper `javascript_include_tag 'application'`, which generates the above tag, can be placed in your `app/views/layouts/application.html.haml` or other layout template 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 concatenate the contents of all JavaScript files in this directory, compress the result by removing whitespace and performing other simple transformations (the uglifier gem), 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, and 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 also allows us to use languages like CoffeeScript, as we'll see later. 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.

Minifying is a term used to describe the compression transformations, which reduce the size of the jQuery 1.7.2 library from 247 KiB to 32 KiB.

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 11.4, jQuery defines a single global function `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 single global variable `RP` to enclose all

JavaScript code in RottenPotatoes.



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.
- 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 in hash values, allowing a hash to collect a set of related functions as a class would.
- 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.

ELABORATION: Obtrusive JavaScript

As the following code fragment shows, there are three other ways to embed JavaScript into HTML pages, all deprecated because they are obtrusive in that they mix JavaScript code with presentational markup. Sadly, all the bad practices shown here are exceedingly common in the “street JavaScript” that pervades poorly-engineered sites, yet every one of them is automatically avoided by using the **script src=** method of including JavaScript as we have recommended and by using the unobtrusive techniques described in the rest of this chapter for connecting JavaScript code to HTML elements.

<http://pastebin.com/sTM4Z8wS>

```
1   <html>
2     <head><title>Update Address</title></head>
3     <body>
4
5       <!-- BAD: embedding scripts directly in page, esp. in body
-->
6       <script>
7         <!-- // BAD: "hide" script body in HTML comment
8             // (modern browsers may not see script at all)
9
10      function checkValid() {    // BAD: checkValid is global
11          if !(fieldsValid(getElementById('addr'))) {
12
13            // BAD: > and < may confuse browser's HTML parser
14            alert('>>> Please fix errors & resubmit. <<<');
15
16            </script>
17
18            <!-- BAD: using HTML attributes for JS event handlers -->
19
20            <form onsubmit="return checkValid()" id="addr" action="/up
21            date">
22              <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>
```

Self-Check 11.2.1.

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 methods are first-class objects in JavaScript, a method name without parentheses would be an expression whose value is the method itself, rather than a call to the method, as lines 7–8 in Figure 11.3 show.

In Chapter 3 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. JavaScript is strongly object-oriented, but lacks classes or inheritance. Instead, a mechanism called ***prototype inheritance*** provides reuse.

Prototypes come from Self, a language originally designed at Xerox PARC, where many of today's modern personal computing technologies were invented. Self's design heavily influenced [NewtonScript](#), the programming language for the ill-fated Apple Newton "handheld."

<http://pastebin.com/EWYr0Ekp>

```
1 var 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   return(true);
9 };
10 // using 'new' makes Movie the new objects' prototype:
11 var pianist = new Movie('The Pianist', 2002, 'R');
12 var chocolat = new Movie('Chocolat', 2000, 'PG-13');
13 pianist.full_title(); // => "The Pianist (2002)"
14 chocolat.full_title(); // => "Chocolat (2000)"
15 // adding methods to an individual object:
16 pianist.title_with_rating = function() {
```

```

17     return (this.title + ': ' + this.rating);
18 }
19 pianist.title_with_rating(); // => "The Pianist: R"
20 chocolat.title_with_rating(); // => Error: 'undefined' not a
  function
21 // but we can add functions to prototype "after the fact":
22 Movie.prototype.title_with_rating = function() {
23     return (this.title + ': ' + this.rating);
24 }
25 chocolat.title_with_rating(); // => "Chocolat: PG-13"
26 // we can also invoke using 'apply' -- 'this' will be bound
  to the
27 // first argument:
28 Movie.full_title.apply(chocolat); // "Chocolat (2000)"
29 // BAD: without 'new', 'this' is bound to global object!!
30 juno = Movie('Juno', 2007, 'PG-13'); // DON'T DO THIS!!
31 juno;           // undefined
32 juno.title;    // error: 'undefined' has no properties
33 juno.full_title(); // error: 'undefined' has no properties

```

Figure 11.4: Prototype inheritance at work. Note the pitfall in lines 29–33. In all browsers except Internet Explorer, `obj.__proto__` reveals an object's prototype. The text's discussion should help you understand why the clumsier notation `obj.constructor.prototype` works in all browsers: every object has a constructor, and every constructor, being a function, has a `prototype` property.

While prototype inheritance can be used to mimic class inheritance, like Ruby mixins it's best treated as a different kind of reuse with its own benefits. The idea of a prototype is simple: similar to a superclass, if you look up a property on an object that doesn't have that property, its prototype is checked, then the prototype of its prototype, and so on up the prototype chain. Every object inherits from exactly one prototype object: new strings inherit from `String.prototype`, new arrays from `Array.prototype`, and so on, except for `Object` (the empty object), which has no prototype and is the top of the prototype chain. If the property is still not found there, the result of the lookup is `undefined`. (Remember that unlike Ruby, which uses `nil` as both an explicit null value and the value returned for undefined instance variables, JavaScript distinguishes `undefined` from `null` and Boolean `false`, though all are “falsy.”) If you define a property on an object, the object's own property “shadows” any property with the same name in its prototype chain. `obj.hasOwnProperty('foo')` returns true if `obj` has a “local” `foo` property or false if the property would be looked up in the prototype (but says nothing about

whether such a lookup would succeed).

Some functions can be designated as **constructor functions** that return a new object. When such a function is called using JavaScript's **new** keyword (which works similarly to Java's), two things happen:

- In the function body, **this** will be bound to a new object (hash) that will eventually be returned by the function—even if the function doesn't have an explicit **return** statement. This behavior is similar to Ruby's **self** inside an **initialize** constructor method.
- The returned JavaScript object will have as its prototype whatever the function's own **prototype** property is set to.

For example, in the call in lines 11–12 of Figure 11.4, JavaScript will bind **this** in the body of **Movie** to a new object whose prototype is the value of **Movie.prototype**, and return the new object (even though there is no **return** statement). In this straightforward case, you can think of **Movie** as the constructor for a “class” like Ruby's **initialize** (although remember JavaScript doesn't have classes) and the returned object as a new instance of that “class.” Keep in mind that while **pianist** and **chocolat** each have their own copies of the **title** property (you can think of it as an instance variable), they “share” **Movie**'s copy of the **full_title** property, which happens to be a function.

Lines 15–25 show that functions can be added to an object or to its prototype after initial creation. Since JavaScript is a dynamic language like Ruby, objects immediately “inherit” new functions added to their prototype, in the sense that prototype lookup occurs every time a function is invoked on an object. Line 20 reminds us that there is no error in referring to the nonexistent property **chocolat.title_with_rating**—JavaScript just returns **undefined**—but it *is* an error to try to use **undefined** as a function! Lastly, line 28 shows that any function can be invoked with **apply**, in which case the first argument to **apply** becomes the value of **this** in the function body.

However, a JavaScript misfeature can trip us up here. It is (unfortunately) perfectly legal to call **Movie** as a plain old function *without* using the **new** keyword, as in lines 29–33. 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 is the object named by **this** outside the scope of any function and the target of any function call lacking an explicit target. (The term *target* in JavaScript is equivalent to the term **receiver** in Ruby.) The global object provides prototypes for JavaScript's basic types such as **Array** and **String**, defines special constants such as **Infinity**, **Nan**, and **null**, and supplies various other parts of the JavaScript environment. But as you can see, the global object has absolutely nothing to do with **Movie**! Second, since **Movie** doesn't explicitly return anything, its return value (and therefore the value of **juno**) will be **undefined**. (The **return** in line 6 belongs to the **full_title** function, not to

`Movie` itself.) Hence, lines 32 and 33 give errors because we're trying to reference a property (`title`) on something that isn't even an object. Note that this is different from line 20, where the error occurs because even though `chocolat` is a valid object, it lacks the property we're asking for, and the error is that we cannot treat a nonexistent property as a function. Figure 11.5 summarizes this unfortunate difference in behaviors, which is considered one of JavaScript's worst design defects.

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`.

	Called as constructor:	Called as regular function:
Syntax	<code>foo = new Foo()</code>	<code>foo = Foo()</code>
<code>this</code> value in function body	new object that will eventually be returned (like Ruby <code>self</code> or Java <code>this</code>)	global object
Return value	New object	<code>undefined</code> , since no <code>return</code> statement
Returned object's prototype	<code>Foo.prototype</code>	N/A since <code>undefined</code> is returned

Figure 11.5: Calling a function as a constructor using `new` vs. doing a “bare” function call results in dramatically different behaviors. Using `new` allows the returned object to call function-valued properties on its prototype, as in lines 10–11 of Figure 11.4; some JavaScript programmers refer to this as a “method call” rather than a function call, though JavaScript makes no distinction.

Summary: Prototype inheritance

- All objects except the root object have a prototype. If an object property is referenced and not found, the lookup is “passed up the chain” of prototypes until either some parent object responds or we reach the top of the chain, in which case `undefined` is returned. This prototype-based inheritance is the basis of reuse in JavaScript.
- If the `new` keyword is used to call a function, `this` in the function body will refer to a new object whose prototype matches the function's own prototype. This mechanism is similar to creating new instances of a class, though JavaScript lacks classes.
- However, beware that if a function intended to be a constructor is called *without* the `new` keyword, `this` will refer to the global object, which is almost

never what you wanted, and the function will return `undefined` (since constructor functions don't have an explicit `return`). To avoid the pitfall, capitalize the names of all and only those functions intended to be called as constructors with `new`.

ELABORATION: Private properties

A possible disadvantage of prototype inheritance is that unlike languages with class hierarchies, all attributes are public. (Recall that in Ruby, *no* attributes are public: getter and setter methods, defined either explicitly or using `attr_accessor`, are the only way to access attributes from outside the class.) However, we can take advantage of closures to get private attributes. Here is a simple constructor for a user "class" in which the password cannot be inspected by another object, but a `checkPassword` method tells whether a supplied password is correct. This "accessors only" idiom is used throughout jQuery, which we'll meet shortly. (We don't call the constructor `User` because we follow the convention that capitalization implies the need to use `new` with the constructor, which isn't the case here.)

<http://pastebin.com/8FFvMKq0>

```
1  var new_user = function (attribs) {
2      var secretPassword;          // private attribute
3      var user = attribs;         // object we will return
4      // public properties and methods:
5      user.setPassword = function (newPassword) {
6          secretPassword = newPassword;
7      }
8      user.checkPassword = function (tryPassword) {
9          return (tryPassword == secretPassword);
10     }
11     return (user);
12 };
13 dave = new_user({name: 'dave'}).setPassword('patterson');
14 dave.checkPassword('patt');        // => false
15 dave.checkPassword('patterson'); // => true
16 dave.secretPassword;            // => undefined
17
```



Self-Check 11.3.1. What is the difference between evaluating `square.area` and `square.area()` in the following JavaScript code?

<http://pastebin.com/jgHWhCwU>

```
1  var square = {  
2      side: 3,  
3      area: function() {  
4          return this.side*this.side;  
5      }  
6  };
```

➊ `square.area()` is a function call that in this case will return 9, whereas `square.area` is an unapplied function object.

Self-Check 11.3.2. Write down another way to call `area` using `apply` so that the return value is 9. (Hint: see Figure 11.4.)

➋ <http://pastebin.com/mSEkNWRE>

```
1  square.area.apply(square)
```

Self-Check 11.3.3. Given the code in Self-Check 11.3.1, explain why it's is incorrect to write `s=new square`.

➌ `square` is just an object, not a function, so it cannot be called as a constructor (or at all).

Self-Check 11.3.4. True or false: continuing the example in Figure 11.4, changing `Movie`'s prototype as follows will cause all existing "instances" of `Movie` to inherit the `for_children` method:

<http://pastebin.com/7gJNdEME>

```
1  Movie.prototype = {  
2      for_children: function () { return( /^G|  
3      PG$/.test(this)); }  
3  };
```

False. Giving a new value to `Movie.prototype` affects the objects created by *future* calls to `Movie`—existing instances already have as their prototype the object originally referenced by `Movie.prototype`. However, we can get the desired behavior by adding a property to the *existing* prototype:

<http://pastebin.com/V17ng7yH>

```
1   Movie.prototype.for_children = function () {  
2       return ( /^G|PG$/.test(this));  
3   };
```

DOM technically refers to the DOM standard itself, but developers often use it to mean the specific DOM tree corresponding to the current page.

The [World Wide Web Consortium Document Object Model \(W3C DOM\)](#) 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 [11.6](#) 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. By calling functions defined by the global object, JavaScript code can query and traverse the DOM of the currently-loaded page, and more importantly, by modifying the DOM (adding elements, removing elements, or modifying element contents or attributes), cause the browser to redraw parts of the page.

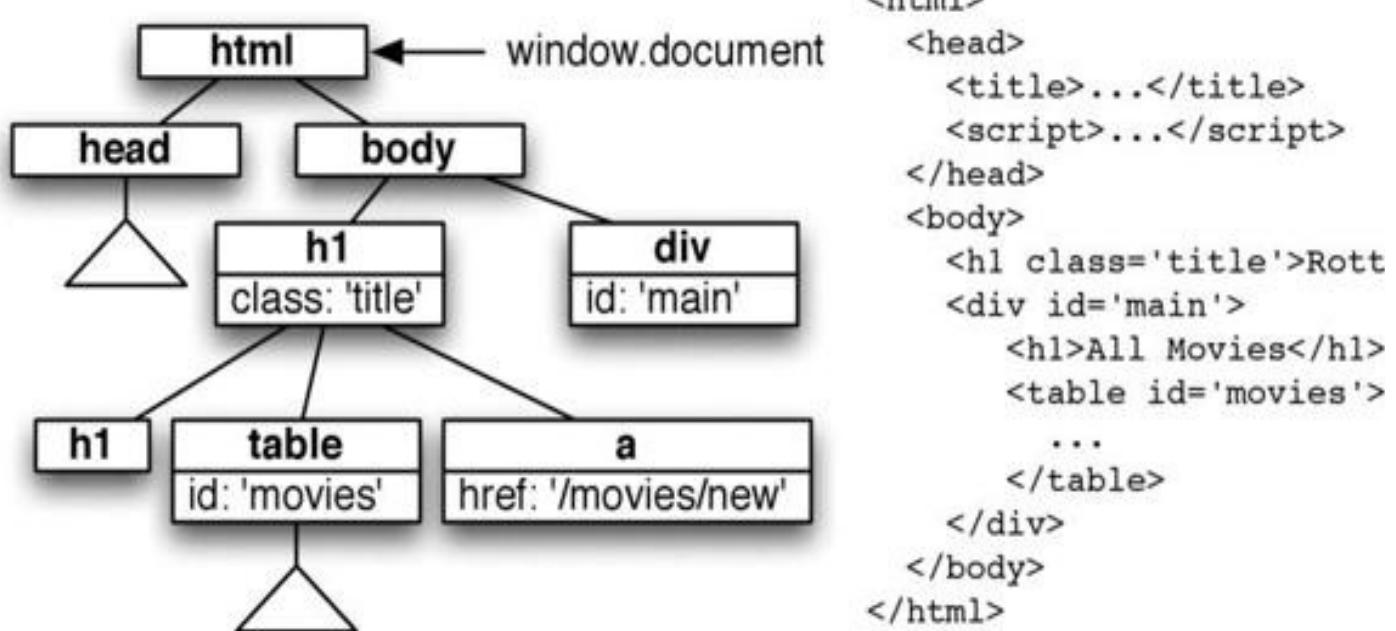


Figure 11.6: A simplified view of the DOM tree corresponding to the RottenPotatoes “list of movies” page with skeletal HTML markup. An open triangle indicates places where we’ve elided the rest of the subtree for brevity. `this.document` is set to point to the DOM tree’s root when a page is loaded.

How does JavaScript get access to the DOM? When JavaScript is embedded in a browser, the global object defined by the browser, named by the global variable `window`, defines additional properties and functions specific to Web browsers, including JavaScript functions for interacting with the loaded HTML page’s DOM. This set of functions is collectively referred to as the JSAPI. Unfortunately, whereas most browsers implement the JavaScript language itself fairly consistently, they differ considerably in their implementations of the JSAPI. For this reason, we instead use the more productive API provided by the JavaScript library [jQuery](#), which is layered on top of each browser’s native JSAPI and hides most cross-browser differences. jQuery also adds additional features and behaviors absent from the native JSAPIs, such as animations and better support for CSS and AJAX (Section [11.6](#)).

[quirksmode.org](#) tells you more about JSAPI browser incompatibilities than you want to know.

Property of `window`

`location`

`document`

Value/description

The URL of the current document; changing this property’s value causes the browser to reload from a new URL.

The root of the DOM tree for the current document. See Figure [11.8](#) for properties and methods that jQuery makes available on any DOM node, including this root.

<code>document.head</code> , <code>document.body</code>	DOM elements representing the head and body of the HTML page, available even if the page was ill-formed and lacked either of these elements.
<code>document.write(string)</code>	Appends <i>string</i> to the end of the current document. Largely obsoleted by jQuery's more expressive API.
<code>alert(string)</code> , <code>confirm(string)</code>	Display a pop-up alert box showing <i>string</i> , which must be plain text. <code>alert</code> displays only an OK button; <code>confirm</code> displays OK and Cancel buttons and returns <code>true</code> if the user clicked OK for <code>false</code> if Cancel.

Figure 11.7: Some of the additional browser-specific properties provided by `window`, which serves as the global object for browser-embedded JavaScript.

When a new page is loaded, a new global object is created that shares no data with the global objects of other visible pages. Figure 11.7 shows some of the JSAPI properties and functions defined by the global object for interacting with the browser. In particular, the value of the `document` property is a DOM node that refers to the root of the current document's DOM tree. However, instead of using browser-defined DOM elements directly, we'll use jQuery's "enhanced" DOM objects, and instead of using the browser's built-in JSAPI functions for querying the DOM, we'll use jQuery's superior versions of those functions.

jQuery defines a global function `jQuery()` (aliased as `$()`) that is [polymorphic](#): its behavior depends on the number and types of arguments with which it's called.

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 such as [Prototype](#) that also tries to define `$`.

The most common such call is to pass a CSS selector that identifies one or more of the current page's DOM elements, such as `jQuery('#movies')` or `$('#movies')` or `$('.h1.title')`. (Recall from Section 2.3 that these selectors mean respectively "the unique element whose `id` is `movies`" and "any `h1` element that has the `title` CSS class". Figure 2.5 shows some common types of CSS selectors.) The returned value is a node set (collection of one or more elements) matching the selector, or `null` if there were no matches. You can then do various things with the elements in the node set, as Figure 11.8 shows:

- To change an element's visual appearance, use jQuery to change the CSS class(es) of the element.
- To change an element's content, use jQuery to set the element's markup content or plain-text content.

- To animate an element (show/hide, fade in/out, and so on), invoke a JavaScript library function on that element that manipulates the DOM to achieve the desired effect.

Screencast [11.4.1](#) shows some simple examples of these behaviors from the browser's JavaScript console.

Screencast 11.4.1: [Manipulating the DOM with jQuery](#)

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 that modifying the elements causes the browser to redraw them. This behavior is the key to client-side JavaScript.

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. As we'll see, following the Iterator design pattern, jQuery provides an `each` iterator to walk through this collection.

Property or function

`$(dom-element)`

`$(this)`

`is(cond)`

`addClass()`, `removeClass()`,
`hasClass()`

`insertBefore()`, `insertAfter()`

`remove()`

Value/description

Inside a handler, returns a jQuery object corresponding to the DOM object that is the argument.

Test if the element is '`:checked`', '`:selected`', '`:enabled`', '`:disabled`'. Note that these strings were chosen to resemble Ruby symbols, though JavaScript doesn't have symbols.

Shortcuts for manipulating the `class` 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.

Insert the element(s) describe by the argument before or after the target.

Remove the target element(s) from the

`replaceWith(new)`

DOM.

Replace the target element(s) with the *new* element(s) provided.

`clone()`

Return a complete copy of the target element, recursively cloning its descendants.

`html(), text()`

Return or set 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.

`val()`

Return or set 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.

`attr(attr,[newval])`
`$('img').attr('src',
'http://imgur.com/xyz')`
`hide(duration,callback), show(),
toggle()`
`slideUp(), slideDown(),
slideToggle()`
`fadeOut(), fadeIn(),
fadeTo(duration,target,callback)`

Return or (with second argument) set the value of the given attribute on the element.

Hide or show elements selected by *sel*. Optional *duration* is one of '`fast`', '`slow`', or the integer number of milliseconds that the animation should last. Optional *callback* is a function to call when animation completes. Other sets of animations with same arguments include `slideDown/slideUp/slideToggle` and `fadeOut/fadeIn`. For `fadeTo`, second argument is target opacity, from 0.0 (transparent) to 1.0 (opaque).

`e.evt(func)`
`$('li').click(function() {
 $(this).hide();
});`

Set *func* as the handler for event *evt* on the element(s) selected by *sel*. *func* can be an anonymous function or a named function. See Figure 11.9 for some of the most important event types.

Figure 11.8: 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 line 8 of Figure 11.11. See the [jQuery documentation](#) for more features beyond this subset.

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 provided 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 JSAPIs and adds many enhanced functions such as CSS-based DOM traversal, animation, and other special effects.

Self-Check 11.4.1. 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 11.4.2. 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.

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.

Dynamic HTML is an unofficial and obsolete term that some vendors used to describe the effects achievable by combining HTML, JavaScript-based DOM manipulation, and CSS.

Events on arbitrary elements	<code>click, dblclick, mousedown/mouseup, mouseenter/mouseleave, keypress</code> (<code>event.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 or text fields, menus)	<code>change</code> fires when any control's state or content is changed. <code>select</code> (user selects text; string <code>event.which</code> is selected text) <code>submit</code> fires when the user attempts to submit the form by any means.

Figure 11.9: A subset of the most important JavaScript events defined by the jQuery API. Set a handler for an event with `element.bind('evt', func)` or as a shortcut, `element.evt(func)`. Hence, `$('#h1').bind('click', function() {...})` is equivalent to `$('#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.

Figure 11.9 summarizes the most important events defined by the browser's native JS API 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**, 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`. 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, 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 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.

There is one last item to consider in putting all the pieces together. 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, or else we might be trying to bind callbacks on elements that don't exist yet! Calling `jQuery()` (alias `$()`) and passing it a function will cause the passed-in function to be executed once the page is finished loading. (Recall that `$()` is polymorphic; Figure 11.10 shows the other ways to call it.)

Uses of `$()` or `jQuery()` with example

```
$(sel)
$('.mov span')

$(elt)
$(this), $(document),
$(document.getElementById('main'))

$(HTML[, attrs])
$(<p><b>bold</b>words</p>),
$(<img/>, {
src: '/rp.gif',
click: handleImgClick })
```

```
$(func)
(function () {. . .});
```

Value/side effects

return collection of elements selected by CSS3 selector *sel*

Wraps the element in a jQuery object to which the operations in Figure 11.8 can be applied

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 *attrs*, it is used to construct the element's attributes. (The new element is not inserted into the document; see the following table for how to do that.)

Run the provided function once the document has finished loading and the DOM is ready to be manipulated. A wrapper around the `onLoad()` handler of the browser's JSAPI, and a shortcut for `$(document).ready(func)`.

Figure 11.10: The four ways to invoke the function `jQuery()` or `$()` and the effects of each.

Putting all these pieces together, the code in Figure 11.11 does the following, after the document is ready. To stay unobtrusive, we collect the three functions that make up our code into a single global hash `RP` with three function-valued properties.

<http://pastebin.com/85fZTe4J>

```
1 RP = {
2     setup: function() {
3         // construct new DOM elements
4         $('<label for="filter" class="explanation">' +
5             'Restrict to movies suitable for children' +
6             '</label>' +
7             '<input type="checkbox" id="filter"/>' +
8             ).insertBefore('#movies').change(RP.filter_adult);
9     },
10    filter_adult: function () {
11        // 'this' is element that received event (checkbox)
12        if ($(this).is(':checked')) {
13            $('#movies tbody tr').each(RP.hide_if_adult_row);
14        } else {
15            $('#movies tbody tr').show();
16        };
17    },
18    hide_if_adult_row: function() {
19        if (! /^G|PG$/i.test($(this).find('td:nth-
child(2)').text())) {
20            $(this).hide();
21        }
22    }
23 }
24 $(RP.setup);           // when document ready, run setup code
```

Figure 11.11: Using jQuery to add a simple behavior to RottenPotatoes' list of movies page. 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

Function `setup` (lines 2–12) will be called when the document is ready (line 24). It first creates a new checkbox and its accompanying explanatory label by specifying some HTML markup as a string literal and then calling `jQuery` (alias `$`) on it to convert it into a jQuery DOM element. Line 8 does two things: it inserts our new elements immediately before the element whose ID is `movies` (namely the movies table), and it binds the `filter_adult` function to the `change` event on the checkbox. Most jQuery functions such as `insertBefore` return the target object itself, allowing “chaining” of function calls as we’ve seen in Ruby.

`<input> outside <form>`? Yes—it’s legal in HTML 4 and later, as long as you manage all the input’s behaviors yourself, as we’ve done.

Why did we bind the `change` event rather than a `click` event on the check box? `change` is 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.

The `filter_adult` function is where the action happens. Because of line 8, this function is called whenever the check box’s state changes, with `this` bound to the check box element itself. But note that `this` refers to the browser’s JSAPI DOM element: to endow it with jQuery behaviors, we must call `$()` to wrap it, which is why line 12 refers to `$(this)` rather than just `this`. The callback function must examine the condition of the check box (line 12) to determine what to do; recall from Figure 11.8 that `:checked` is one of jQuery’s built-in behaviors for checking the state of an element. If the box is unchecked (line 15), we just show all the table rows (technically redundant since not all rows were hidden, but this is more readable and has the same effect). If checked (line 13), we cycle through all the table rows using jQuery’s `each` iterator, which expects to be passed a function to be called on each element. Similar to Ruby `each`, inside that function `this` takes on the value of each entity in the collection in turn.

The function we pass to the iterator is `hide_if_adult_row` (lines 18–22), which hides a row whose second column contains anything other than G or PG. `find` locates elements within the subtree of its target that match a CSS selector. (Indeed, you can think of `$('#movie')` as a shortcut for `$(document).find('#movie')`.) In this case, `this` is a table row yielded by `each`, so `find` will locate elements that are within that table row’s subtree. The CSS selector `td:nth-child(2)` shows one of the more sophisticated features of CSS: each `tr` element has four `td` children (one for each table column), but we’re only interested in the second column, which contains the movie rating. `text()` returns that `td` element’s textual content with any HTML tags stripped out. (A companion function `html()` returns the complete HTML markup of the table cell including any HTML tags used for formatting it.) Various design decisions support our best practices of graceful degradation and

unobtrusiveness:

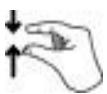
Graceful degradation. Because the checkbox is added by our `setup` function, legacy browsers will not render the checkbox at all. If the checkbox was part of the HTML page, users of such browsers would see the checkbox, but nothing would happen when they clicked on it.

Unobtrusive. Perhaps if we were certain the browser had JavaScript enabled, we could have included the checkbox in the HTML and then used inline-HTML notation to attach the handler to the checkbox—a technique commonly seen in poorly-factored pages. But just as we strive to keep Ruby code out of Rails views, separation of concerns leads to more modular code that can be separately debugged and more easily maintained.



We also did additional things to keep our code beautiful:

- We added only one object to the global namespace, `RP`. However, that means all our functions are just properties of that object, which is why we must include `RP`. when referencing the functions in lines 8, 13, and 24. As well, commas are needed in lines 9 and 17 because the property/value pairs of an object must be separated by commas. (CoffeeScript, introduced briefly in To Learn More, tries to remedy the “punctuation soup” resulting from JavaScript’s heavy use of objects and anonymous functions.)
- We used a `label` element to tie an explanatory label to our checkbox. This connection helps browsers support users with disabilities and provide keyboard shortcuts. The label has its own CSS class so we can style it in the CSS file, even though the style will be unused by legacy browsers.



One last point worth mentioning is an alternative and more concise implementation of our desired feature. Suppose we modify our server-side code to identify rows for adult movies by attaching a special CSS class to them, changing line 13 of Figure 4.5 as follows:

<http://pastebin.com/bcZFwH2w>

```
1      %tr{:class => ('adult' unless movie.rating =~ /^G|  
PG$/)}{}
```

Then `hide_if_adult_row` (lines 18–22 of Figure 11.11) could be deleted, and line 13 could be changed as follows:

<http://pastebin.com/9hSQ5Kv>

```
1      $('#movies tr.adult').hide();
```

The lesson is simple: think about where the server can best help and where the browser can best help in implementing a desired feature. `td:nth-child(2)` is certainly a clever use of CSS, but you don't get points for cleverness. 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.

- You can set or override how various HTML elements react to user input by binding JavaScript handlers to specific events on specific elements. (Handlers are also referred to as callbacks.) jQuery allows you to bind both "true" user events such as clicking the mouse and "logical" pseudo-events such as form submission.
- Inside an event handler, `this` is bound to the browser's DOM representation of the element that received the event. We usually pass the DOM element to jQuery to get `$(this)`, giving us a "jQuery-wrapped" element that supports enhanced jQuery operations, such as `$(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 `$()` causes the `document.ready` property to be set to that function.

ELABORATION: Custom events

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. For example, 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 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.



Self-Check 11.5.1. Explain why calling `$(selector)` is equivalent to calling `$(document).find(selector)`.

• `find` locates all elements matching the selector that are in the subtree of its target, and `document` points to the root of the entire document.

Self-Check 11.5.2. In Self-Check 11.5.1, why did we need to write `$(document).find` rather than `document.find`?

• `document` is a property of the browser's built-in global object that refers to the browser's representation of the root of the DOM. But `find` is a jQuery function, so `$(document)` "wraps" the DOM object in jQuery's enhanced representation to give it access to jQuery functions.

Self-Check 11.5.3. What would happen if we omitted the last line of Figure 11.11, which calls `$(RP.setup)`?

• 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 line 8 of the `setup` function binds our JavaScript handler for the checkbox's `change` event.

In 1998, Microsoft added a new function to the JavaScript global object defined by Internet Explorer 5. `Xm1HttpRequest` (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. The result was a much more 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:

You create a controller action (or modify an existing one) to handle the `xhr` requests made by your JavaScript code (Section 4.4). Rather than rendering an entire view, the action will render a partial (Section 7.1) to generate a chunk of HTML for insertion into the page.

You 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 facilities for many common cases.

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 11.5) that will be triggered when the server responds or an error occurs.

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. Recall that 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.



Let's illustrate how each step works for the feature shown in Screencast [11.1.1](#), in which movie details appear 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 [11.12\(a\)](#) 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.

We can determine whether `show` was called from an AJAX handler because 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 [11.12\(b\)](#) shows the controller action that will render the partial.

<http://pastebin.com/TUcBd0WD>

```
1 %p= movie.description
2
3 = link_to 'Edit Movie', edit_movie_path(movie)
4 = link_to 'Close', '', {:id => 'closeLink'}
```

<http://pastebin.com/ajk95r8D>

```
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 and return
if request.xhr?
6   # will render app/views/movies/show.<extension> by defau
lt
7 end
8 end

```

Figure 11.12: (a) Top: a simple partial that will be rendered and returned to the AJAX request. (b) Bottom: The controller action that renders the partial, obtained by a simple change to Figure 4.8: if the request as 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.html.haml` view as usual.

<http://pastebin.com/SRNyd3j2>

```

1 RP = {
2   setup: function() {
3     // add invisible 'div' to end of page:
4     $('<div id="movieInfo"></div>').
5       hide().
6       appendTo($('body'));
7     $('#movies a').click(RP.getMovieInfo);
8   },
9   getMovieInfo: function() {
10    $.ajax({type: 'GET',
11           url: $(this).attr('href'),
12           timeout: 5000,
13           success: RP.showMovieInfo,
14           error: function() { alert('Error!'); }
15         });
16    return(false);
17  },
18  showMovieInfo: function(data) {
19    // center a floater 1/2 as wide and 1/4 as tall as s
creen
20    var oneFourth = Math.ceil($(window).width() / 4);
21    $('#movieInfo').
22      html(data).
23      css({'left': oneFourth, 'width': 2*oneFourth, 'zIndex': 1000});
24  }
}

```

```

top': 250}).

24      show();
25      // make the Close link in the hidden element work
26      $('#closeLink').click(RP.hideMovieInfo);
27      return(false); // prevent default link action
28  },
29  hideMovieInfo: function() {
30      $('#movieInfo').hide();
31      return(false);
32  },
33 }
34 $(RP.setup);

```

Figure 11.13: 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 `ajax` are available, in particular for more robust error handling.

Moving on to step 2, how should our JavaScript code construct and fire off the XHR request? As the screencast showed, we want the floating window to appear when we click on the link that has the movie name. As Section 11.5 explained, we can “hijack” the built-in behavior of an element by attaching an explicit JavaScript `click` handler to it.

| **Hijax** is sometimes humorously used to describe this technique.

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 11.5, our `setup` function (lines 2–8 of Figure 11.13) 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 Figure 11.13 shows.

| You’ve probably already guessed that `$.ajax` is an alias for `jQuery.ajax`.

Screencast 11.6.1 uses the Firebug interactive debugger as well as the Rails debugger to step through the rest of the code in Figure 11.13. 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 11 shows. Lines 13–14 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.

<http://pastebin.com/kBgDCnVB>

```
1 #movieInfo {  
2   padding: 2ex;  
3   position: absolute;  
4   border: 2px double grey;  
5   background: wheat;  
6 }
```

Figure 11.14: 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.

Screencast 11.6.1: [Interactively single-stepping through AJAX](#)

AJAX debugging requires a combination of a JavaScript debugger such as Firebug and a server-side debugger such as `ruby-debug`, which you met in Chapter [4](#).

Some interesting CSS trickery happens in lines 20 and 23 of Figure [11.13](#). 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 [11.14](#). 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` heuristically computes the dimensions and coordinates of a floating div half as high 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 an element, so 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.)

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.
 - For graceful degradation, construct the extra AJAX elements and bind any JavaScript handlers in your setup function rather than including that functionality on the HTML page itself.
-

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 logic 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 [WebWorkers](#) 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 these must be managed explicitly by the application.

Self-Check 11.6.1. In Figure [11.12](#), what would happen on an XHR request if we omitted `and return` from line 5?

 The controller action would raise a `DoubleRenderError` since line 5 renders a partial but the controller’s default behavior once the method terminates is to render a whole view.

Self-Check 11.6.2. In line 13 of Figure [11.13](#), why did we write `RP.showMovieInfo` instead of `RP.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 11.6.3. In line 34 of Figure 11.13, why did we write `$(RP.setup)` rather than `$('RP.setup')` or `$(RP.setup())`?

 We need to pass the actual function to `$`, not its name or the result of calling it.

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 a code quality tool such as [JSLint](#) to warn you of both potential pitfalls and opportunities to beautify your JavaScript code. Some specific pitfalls to avoid



include the following:

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:

<http://pastebin.com/hwpLn8hq>

```
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};
```

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 8.4?) 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.

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.

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 get the value `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 following code shows:

<http://pastebin.com/zjyFXTwM>

```
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' is
n't true object
6 fake.newprop      // => undefined
7
```

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 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.)

 **Pitfall: Creating a site that fails without JavaScript rather than being enhanced by it.**

For reasons of accessibility by people with disabilities, security, cross-browser compatibility, and widely-varying Internet performance (10-100 megabits per second from home or office vs. 100 to 1000 kilobits per second on a smart phone or tablet), a well-designed site should work *better* if JavaScript is available, but *acceptably* otherwise. 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, leaving more expensive JavaScript testing for the UI itself. 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.

 **Pitfall: Silent JavaScript failures.**

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 (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 `$.get(url,data,callback)`, `$.post(url,data,callback)`, and `$.load(url_and_selector)` as wrappers around the `$.ajax()` method we saw in Section 11.6, 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.

 **Fallacy: AJAX will surely improve my app’s responsiveness because more action happens right in the browser.**

In a carefully-engineered app, AJAX may well have the *potential* to improve responsiveness of certain interactions. However, many factors in using AJAX also work against this goal. Your JavaScript code must be fetched from the server, as must any libraries or frameworks on which it relies, such as jQuery; on platforms such as mobile phones, this may incur an up-front latency that negates any later savings. Wide variation in JavaScript performance across different browser types

and versions, Internet connection speeds spanning a range from 1 Mbps (smart phones) to 1000 Mbps (high-speed wired networks), and other factors beyond your control, all conspire to make overall AJAX performance effects difficult to predict; in some cases, AJAX may slow things down. Like all powerful tools, AJAX should be used with a solid understanding of precisely how and why it will improve responsiveness, rather than added on in the vague hope that it will somehow help because the app feels slow. The techniques in Chapter [12](#) will help you identify and resolve some common performance problems.

 **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. JavaScript's privileged position as the client-side language of the Web has focused a lot of energy on it. Because of increased reliance on JavaScript for both "Web 2.0" sites and complex Web apps such as Google Docs, attention has turned increasingly to code beauty and productivity in JavaScript. As well, many developers have discovered that rich mobile device apps can be created using HTML, CSS and JavaScript, rather than creating separate versions for different mobile platforms such as iOS and Android. Frameworks like PhoneGap make JavaScript a productive path to creating mobile apps, especially when combined with flexible UI frameworks such as [jQuery Mobile](#) or Sencha Touch.

There is also a focus on performance, as just-in-time compilation (JIT) techniques and other advanced language engineering features are brought to bear on the language. As well, JavaScript is one of the first languages to receive attention when new hardware becomes available that could be useful for user-facing apps: WebCL proposes JavaScript bindings for the OpenCL language used for programming Graphics Processing Units (GPUs). Evaluating the performance of interpreted languages on such applications is tricky, since results depend on the implementation of the interpreter itself as well as the specific application, but [benchmarks of the Box2D physics engine](#) found JavaScript to be 5x slower than Java and 10–12x slower than C. 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. Indeed, the same benchmark measured performances differences of 3x on the same app using different JavaScript

interpreters.

All these trends suggest that JavaScript is well on the way to becoming the language of choice for mobile client applications: you need a good reason to write the app in native code, usually because of either a demonstrably sluggish user experience or the inability to access specific hardware functionality on the mobile device. Indeed, in May 2011 Hewlett-Packard rewrote large parts of its Palm webOS operating system for mobile devices in JavaScript.

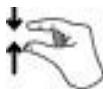
JavaScript's single-threaded execution model seems unlikely to change anytime soon. Some bemoan the adoption of JavaScript-based server-side frameworks such as Node, a JavaScript library of event-driven versions of the same POSIX operating system facilities used by task-parallel code. Rails core committer 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 external stimuli like user-initiated user interface events, the asynchronous model makes more sense.

Finally, there is a trend toward SaaS apps in which a skeletal client page is served and performs AJAX calls to retrieve JSON content from the server and perform rendering on the client side. Indeed, Yahoo's open-source [Mojito](#) framework allows the same JavaScript code to render HTML from JSON on either the client or a Node-based server. However, there is a potential huge downside to apps taking this route: their content will be neither indexable nor searchable by search engines, without which the Web loses a great deal of its utility. There are technological solutions to this problem, but at the moment there is little discussion about them.

Finally, whereas in the early days of the Web it was common for pages to be hand-authored in HTML and CSS (perhaps using WYSIWYG authoring tools), today the vast majority of HTML is generated by frameworks like Rails. In a similar way, developments such as CoffeeScript (see Section [11.9](#)) suggest that while JavaScript will remain the *lingua franca* of browser programming, it may increasingly become a target language rather than the one in which most people code directly.

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](#) library for Java and the [Nokogiri](#) gem for Ruby.

Despite some elegant features, JavaScript's syntax is hardly concise, in part because of the marketing-driven requirement that it resemble Java. [CoffeeScript](#) tries to restore some syntactic conciseness and beauty befitting JavaScript's better nature. A source-to-source translator compiles CoffeeScript (.coffee) files into .js files containing regular JavaScript, which are served to the browser. The Rails asset pipeline automates this compilation so you don't need to manually generate the .js files or include them in your source tree. Since CoffeeScript compiles to JavaScript, it can't do anything that JavaScript doesn't already do, but it provides more concise syntactic notation for many common constructs. As an example, Figure [11.15](#) shows the CoffeeScript version of the jQuery example of Figure [11.11](#).



<http://pastebin.com/1568HJag>

```
1 RP =
2   setup: ->
3     # construct new DOM elements
4     $('<label for="filter" class="explanation">' +
5       'Restrict to movies suitable for children' +
6       '</label>' +
7       '<input type="checkbox" id="filter"/>' +
8       ).insertBefore('#movies').click(RP.filter_adult)
9   filter_adult: ->
10    # 'this' is element that received event (checkbox)
11    if $(this).is(':checked')
12      $('#movies tbody tr').each(RP.hide_if_adult_row)
13    else
14      $('#movies tbody tr').show()
15   hide_if_adult_row: ->
16     $(this).hide() unless /^G|
PG$/i.test($(this).find('td:nth-child(2)').text())
17 $(RP.setup)          # when document ready, run setup code
18
```

Figure 11.15: This is the CoffeeScript version of Figure 11.11. Among other differences, CoffeeScript provides a [Haskell](#)-like concise syntax for anonymous functions, uses Haml-like indentation rather than braces to indicate structure, and allows Ruby-like omission of most parentheses as well as borrowing the `@` instance-variable notation to refer to properties of `this`. Your author has found that his most common JavaScript mistakes involve mismatched braces and parentheses where anonymous functions and properties are concerned, so he appreciates that aspect of CoffeeScript in particular.

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](#)
- jQuery is an extremely powerful library whose potential we barely tapped. *jQuery: Novice to Ninja* ([Castledine and Sharkey 2010](#)) is an excellent reference with many examples that go far beyond our introduction.

- *JavaScript: The Good Parts* ([Crockford 2008](#)), by the creator of the [JSLint](#) 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.

E. Castledine and C. Sharkey. *jQuery: Novice to Ninja*. SitePoint Books, 2010.

D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 2008.

P. Seibel. *Coders at Work: Reflections on the Craft of Programming*. Apress, 2009.

ISBN 1430219483. URL <http://www.amazon.com/Coders-Work-Reflections-Craft-Programming/dp/1430219483>.

Project 11.1.

Write JavaScript to create cascading menus for day, month and year that allow the entry of valid dates only. For example, if February is selected as the month, the Day menu should only go from 1–28, unless the Year menu indicates a leap year, in which case the Day menu should go from 1–29, and so on.

As a bonus, wrap your JavaScript in a Rails helper that results in date menus with the same menu names and option tags as the Rails' built-in helpers, making your JavaScript menus a drop-in replacement. **Note:** it's important that the menus also work in non-JavaScript-enabled browsers; in that case, the menus should statically display 1–31 for the days of the month.

Project 11.2.

Create the AJAX code necessary to create cascading menus based on a `has_many` association. That is, given Rails models A and B where A `has_many` Bs, the first menu in the pair should list the A choices, and when one is selected, retrieve the corresponding B choices and populate the B menu.

Project 11.3.

Augment the validation functionality in ActiveModel (which we met in Chapter [7](#)) to automatically generate JavaScript code that validates form inputs before the form is submitted. For example, given RottenPotatoes' `Movie` model asserts that a movie must have a nonblank title, JavaScript code should prevent the “Add New Movie” form from being submitted if the validation is not met, displays a helpful message to the user, and highlights the field(s) that had validation problems. Handle at least the built-in validations such as nonblank, minimum/maximum string lengths, and numericality.

Project 11.4.

Following the approach of the jQuery example in Section [11.5](#), use JavaScript to implement a set of checkboxes for the list of movies page, one for each rating (G, PG, and so on), which when checked allow movies with that rating to stay in the list. When the page first loads, all should be checked; unchecking any of them should hide the movies with that rating.

Project 11.5.

Extend the example of Section [11.6](#) so that if the user repeatedly expands and collapses the same row in the movies table, a request to the server for that movie's info is only made the first time. In other words, implement client-side JavaScript

caching for the movie info retrieved on each AJAX call.

Project 11.6. If you visit `twitter.com` and the page takes more than a few seconds to load, a popup appears apologizing for the delay and suggesting you try reloading the page. Explain how you would implement this behavior using JavaScript. **Hint:** Remember that JavaScript code can begin executing as soon as it's loaded, whereas the `document.ready` function won't run until the document has been completely loaded and parsed.



Barbara Liskov (1939–) 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*

- 12.1 [From Development to Deployment](#)
- 12.2 [Quantifying Availability and Responsiveness](#)
- 12.3 [Continuous Integration and Continuous Deployment](#)
- 12.4 [Upgrades and Feature Flags](#)
- 12.5 [Monitoring and Finding Bottlenecks](#)
- 12.6 [Improving Rendering and Database Performance With Caching](#)
- 12.7 [Avoiding Abusive Queries](#)
- 12.8 [Defending Customer Data](#)
- 12.9 [Fallacies and Pitfalls](#)
- 12.10 [Concluding Remarks: Performance, Security, and Leaky Abstractions](#)
- 12.11 [To Learn More](#)
- 12.12 [Suggested Projects](#)

Unlike shrink-wrapped software, SaaS developers are typically much closer to post-release operations and maintenance. This chapter covers what your SaaS app should *not* do when released: crash, become unresponsive when it experiences a surge in popularity, or compromise customer data. Since many of these concerns are greatly alleviated by deploying in a well-curated PaaS (Platform-as-a-Service) environment such as Heroku, we focus on how to steward your app to leverage those benefits as long as possible by monitoring to identify problems that interfere with responsive service, addressing those problems with caching and efficient database usage, and thwarting common attacks against customer data.

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 accidental 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. 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.

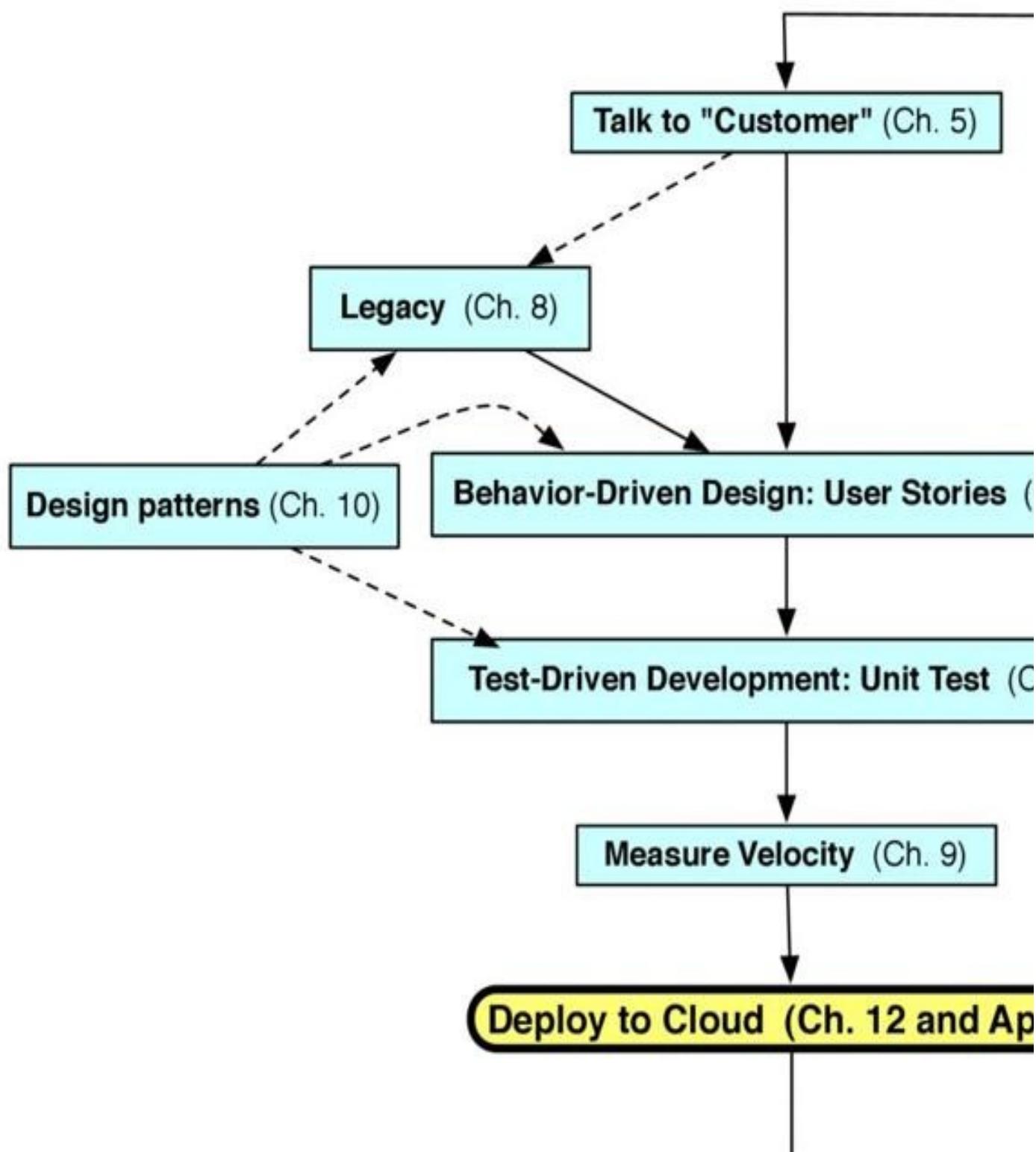


Figure 12.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter covers deploying the app into the cloud so that the customer can evaluate this Agile iteration.

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, which for small sites 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 2.4) 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. Other emerging PaaS’s are CloudFoundry, a PaaS software layer that can be deployed on either a company’s existing servers or a public cloud, and Microsoft Azure, a set of managed services based on Windows Server and running in Microsoft’s cloud.

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.

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.

- Availability and responsiveness: what percentage of the time is your app correctly serving requests, and how long do most users wait before the app delivers a useful response? (Section 12.2)

- Release management: how can you deploy or upgrade your app “in place” without reducing availability and responsiveness? (Sections [12.3](#) and [12.4](#))
- 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? Chapter [2](#) noted that 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 he or she claims to be, by verifying a password or using third-party authentication such as Facebook Connect 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 when tampering has occurred or 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 make up *security*, which we discuss in Section [12.8](#).

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.

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.

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 Chapter 1, **availability** refers to the fraction of time your site is available and working correctly. For example, Google Apps **guarantees** its enterprise customers a minimum of “three nines” or 99.9% availability, though Nygard wryly notes ([Nygaard 2007](#)) that less-disciplined sites provide closer to “two eights” (88.0%). **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 could sustain at most 56 Kbps (56×10^3 bits per second) while transferring the rest, so a Web page or image 50 KBytes in size or 400 Kbits could take more than eight seconds to deliver. However, today’s home customers increasingly use broadband connections whose throughput is 1–20 Mbps, so 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 this site is 200 ms, it is true that the *average* response time of $(8(100) + 1(250) + 1(850))/10 = 190\text{ms}$ 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:

- 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.

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.

- The [Apdex](#) score (Application Performance Index) is an [open standard](#) 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 $(8 + 0.5(1))/10 = 0.85$.

Of course, the total response time perceived by the users includes many factors beyond your SaaS app's control, including 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.

[Google](#) believes that this fact puts them under even more pressure to be responsive, so that getting a response from any Google service is no slower than contacting the service to begin with.

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 [2.4](#) suggested 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, companies like [RightScale](#) offer just this service on top of Amazon EC2. Overprovisioning also lets you deal gracefully with server crashes: temporarily losing one server degrades performance by $1/N$, so an easy solution is to overprovision by deploying $N+1$ servers. However, at large scale, systematic overprovisioning is infeasible: services using 1,000 computers cannot readily afford to keep 200 additional servers turned on just for overprovisioning.

Nonetheless, 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.
- 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.”
- 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.

Self-Check 12.2.1.

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

As we discussed in Section 1.3, 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. As you've seen, Heroku provides support for deployment automation, though automation tools such as [Capistrano](#)



help automate Rails deployments in non-PaaS environments. 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, two things change in deployment. First, behavioral or performance differences between the deployed and development versions of your app may arise from differences between the development and production environments or differences between users' browsers (especially for JavaScript-intensive apps). Second, deployment also requires testing 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 12.8)—and ensuring that it survives those conditions without compromising customer data or responsiveness.

A key technology in improving assurance for deployed code is [continuous integration](#) (CI), in which every change pushed to the code base triggers a set of integration tests to make sure nothing has broken.

With compiled languages such as Java, CI usually means building the app and then testing it.

The idea is similar to how we used autotest in Chapter 6, except that the complete integration test suite may include tests that a developer might not normally run on his own, such as:

- Browser compatibility: correct behavior across different browsers that have differences in CSS or JavaScript implementations
- Version compatibility: correct behavior on different versions of the Ruby interpreter (Ruby 1.9, JRuby, and so on), the Rack application server, or for software that may be hosted in a variety of environments, different versions of Ruby gems
- Service-oriented architecture integration: correct behavior when external services on which the app depends behave in unexpected ways (very slow connection, return a flood of garbage information, and so on)
- Stress: performance and stress tests such as those described in Section 12.5
- Hardening: testing for protection against malicious attacks such as those described in Section 12.8



CI systems are typically integrated into the overall development process rather than simply running tests passively. For example, Salesforce's CI system runs 150,000+ tests in parallel across many machines, and if any test fails, it performs binary searches across checkins to pinpoint the culprit and automatically opens a

bug report for the developer responsible for that checkin ([Hansma 2011](#)). [Travis](#), an open-source hosted CI system for Ruby apps, runs integration tests whenever it is notified of a new code push via GitHub's ***post-receive URI*** repo; it then uses OAuth (which we met in Section [7.2](#) to check out the code runs `rake test`, another demonstration of using `rake` tasks for automation. [SauceLabs](#) provides hosted CI focused on cross-browser testing: your app's Webdriver-based Cucumber scenarios are run against a variety of browsers and operating systems, with each test run captured as a screencast so you can visually inspect what the browser looked like for tests that failed.

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.

Summary of Continuous Integration (CI):

- CI consists of running a set of integration tests prior to deployment that are usually more extensive than what a single developer would run on his own.
- CI relies heavily on automation. Workflows can be constructed that automatically trigger CI when commits are pushed to a specific repo or branch.
- Continuous deployment (automatic deployment to production when all CI tests pass) may result in several 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.

ELABORATION: Staging

Many companies maintain an additional environment besides development and production called the ***staging site***. The staging site is usually identical to production except that it is usually smaller in scale, uses a separate database with test data (possibly extracted from real customer data), and is closed to outside users. The rationale is that stress testing and integration testing a staging version is the closest possible experience to the production site. Rails and its associated tools

support staging by defining an additional `staging`: environment in `config/environments/staging.rb` and `config/database.yml`.

Self-Check 12.3.1.

Given the prevalence of continuous deployment in Agile software companies, how would you characterize the difference between a deployment and a release?

 A release typically contains new features visible to the customer, whereas a deploy might contain new code that builds toward those features incrementally. As we know from Chapter 4, app changes sometimes require migrations to change the database schema. The 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.

<http://pastebin.com/5dj9k1cj>

```
1 class ChangeNameToFirstAndLast < ActiveRecord::Migration
2   def up
3     add_column 'moviegoers', 'first_name', :string
4     add_column 'moviegoers', 'last_name', :string
5     Moviegoer.all.each do |m|
6       m.update_attributes(:first => $1, :last => $2) if
7       m.name =~ /^(.*)\s+(.*)$/
8     end
9     remove_column 'moviegoers', 'name'
10   end
11 end
12
```

Figure 12.2: A migration that changes the schema and modifies the data to accommodate the change. In Section 12.4 we explain why there is no down-migration method. (Use Pastebin to copy-and-paste this code.)

<http://pastebin.com/TYx5qaSB>

```

1 class SplitName1 < ActiveRecord::Migration
2   def up
3     add_column 'moviegoers', 'first_name', :string
4     add_column 'moviegoers', 'last_name', :string
5     add_column 'moviegoers', 'migrated', :boolean
6     add_index 'moviegoers', 'migrated'
7   end
8 end
9
10
```

Figure 12.3: A partial migration that only adds columns but doesn't change or remove any. Section [12.7](#) explains why the index (line 6) is a good idea.

<http://pastebin.com/qqrLfuQh>

```

1 class Moviegoer < ActiveRecord::Base
2   # here's version n+1, using Setler gem for feature flag:
3   scope :old_schema, where :migrated => false
4   scope :new_schema, where :migrated => true
5   def self.find_matching_names(string)
6     if Featureflags.new_name_schema
7       Moviegoer.new_schema.where('last_name LIKE :s OR first
_name LIKE :s',
8         :s => "%#{string}%" ) +
9       Moviegoer.old_schema.where('name like ?', "%#{string
}%")
10    else # use only old schema
11      Moviegoer.where('name like ?', "%#{string}%")
12    end
13  end
14  # automatically update records to new schema when they are
 saved
15  before_save :update_schema, :unless => lambda { |
m| m.migrated? }
16  def update_schema
17    if name =~ /^(.*)\s+(.*)$/
```

```
18     self.first_name = $1
19     self.last_name = $2
20   end
21   self.migrated = true
22 end
23 end
24
```

Figure 12.4: Feature-flag wrapping for a model method that finds moviegoers by matching a string against their first or last names. Lines 15–22 install a before-save callback that will automatically update a record to the new schema whenever it's saved, so normal usage of the app will cause the records to be migrated piecemeal.

We could try to solve this problem by deploying the code and migration [atomically](#): take the service offline, apply the migration in Figure 12.2 to perform the schema change and copy the data into the new column, and bring the service back online. This approach is the simplest solution, but may cause unacceptable unavailability if the migration takes a long time to run because the database is large. 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 below is nondestructive: as we did with refactoring in Chapter 8, if something goes wrong at a given step the app is still left in a working intermediate state.

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, as in Figure 12.3.

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. Figure 12.4 shows an example.

Deploy version $n + 1$, which may require pushing the code to multiple servers, a process that can take several minutes.

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. In the example in Figure 12.4, 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).

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.

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.

When we introduced migrations in Chapter 4, we noted that a migration can include both an `up` and a `down` method, yet the example in Figure 12.3 has no `down` method. Shouldn't we include one in case the upgrade goes awry? Surprisingly, no. Down-migrations are useful during development, but risky in production. Because they are rarely used, they are usually less-than-thoroughly tested, and in the inevitable panic following the discovery that something has gone awry, it is difficult to be confident that they will really work without causing even more damage.

Even if you are confident that the down-migration works correctly, other developers may have pushed irreversible migrations after the one you're trying to down-migrate. And at some point you yourself will need to create an irreversible migration, and you will need a way to recover from problems when applying it. Feature flags can help: if something is going wrong, revert the value of the feature flag so that the code reverts to its previous behavior, then take your time debugging the problem.

Summary:

- 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. Start with a migration and code push that include both the old and new versions, and when this intermediate version is running, change the feature flag's value to enable the new code paths that use the new schema.
- 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 old schema. On the other hand, if anything goes wrong during the rollout, you can change the feature flag's value back in

order to continue using the old schema and code until you determine what went wrong.

- Deployed apps always move forward: if something goes wrong, fix it with another up-migration that undoes the damage rather than trying to apply a down-migration that hasn't been tested in production and may make things worse if applied.
-

ELABORATION: Other uses for feature flags

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](#) gem supports the use of feature flags for all these cases.

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.

If you're not monitoring it, it's probably broken. *variously attributed*

Given the importance of availability and responsiveness, 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.

SaaS apps can be monitored internally or externally. Internal or passive monitoring works by instrumenting your app, adding data collection code to the app itself, the environment in which it runs, or both. Before cloud computing and the prominence of SaaS and highly-productive frameworks, such monitoring required installing programs that collected metrics periodically, inserting instrumentation into the source code of 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 Relic](#) unobtrusively collects instrumentation about your app's controller actions, database queries, and so on. 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. The free level of New Relic RPM is available as a Heroku add-on or a standalone gem you can deploy in your own non-PaaS production environment.

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 SQLite3 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](#)).

A second kind of monitoring is external monitoring (sometimes called probing or active monitoring), in which 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 down altogether, especially 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." Figure [12.5](#) distinguishes different

types of monitoring and some tools to perform them, many delivered as SaaS.



What is monitored	Focus	Example tool
What is my site's availability and business-level average response time, as seen by users around the world?		Pingdom , SiteScope

What pages (views) in my app are most popular and what paths do customers follow through the app?	business-level	Google Analytics
What controller actions or database queries are slowest?	app-level	New Relic, Scout
What unexpected exceptions or errors did customers experience and what were they doing at the moment the error occurred?	app-level	Exceptional, AirBrake
What is the health and resource usage of the OS-level processes that support my app (Apache web server, MySQL DB server, and so on)?	infrastructure/process-level	god, monit

Figure 12.5: Different types of monitoring and example tools that provide them for Rails SaaS apps. All except the last row (process-level health monitoring) are delivered as SaaS and offer a free service tier that provides basic monitoring.

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 Laboratories](#), can simulate a specified number of users requesting simple sequences of URIs from an app and 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.

ELABORATION: Longevity bugs and rolling reboot

Stress testing can also expose bugs that only appear 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. Another example is running out of session storage when sessions are stored in a database table, which is why Rails’ default behavior is to serialize each user’s session object into a cookie stored at the user’s browser, although this limits each user’s session object to 4KiB in size.

Summary:

- As with testing, no single type of monitoring will alert you of all 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.
-

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. 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 Haml with embedded Ruby code and variables into HTML). In any caching scenario, we must address two issues:

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?

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?

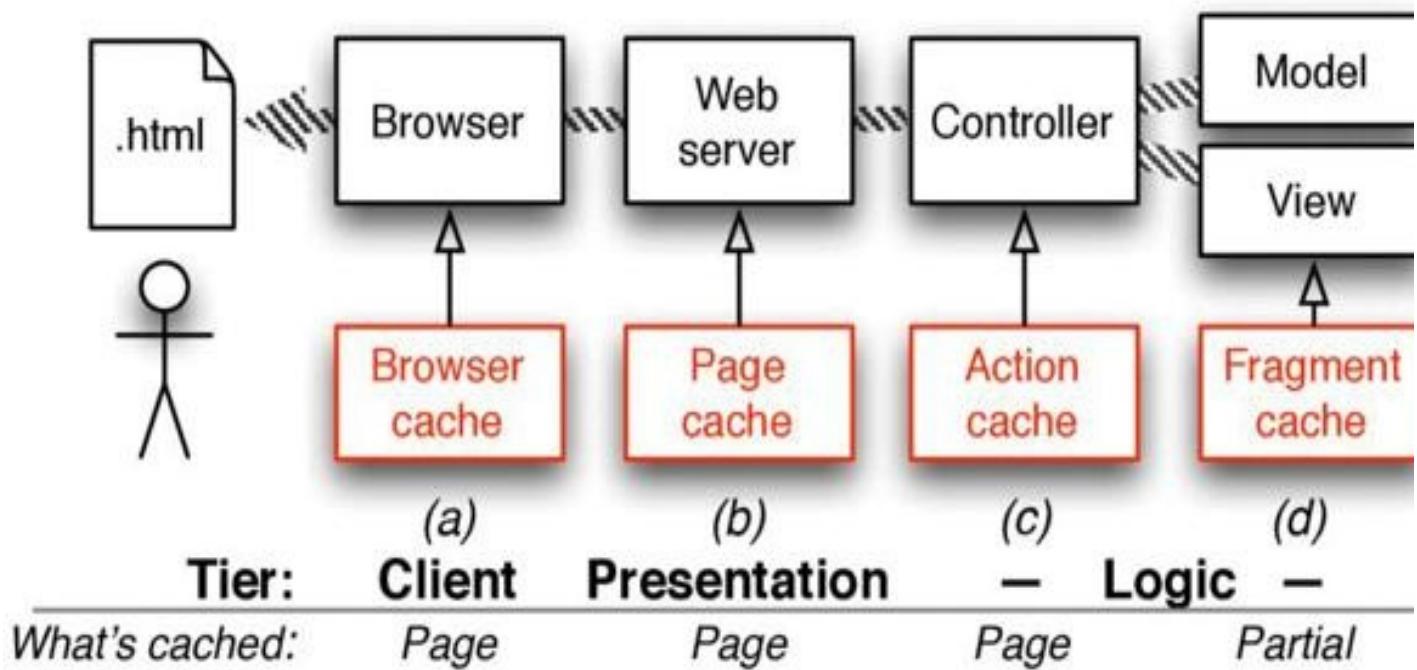


Figure 12.6: The goal of multiple levels of caching is to satisfy each HTTP request as close to the user as possible. (a) A Web browser that has previously visited a page can reuse the copy in its local cache after verifying with the server that the page hasn't changed. (b) Otherwise, the Web server 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`.

Figure 12.6 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 Haml template).

Figure 12.7 shows how to cache the entire HTML page for a movie, so that future requests to that page neither access the database nor re-run the Haml renderer, as in Figure 12.6(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 12.6(c). Figure 12.9 shows the benefits of page and action caching for this simple example. Note that in Rails page caching, the name of the cached object *ignores* embedded parameters in URLs 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.haml 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.

<http://pastebin.com/iAa8mvpL>

```
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
```

<http://pastebin.com/kBknjCF9>

```
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
```

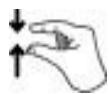
Figure 12.7: (Top) Line 2 specifies that Rails should cache the result of the `show` action. Action caching is implemented as a before-filter that checks whether a cached version should be used and an around-filter that captures and caches the rendered output, making it an example of the Decorator design pattern (Section 10.4). (Bottom) This “sweeper,” referenced by line 3 of the controller, uses the Observer design pattern (Section 10.7) to add ActiveRecord lifecycle hooks (Section 7.1) to expire any objects that might become stale as a result of updating a particular movie.

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, as Figure 7.2 in Section 7.1 shows. Figure 12.8 shows how adding a single line to the partial of Figure 7.2 caches the rendered HTML fragment corresponding to each movie.

<http://pastebin.com/Sf9abAXh>

```
1 -# A single row of the All Movies table
2   - cache(movie) do
3     %tr
4       %td= movie.title
5       %td= movie.rating
6       %td= movie.release_date
7       %td= link_to "More about #{movie.title}", movie_path(m
ovie)
```

Figure 12.8: Compared to Figure 7.2 in Section 7.1, only line 2 has been added. 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 12.7 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:

<http://pastebin.com/m2fqxPfX>

```
1  -# a cacheable partial for top movies
2  - cache('top_moviegoers') do
3      %ul#topmovies
4          - @top_5.each do |movie|
5              %li= moviegoer.name
```

<http://pastebin.com/TgTWttZa>

```
1  class MoviegoersController < 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 5 of the Haml template, 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 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.

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.



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 noncacheable entity, such as a single model instance, into its own partial that can be fragment-cached.

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 12.9: 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`.

Summary of caching:

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 filesystem. On Heroku, the Memcached addon stores 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 [10.6](#).

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.

As we saw in Section 2.4, 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 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 Tracker](#) 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:

The *n+1 queries* problem occurs when traversing an association performs more queries than necessary.

The *table scan* problem occurs when your tables lack the proper [*indices*](#) to speed up certain queries.

<http://pastebin.com/McgXtBUb>

```
1 # assumes class Moviegoer with has_many :movies
2
3 # in controller method:
4 @fan = Moviegoer.find_by_email(email) # causes table scan if
no index
5
6 # in view:
7 - @fan.favorite_movies.each do |movie|
```

```

8 // BAD: each time thru this loop causes a new database que
ry!
9 %p= movie.title
10
11 # better: eager loading of the association in controller.
12 # Rails automatically traverses the through-
association between
13 # Moviegoers and Movies through Reviews
14 @fan = Moviegoer.includes(:movies).find_by_email(email)
15 # now we have preloaded all the movies this moviegoer review
ed.
16
17 # in view:
18 - @fan.movies.each do |movie|
19   // GOOD: this code no longer causes additional queries
20   %p= movie.title
21
22 # BAD: preload association but don't use it in view:
23 %p= @fan.name
24 // BAD: we never used the :favorite_movies that were preload
ed!

```

Figure 12.10: The query in the controller action (line 4) accesses the database once to retrieve the `@fan` model, but each pass through the loop in lines 7–9 causes another separate database access, resulting in $n + 1$ accesses for a moviegoer with n favorite movies. Line 12, 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.

Lines 1–18 of Figure 12.10 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 20–22 of Figure 12.10. The [bullet](#) gem helps detect both

problems.



<http://pastebin.com/bcAq3Zid>

```

1 class AddEmailIndexToMoviegoers < ActiveRecord::Migration
2   def up
3     add_index 'moviegoers', 'email', :unique => true
4     # :unique is optional - see text for important warning!
5   end
6 end

```

Figure 12.11: 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 [7.1](#).

Another database abuse to avoid is queries that result in a [**full table scan**](#). Consider line 4 of Figure [12.10](#): in the worst case, the database would have to examine every row of the moviegoers table to find a match on the email 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.email` column, as Figure [12.11](#) 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` or `find_by_x` queries, [**foreign keys**](#) (the subject of the association) should usually be indexed. For example, in the example in Figure [12.10](#), the `moviegoer_id` field in the movies table would need an index in order to speed up the query implied by `movie.moviegoers`.

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 [12.12](#) shows an example of the dramatic performance improvement provided by indices.

# of reviews:	2000	20,000	200,000
Read 100, no indices	0.94	1.33	5.28

Read 100,	0.57	0.63	0.65
FK indices			
Performance	166%	212%	808%
Create 1000, no indices		9.69	
Create 1000, all indices		11.30	
Performance		-17%	

Figure 12.12: For a PostgreSQL shared database on Heroku containing 1K movies, 1K moviegoers, and 2K to 20K 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.

Summary of avoiding abusive queries:

- 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 run](#) 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](#) 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

My response was “Congratulations, Ron, that should work.” *Len Adleman, reacting to Ron Rivest’s encryption proposal, 1977*



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 $C = m^e \bmod N$.

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.

ssh-keygen is a standard keypair-generation utility included with the bookware.

A keypair has two important properties:

A message encrypted using the private key can only be decrypted using the public key, and vice-versa.

The private key cannot be deduced from the public key, and vice-versa.

Property 1 provides the foundation of SSL: 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."

Alice and Bob are the [**archetypal principals**](#) who appear in security scenarios, along with eavesdropper Eve, malicious Mallory, and other colorful characters.

To offer SSL-secured 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 an [SSL certificate](#) that states, in effect, “`rottenpotatoes.com` has public key KU .” Bob installs the certificate on his server and enables his SaaS stack to accept SSL connections—usually trivial in a PaaS environment. Finally, he enables SSL in his Rails app by adding `force_ssl` to any controller to force all its actions to use SSL, or using the `:only` or `:except` filter options to limit which actions are affected.

`force_ssl` is implemented as a before-filter that causes an immediate redirect from `http://site/action` to `https://site/action`.

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 SSL 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”:

<http://pastebin.com/p5ShmHkA>

```
1  <p>Here's a risque picture of me:  
2      
3  </p>
```

When Alice views the blog post, or if she receives an email with this 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.haml` and `app/controllers/application_controller.rb` respectively.

<http://pastebin.com/PbiR8v8E>

```
1 class MoviesController
2   def search
3     movies = Movie.where("name = '#{params[:title]}'") # UNS
AFE!
4     # movies = Movie.where("name = ?", params[:title])    # s
afe
5   end
6 end
```

<code>params[:title]</code>	SQL statement
Aladdin	<code>SELECT "movies".* FROM "movies"</code> <code>WHERE (title='Aladdin')</code>
<code>');</code>	<code>DROP TABLE "movies"; --</code>
	<code>SELECT "movies".* FROM "movies"</code> <code>WHERE (title='');</code>
	<code>DROP TABLE</code>
	<code>"movies"; --</code>

Figure 12.13: If Mallory enters the text in the second row of the table as a movie title, line 3 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`.) Line 4 thwarts the attack using a ***prepared statement***, which lets ActiveRecord “sanitize” Mallory’s input before inserting it in the query. SQL injection was often successful against early frameworks such as PHP, in which queries were hand-coded by programmers.

SQL injection and cross-site scripting. Both of these attacks exploit SaaS apps that handle attacker-provided content unsafely. In [SQL injection](#), Mallory enters form data that she hopes will be interpolated directly into a SQL query statement executed by the app. Figure 12.13 shows an example and its defense—using prepared statements. In [cross-site scripting](#), 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 12.14 shows a benign example and the defense; 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.

Security firm Symantec [reported](#) that XSS accounted for over 80% of security vulnerabilities in 2007.

JavaScript is the language of choice for XSS, but any technology that mixes code into HTML pages is vulnerable, including ActiveX, VBScript, and Flash.

<http://pastebin.com/jXdN0jmS>

```
1 <h2><%= movie.title %></h2>
2 <p>Released on <%= movie.release_date %>. Rated <%= movie.ra
ting %>.</p>
```

<http://pastebin.com/trzi3XpK>

```
1 <h2><script>alert("Danger!");</script></h2>
2 <p>Released on 1992-11-25 00:00:00 UTC. Rated G.</p>
```

<http://pastebin.com/FhTTm7DF>

```
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 12.14: Top: a fragment of a view template using Rails' built-in eRB renderer rather than Haml. Middle: Mallory manually enters a new movie whose "title" is the string <script>alert("Danger!");</script>. When this movie's Show action is rendered, the "title" will be inserted directly into the HTML view, causing the JavaScript code to be executed when Alice's browser renders the page. Bottom: The defense is to "sanitize" any input that will be interpolated into HTML. Happily, Haml's = operator does this automatically, resulting in impotent code where the angle brackets have been properly escaped for HTML.

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. (Uploading files also carries other risks, so you should "outsource" that responsibility to other services, such as [the Progstr-Filer Heroku addon](#).) A defense is to use a separate background task such as a [Heroku worker](#) to offload long-running jobs from the main app server.

When Amazon put 1000 Xbox 360 units on sale for just \$100 rather than the list price of \$399, during the first 5 minutes the site was brought to its knees by millions of users clicking "Reload" to get the deal.

Attack

Eavesdropping

Man-in-the-middle attacks (replay, session hijacking)

Cross-site request forgery (CSRF)

Cross-site scripting (XSS)

SQL injection

Mass assignment of sensitive

Rails Defenses

Install SSL certificate and use `force_ssl` in controllers (optional: `:only=>` or `:except=>` specific actions) to encrypt traffic using SSL

Include a [`nonce`](#) in the session (TBD find place where you set session secret in rails app)

Render `csrf_meta_tags` in all views (for example, by including it in main layout) and specify `protect_from_forgery` in `ApplicationController`

Use Haml's = to sanitize HTML during rendering

Use prepared queries with placeholders, rather than interpolating strings directly into queries

Use `attr_protected` or `attr_accessible`

attributes	to protect sensitive attributes from user assignment (Section 7.2.1)
Executing protected actions	Use <code>before_filter</code> to guard sensitive public methods in controllers; declare nonpublic controller methods as <code>private</code> or <code>protected</code>
Self-denial-of-service, pathologically slow clients	Use separate background workers to perform long-running tasks, rather than tying up the app server

Figure 12.15: Some common attacks against SaaS apps and the Rails mechanisms that defend against them.

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 [7.2](#), to avoid [embarrassing incidents of password theft](#). 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 [credit card fraud](#)), 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 [PayPal](#) or [Stripe](#) that specialize in meeting these heavy burdens.

Summary of defending customer data:

- SSL and TLS keep data private as it travels over an HTTP connection, but provide no other privacy guarantees. They also assure the browser of the server’s identity (unless the Certificate Authority that originally certified the server’s identity has been compromised), but not vice-versa.
- Developers who deploy on a well-curated PaaS should focus primarily on attacks that can be thwarted by good coding practices. Figure [12.15](#) 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.

Self-Check 12.8.1. True or false: If a site has a valid SSL 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 12.8.2. Why can't CSRF attacks be thwarted by checking the Referer : header of an HTTP request?

 The header can be trivially forged.

 **Fallacy: All the extra effort for testing very rare conditions in Continuous Integration tests is more trouble than it's worth.**

1 million hits per day was Slashdot's volume in 2010, at a time when [Facebook was handling](#) 8 billion (8×10^{12}) hits a day. At 1M hits per day, a one-in-a-million event is statistically likely every day. At 8B hits per day, 8,000 such "unlikely" 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 provided by error-handling code will help you sleep better at night.

 **Pitfall: Missing the forest for the trees by abusing continuous deployment.**

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 8) 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: This is a typical 3-tier app using cloud computing, so it will scale.**

This isn't really a fallacy, because if you're using well-curated PaaS, 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.

 **Fallacy: It's still in development, so we can ignore performance.**

It's true that Knuth 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 is just as bad as focusing myopically on performance at an early stage. Avoid truly egregious performance mistakes and you will be able to steer a happy path between two extremes.



Fallacy: CPU cycles are free since computers have become so fast and cheap.

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 12.16 clearly shows that when extra CPU time becomes extra latency (and therefore reduced responsiveness) for the end user, CPU cycles aren't free at all.

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 12.16: The measured effects of added latency on users' interaction with various large SaaS apps, from [Yahoo performance engineer Nicole Sullivan's "Design Fast Websites" presentation](#) and a [joint presentation at the Velocity 2009 conference](#) by Jake Brutlag of Google and Eric Schurman of Amazon.

⚠ 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 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: Bugs in naming or expiration logic cause 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 Guide](#) to turn on caching in the testing and development environments, where it's off by default to simplify debugging.



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 becomes suddenly popular because of Slashdot or Digg, you'll be suddenly inundated with traffic. The lesson is: *If your app is publicly deployed, it is a target.*



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.

Performance and security are systemwide concerns that must be constantly reviewed, rather than problems to be solved once and then set aside. In addition, the database abuses described in Section [12.7](#) reveal that Rails and 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 you as a developer to have some understanding of how the abstractions work. For example, the n + 1 select problem is not obvious from looking at Rails code, nor is the solution of providing hints like :[include](#).

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. Indeed, in this book we started out developing our apps using SQLite3 database but deploying on Heroku with PostgreSQL. Given these differences, will performance improvements made during development (reducing the number of queries, adding indices, adding caching) still apply in production? 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 inefficient query strategies such as the n + 1 query problem or for not deploying 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.

Our monitoring examples are based on aggregating metrics such as latency over many requests. A contrasting approach is request tracing, which is used in conjunction with metric aggregation to pinpoint and diagnose slow requests. You can see a simplified version of request tracing in the development-mode Rails logs, which by default will report the database query time, controller action time, and rendering time for each request. True request tracing is much finer-grained, following request through every software component in every tier and timestamping it along the way. Companies that run large sites often build their own request tracing, such as Twitter's Big Brother Bird and Google's Dapper. In a recent article ([Barroso and Dean 2012](#)) two Google architects discuss how request tracing identifies obstacles to keeping Google's massively-parallel systems highly responsive.

Although we focused on monitoring for performance problems, but 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.

- 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 environment based on the original Unix model of processes and I/O. *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.
- The [RailsLab blog](#) maintained by New Relic collects best practices and techniques for tuning Rails apps, including screencasts on Rails tuning and how to [use New Relic in development mode for profiling](#). Be aware, though, that some of the specific code examples, especially around caching, are no longer valid because of changes between Rails 2 and Rails 3.
- Sharding and replication are powerful techniques for scaling a database that require a great deal of design thinking up front. The techniques 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.

- Google’s “[Speed is a Feature](#)” 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.
- James Hamilton, an engineer and architect at Amazon Web Services who previously spent many years architecting and tuning Microsoft SQL Server, writes an excellent [blog](#) that includes a great article on the [cost of latency](#), gathering measured results from a variety of industry practitioners who have investigated the topic.

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 [man-in-the-middle attack](#) that falsifies the certificate attesting to the server’s identity. The impostor site looks and behaves like the real site but collects sensitive information from users. (In September 2011, [hackers impersonated the CIA, MI6, and Mossad sites](#) by compromising DigiNotar, the company that signed the original certificates for those sites, leading to DigiNotar’s bankruptcy.) Even mature software such as Secure Shell and Apache are vulnerable: the US [National Vulnerabilities Database](#) lists 10 new security-related bugs in Apache just between March and May 2012, two of which are “high severity”, meaning that they could allow an attacker to take control of your entire server. 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. The [Basic Rails security guide](#) at the Ruby on Rails site reviews many Rails features aimed at thwarting common attacks against Saas apps.

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 postmortem 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.

In *The Evolution of Useful Things* ([Petroski 1994](#)), engineer Henry Petroski proposes changing the maxim “Form follows function” (originally from the world of architecture) to “Form follows failure” after demonstrating that the design of many successful products was influenced primarily by failures in early designs that led to revised designs. For an example of good design, read Netflix’s [technical blog post](#) on how their design survived the Amazon Web Services outage in 2011 that crippled many other sites reliant on AWS.

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.

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.

B. W. Kernighan and R. Pike. *Unix Programming Environment (Prentice-Hall Software Series)*. Prentice Hall Ptr, 1984. ISBN 013937681X. URL

<http://www.amazon.com/Programming-Environment-Prentice-Hall-Software-Series/dp/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. URL <http://www.amazon.com/Release-Production-Ready-Software-Pragmatic-Programmers/dp/0978739213>.

H. Petroski. *The Evolution of Useful Things: How Everyday Artifacts-From Forks and Pins to Paper Clips and Zippers-Came to be as They are*. Vintage, 1994. ISBN 0679740392. URL <http://www.amazon.com/Evolution-Useful-Things-Artifacts-Zippers-Came/dp/0679740392>.

P. van Hardenberg. Personal communication, April 2012.

For many of these exercises, you will find it useful to create a rake task that creates a specified large number of randomly-generated instances of a given model type.

You will also want to deploy a 'staging' copy of your app on Heroku so that you can use the staging database for experiments without modifying the production database.

Project 12.1.

Seed the staging database with 500 movies (randomly generated is fine) to RottenPotatoes. Profile its deployed performance on Heroku using New Relic. Add fragment caching around the `movie` partial used by the `index` view and re-profile the app. How much improvement in responsiveness do you observe for the `index` view once the cache is warmed up?

Project 12.2.

Continuing the previous exercise, use `httperf` to compare the throughput of a single copy of your app on the `index` action with and without fragment caching. (On Heroku, by default any apps on a free account receive 1 "dyno" or one unit of task parallelism, so requests are performed sequentially.)

Project 12.3.

Continuing the previous exercise, add action caching for the `index` view so that if no sorting or filtering options are specified, the `index` action just returns all movies. Compare the latency and throughput with and without action caching. Summarize the results from all three exercises in a table.

Upgrades and feature flags:

Project 12.4.

Investigate the availability cost of doing an "atomic" schema update and migration as described in Section [12.4](#). To do this, repeat the following sequence of steps for $N = 2^{10}, 2^{12}, 2^{14}$:

Seed the staging database with N randomly-generated movies with random ratings.

Run a migration on the staging database that changes the `rating` column in the `movies` table from a string to an integer (1=G, 2=PG, and so on).

Note the time reported by `rake db:migrate`.

Suppose your uptime target was 99.9% over any 30-day window. Quantify the effect on availability of doing the above migration without bringing your service down.

Project 12.5.

In Section [7.2](#) we integrated third-party authentication into RottenPotatoes by adding an authentication provider name and a provider-specific UID to the `Moviegoer` model.

Now we'd like to go further and allow the same moviegoer to log in to the same account with any one of several authentication providers. That is, if Alice has both a Twitter account and a Facebook account, she should be able to log in to the same RottenPotatoes account with either ID.

Describe the changes to existing models and tables that are necessary to support this scheme.

Describe a sequence of deployments and migrations that make use of feature flags to implement the new scheme without any application downtime.

Security:

Project 12.6. [Wired Magazine's ThreatLevel column for July 2012](#) reported that 453,000 passwords for Yahoo! Voice users were stolen by hackers. The hackers said, in a note posted online, that the passwords were stored in cleartext on Yahoo's servers and that they used a SQL injection attack to gather them. Discuss.

Project 12.7. TBD: exercise about compromising the browser binary

Project 12.8. TBD: exercise about why it's hard for a new CA to go into business



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 [Perspectives on SaaS and SOA](#)

13.2 [Looking Backwards](#)

13.3 [Looking Forwards](#)

13.4 [Evaluating the Book in the Classroom](#)

13.5 [Last Words](#)

13.6 [To Learn More](#)

A.1 [Alpha Edition Guidance](#)

A.2 [Overview of the Bookware](#)

A.3 [Using the Bookware VM With VirtualBox](#)

A.4 [Using the Bookware VM on Amazon Elastic Compute Cloud](#)

A.5 [Working With Code: Editors and Unix Survival Skills](#)

A.6 [Getting Started With Git for Version Control](#)

A.7 [Getting Started With GitHub or ProjectLocker](#)

A.8 [Deploying to the Cloud Using Heroku](#)

A.9 [Fallacies and Pitfalls](#)

A.10 [To Learn More](#)

In this chapter we give perspectives on the big ideas in this book—Agile, Cloud Computing, Rails, SaaS, and SOA—and show how Berkeley students who have graduated and taken jobs in industry rank their importance.

In this book you’ve been mainly a user of a successful distributed architecture (the Web, SOA) and framework (Rails). As a successful software engineer you’ll likely need to create such frameworks, or extend existing ones. Paying careful attention to principles that made these frameworks successful will help.

In Chapter [2](#) we pointed out that by choosing to build SaaS, some architectural choices are made for you. In building different kinds of systems, other choices might be appropriate, but for reasons of scope, we have focused on this one set of choices. But it's worth pointing out that some of the important architectural principles underlying SaaS and SOA apply in other architectures as well, and as Jim Gray's quote at the front of Chapter [3](#) suggests, great ideas take time to mature.

For example, 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 URLs in the early 1990s to REST by the early 2000s;
- From session "hacks" (fat URLs, hidden fields, and so on) in the early 1990s to cookies and real sessions by the mid 1990s; and
- 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 to 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.

Rails itself took off with the shift in the software industry towards Software as a Service (SaaS) using Agile development and deployed via cloud computing. Today virtually every traditional buy-and-install program is offered as a service, including PC standard-bearers like Office (see Office 365) and TurboTax (see TurboTax Online). Tools like Rails made Agile much easier to use and practice than earlier software development methods. Remarkably, not only has the future of software been revolutionized, software development is now easier to learn!

13.2 Looking Backwards

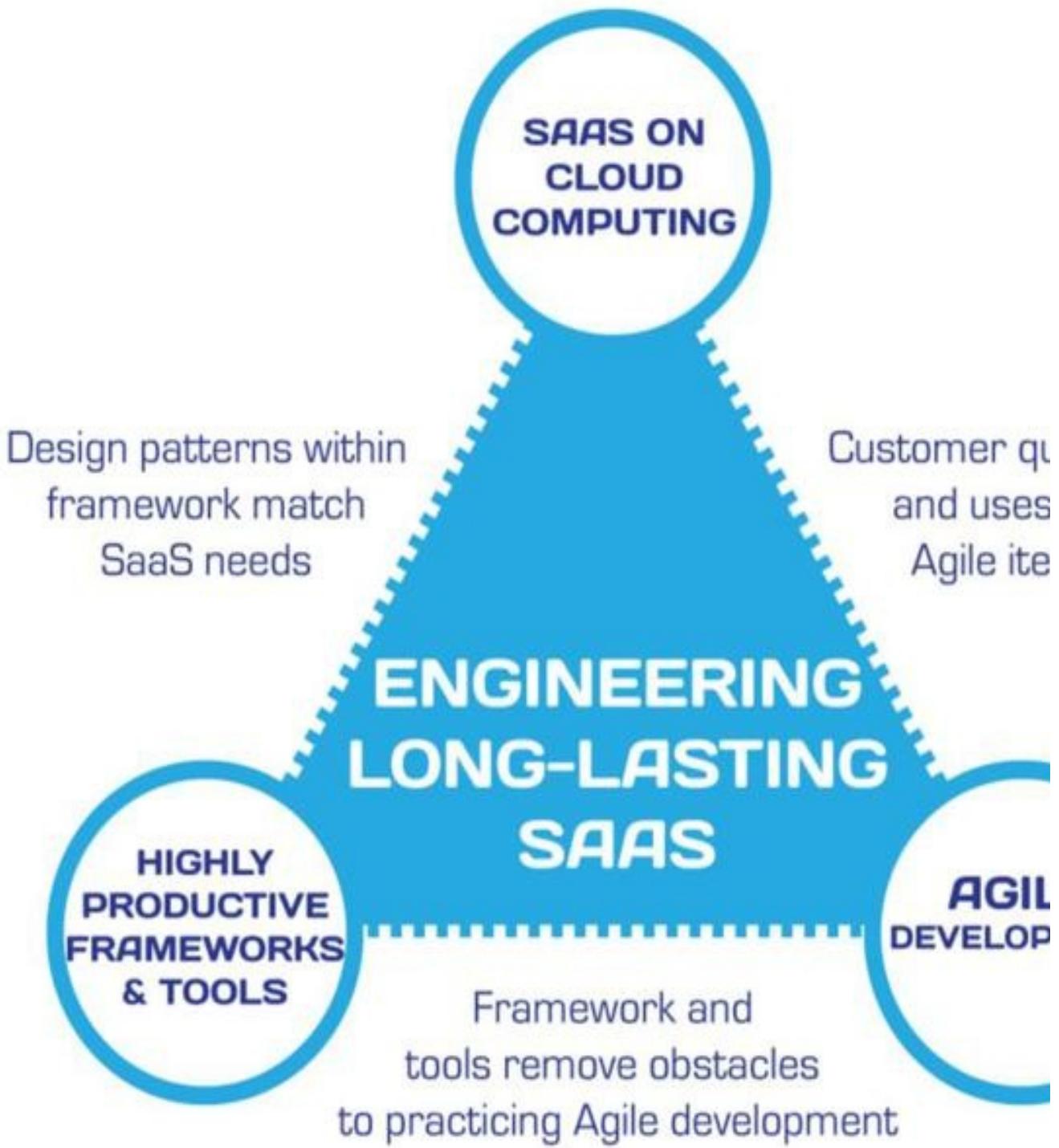


Figure 13.1: The Virtuous Triangle of Engineering Long-Lived SaaS is formed from the three software engineering crown jewels of (1) SaaS on Cloud Computing, (2) Highly Productive Framework and Tools, and (3) Agile Development.

Figure 13.1, first seen in Chapter 1, shows the three “crown jewels” on which the material in this book is based. To understand this virtuous triangle you needed to learn many new terms; Figure 13.2 lists nearly 120 terms from just the first three chapters!

Chapter 1	Chapter 2 (cont'd)	Chapter 3
acceptance test	HTTP cookie	accessor method
agile development process	HTTP method	anonymous lambda
automatic theorem proving	HTTP server	app root
cloud computing	interpolated	backtrace
cluster	IP address	block
DRY(Don't Repeat Yourself)	load balancer	class variable
formal methods	logic tier	closure
functional test	markup language	duck typing
integration test	master-slave	dynamically typed
legacy code	middleware	encapsulation
lifecycle	model	functional program
model checking	multi-homed	gem
module test	MVC (Model-View-Controller)	generator
object oriented programming	network interface	getter method
public cloud service	network protocol	idempotent
regression test	peer-to-peer architecture	instance variable
SaaS (Software as a Service)	persistence tier	instrumentation
SOA (Service Oriented Arch.)	primary key	iterator
system test	public-key cryptography	lexical scoping
test coverage	push-based	looking up a method
unit test	RDBMS (relational database management system)	metaprogramming
utility computing	relational algebra	method chaining
validation	relational database	migration
verification	request-reply protocol	mix-ins
virtual machine	route	mutator method
warehouse scale computer	selector notation	poetry mode
waterfall development process	session	receiver
Chapter 2	SGML (Standard Generalized Markup Language)	reflection
action	sharding	regex
application server	shared-nothing architecture	regular expression
client-server architecture	stateless protocol	root class
controller	structured storage	setter method
CSS (Cascading Style Sheet)	TCP port number	static variable
CRUD (Create, Read, Update, Delete)	TCP/IP (Transmission Control Protocol/Internet Protocol)	symbol
data consistency	URI (Uniform Resource Identifier)	syntactic sugar
design pattern	view	type casting
DNS (Domain Name System)	web application framework	yield
HAML (HTML Abstraction Markup Language)	Web server	
	XHTML (eXtended HyperText	

hostnames	Markup Language)
Language)	e Markup Language)
HTTP (HyperText Transfer Protocol)	



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 13.3 shows our oft-repeated Agile iteration, but this time it is decorated with the tools and services that we use in this book. Without these 14 tools and services, it would be a lot harder to follow the Agile lifecycle *and* to develop SaaS apps. Similarly, Figure 13.4 summarizes the relationship between phases in the waterfall lifecycle and their agile equivalents, showing how the techniques described in this book play the same roles as original techniques in earlier software process models.

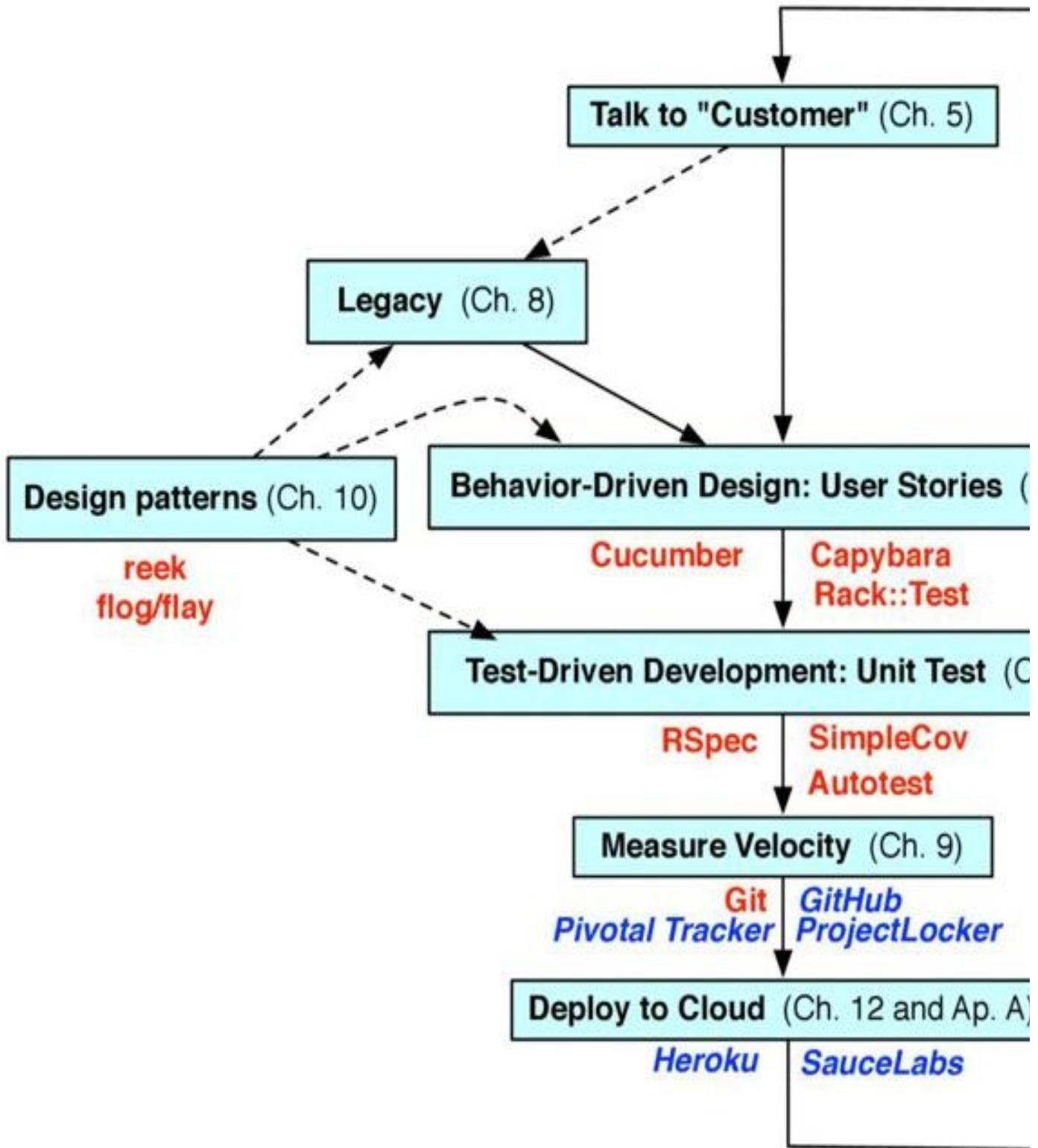


Figure 13.3: An iteration of the Agile software lifecycle and its relationship to the chapters in this book, with the supporting tools (red bold letters) and supporting services (blue italic letters) identified with each step.

Waterfall/Spiral

Requirements gathering and analysis

Agile

BDD with short iterations so

Chapter

5

Periodic code reviews	customer participates in design	
Periodic design reviews	Pair programming (pairs constantly reviewing each others' code)	9
Test entire design after building	Pull requests drive discussions about design changes	9
Post-implementation Integration Testing	TDD to test continuously as you design	6
Infrequent major releases	Continuous integration testing	12
	Continuous deployment	12

Figure 13.4: 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.

I've always been more interested in the future than in the past. *Grace Murray Hopper*

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 to 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 re-running the program using a binary search strategy and automated testing, it methodically narrows the differences between the two runs. Delta debugging discovers dependencies that form a cause-effect chain, which it expands until it identifies a smallest set of changes to input variables that cause 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.

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. While we are currently in the early days of this movement, it does appear to us to be an important change to higher education. Short lecture segments, free video distribution over the Internet, rigorous automated testing, and online forums combine to form a wonderful, low-cost vehicle for motivated students around the world. MOOCs will also likely have the side effect of raising the bar for traditional courses, as students will now have viable alternatives to ineffective live lectures.

The tools and services in Figure [13.3](#) and their resulting productivity allow college

students to experience the whole software life cycle repeatedly in a single course. A typical undergraduate workload of four courses per term and a 50-hour work week gives students about 12 hours per week per course, including lectures, labs, exams, and so forth. This works out to roughly 120 hours per quarter to 180 hours per semester, or just three to four weeks for a full-time developer! The productivity of Rails and its ecosystem of tools and the automation of tasks via services allow students to learn and practice software engineering fundamentals within the time

budget of the classroom.

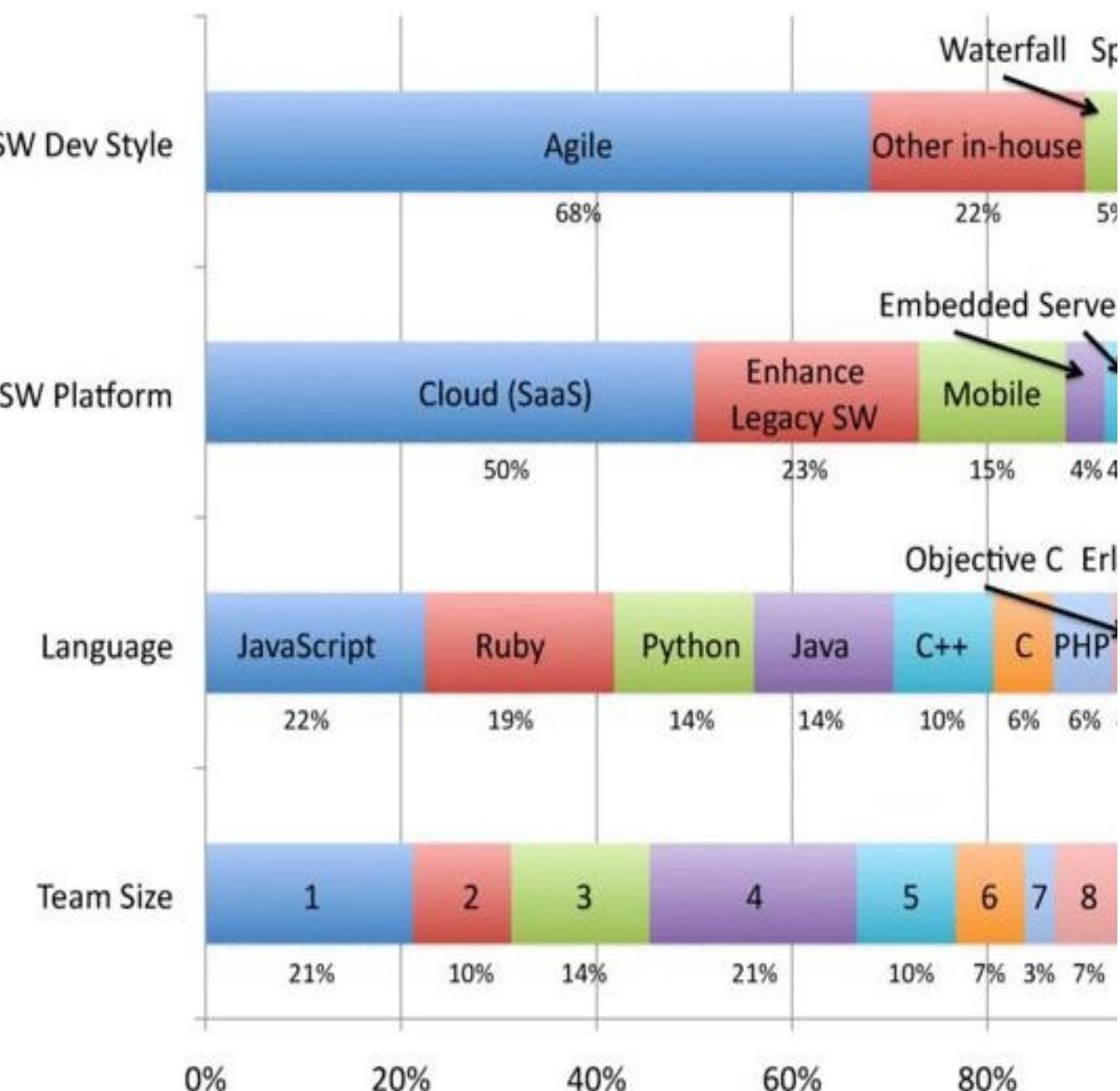


Figure 13.5: A survey of former Berkeley students now in industry concerning software development

styles, software platforms, programming languages used, and size of programming teams. The Waterfall software development process is characterized by much of the design being done in advance in coding, completing each phase before going on to the next one.

The ideas in this book are useful as well as enlightening. Figure 13.5 shows the survey results of Berkeley students from two course offerings of a class based on the material in this book. Just 22 of the 47 respondents had graduated, and just 19 had done significant software projects. The figures below show the results of their 26 software projects. We were surprised that Agile software development was so popular (68%) and that the cloud was such a popular platform (50%). Given that no language was used in more than 22% of the projects, our alumni must be using Agile in projects that use languages other than Ruby or Python. All the class teams had four or five students, which happily matches the average team size from the survey. Agile development and Rails were not selected because we expected them to dominate students' professional careers upon graduation, but because their productivity allows us to fit several critical ideas into a single college course in the hope that students will later apply them to other methodologies, languages, and frameworks.

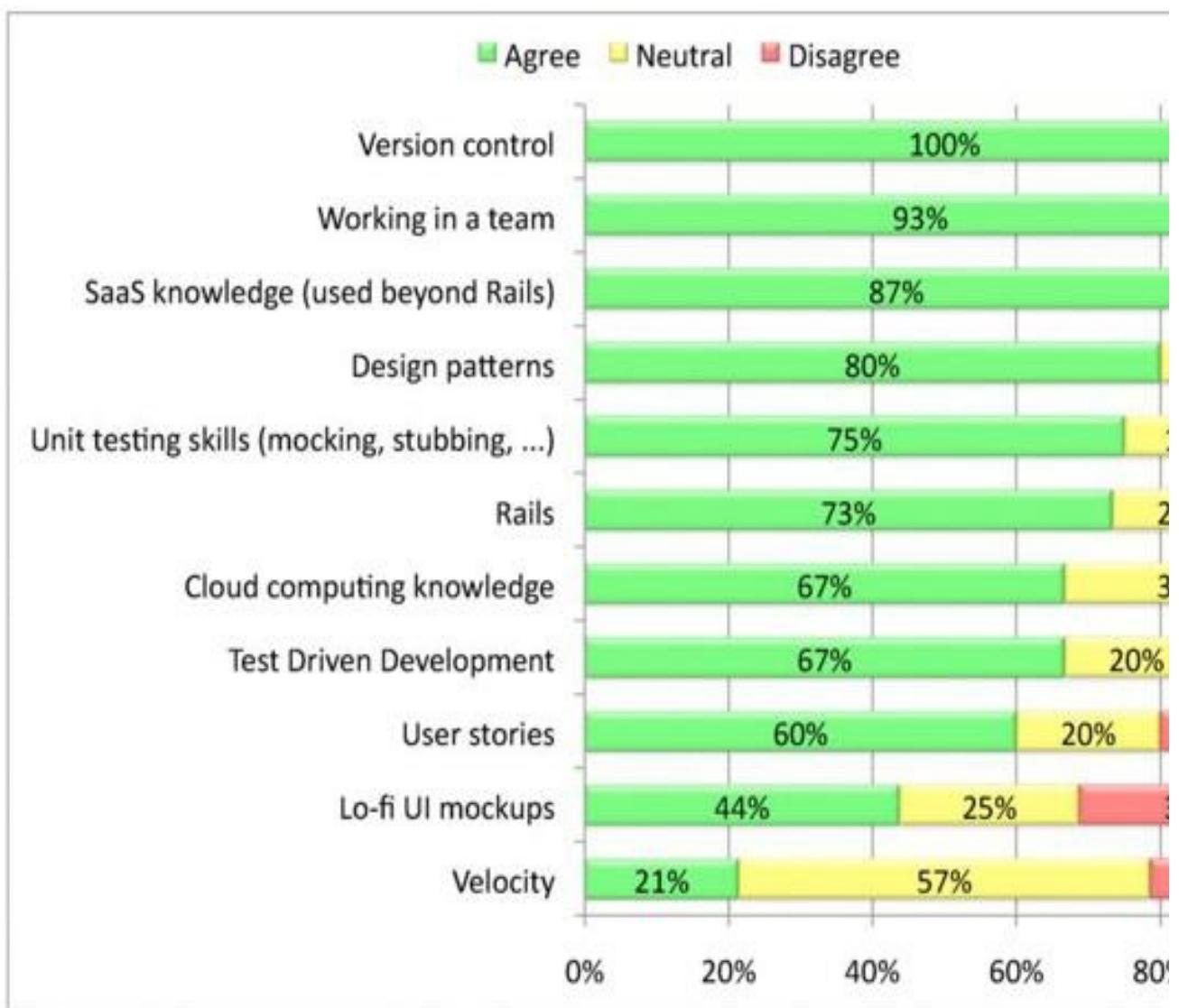


Figure 13.6: A survey of former Berkeley students asking if the topics in this book were useful in their industrial projects.

Figure 13.6 shows the students' ranking of the topics in the book in terms of usefulness in their industrial projects. Most students agreed that the concepts in the course were useful in their jobs. Once again, we were pleased to see that these ideas were still being used, even in industrial projects that did not rely on Agile or on Rails. The two lower ranked topics were Lo-Fi User Interface Mockups, which makes sense since few developers work on the UI of a user-facing project, and Velocity, as progress can be measured in other ways in industry.

Although a small sample and not a conclusive user study, we think our survey offers at least anecdotal evidence that people who study this material continue to use successful software development techniques in later software projects of all kinds. We believe the approach of this book pleases three groups of stakeholders ([Fox and Patterson 2012](#)):

Students like it because they get the pride of accomplishment in shipping code that works and is used by people other than their instructors, plus they get experience that can help land internships or jobs.

Faculty like it because students actually use what they hear in lecture, they can see how big computer science ideas genuinely improve productivity, and they use what they learn beyond the classroom. They also enjoy that the testing and code evaluation tools of Rails can help automatically grade interesting programming assignments. Autograding enables teaching the Agile lifecycle in a [**Massive Open Online Course**](#), thereby allowing thousands of students to learn the material at once.

Colleagues in industry like it because it addresses several of their concerns. An example is this quote from a Google engineer:

“I’d be far more likely to prefer graduates of this program than any other I’ve seen. As you know, we’re not a Ruby shop, but I feel this is a good choice for the class to be able to get real features done. Java or C++ would take forever.”

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 rocket](#) and [Mars Observer](#) as well as the deaths of several patients due to radiation overdoses from the [Therac-25 machine](#).

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 9, 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!

- 13.5 To Learn More** Crossing the software education chasm. *Communications of the ACM*, pages 44–49, May 2012.
- F. Ji and T. Sedano. Comparing extreme programming and waterfall project results. *Conference on Software Engineering Education and Training*, pages 482–486, 2011.
- 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>.



Frances Allen (1932–) received the 2006 Turing Award for pioneering contributions to the theory and practice of optimizing compiler techniques that laid the foundation for modern optimizing compilers and automatic parallel execution.

All the things I do are of a piece. I'm exploring the edges, finding new ways of doing things. It keeps me very, very engaged.

Fran Allen, from Computer History Museum Fellow Award Plaque, 2000

Although we take steps in this book to minimize the pain, such as using Test-Driven Development (Chapter 6) to catch problems quickly and providing a VM image with a consistent environment, errors *will* occur. In addition, the Alpha test of the book is going to be 300 times larger than we originally planned and used in 140 countries rather than just in one classroom, so we expect there will be problems with the bookware that we had not anticipated. You can react most productively by remembering the acronym **RASP**: Read, Ask, Search, Post.

Read the error message. Error messages can look disconcertingly long, but a long error message is often your friend because it gives a strong hint of the problem. There will be places to look in the online information associated with the class given the error message.

Ask a coworker. If you have friends in the class, or have instant messaging enabled, put the message out there.

Search for the error message. You'd be amazed at how often experienced developers deal with an error by using a search engine such as Google to look up key words or key phrases in the error message.

Post a question on a site like [StackOverflow](#) or [ServerFault](#) (*after* searching to see if a similar question has been asked!), sites that 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.

Thanks again for your help with the Alpha test, and we appreciate your patience. The bookware consists of three parts.

The first is a uniform development environment preloaded with all the tools referenced in the book. For convenience and uniformity, this environment is

provided as a ***virtual machine image***.

The second comprises a set of excellent SaaS sites aimed at developers: [GitHub](#), [ProjectLocker](#), [Heroku](#), and [Pivotal Tracker](#). **Disclaimer:** At the time of this writing, the *free* offerings from the above sites were sufficient to do the work in this book. However, the providers of those services or tools may decide at any time to start charging, which would be beyond our control.

The third is supplementary material connected to the book, which is free whether you've purchased the book or not:

- The book's web site (<http://saasbook.info>) contains the latest errata for each book version, links to supplementary material online, a bug reporting mechanism if you find errors, and high-resolution renderings of the figures and tables in case you have trouble reading them on your ebook reader
- GitHub (<http://github.com/saasbook>) contains complete code for the various Rotten Potatoes examples used throughout the book
- Pastebin (<http://pastebin.com/u/saasbook>) contains syntax-highlighted, copy-and-pastable code excerpts for every example in the book
- Vimeo (<http://vimeo.com/saasbook>) hosts all the screencasts referenced in the book

Virtual machine (VM) technology allows a single physical machine (your laptop or PC) to run one or more ***guest operating systems (OS)*** "on top of" your machine's built-in OS, in such a way that each guest believes it is running on the real hardware. These virtual machines can be "powered on" and "powered off" at will, without interfering with your computer's built-in OS (the ***host OS***). A ***hypervisor*** is an application that facilitates running VMs. The bookware VM runs on [VirtualBox](#), a free and open-source hypervisor originally developed by Sun Microsystems (now part of Oracle). VirtualBox runs on Linux, Windows, or Mac OS X, as long as the host computer has an Intel-compatible processor.

The development environment in the VM is based on GNU/Linux. Linux is an open-source implementation of the ***kernel*** (core functionality) of Unix, one of the most influential operating systems ever created and the most widely-used environment for SaaS development and deployment. GNU (a recursive acronym for GNU's Not Unix) is a collection of open-source implementations of nearly all of the important Unix applications, especially those used by developers.

ELABORATION: Thinking about installing your own software locally?

The explanations and examples in each version of the book have been cross-checked against the *specific* versions of Ruby, Rails, and other software included in the VM. Changes across versions are significant, and running the book examples with the wrong software versions may result in syntax errors, incorrect behavior, differing

messages, silent failure, or other problems. To avoid confusion, we strongly recommend you use the VM until you are familiar enough with the environment to distinguish errors in your own code from problems arising from incompatible versions of software components. The book's [web site](#) includes a manifest of everything we installed for each version of the VM and an Amazon EC2-compatible version of the VM in case you have trouble using Virtualbox.

Follow the instructions at <http://beta.saasbook.info/bookware-vm-instructions> to download and setup the VM on your computer. Once it's set up, a typical work session will go like this:

Start VirtualBox, select the saasbook VM, and click the Start button.

Do your work.

When you're ready to quit, select Close from the VirtualBox Machine menu. You have three choices of what to do with your VM state.

- *Save virtual machine state* means the next time you resume, the VM will be exactly as you left it, like standby/resume on your host OS.
- *Send shutdown signal* will shut down the guest OS in an orderly way; your changes will be saved, but the next time you restart the VM, any programs that were running *in the VM* will have been shut down. This is like using the shutdown command in Figure [A.1](#).
- *Power off* is like pulling the plug on the VM, and there is a risk that some data may be lost; not recommended except as a last resort if the other options don't work. This is like using the reboot -q command in Figure [A.1](#).

VM networking uses NAT by default, giving your VM a 10.0.0.* address that is routable from your host computer but not from the outside world. VirtualBox offers other networking options if this doesn't work in your setup.

If your host computer is connected to the Internet, your VM will be too. However, if you suspend (sleep) your *host* computer and then wake it up on a different physical network (for example, sleep your computer at work, then travel home and wake it up there), you will need to restart your VM's networking subsystem. To do this, type sudo /etc/init.d/networking restart in a Terminal window in the VM. If this doesn't work, reboot your VM. Figure [A.1](#) gives some useful commands when working in the VM.

Restart VM networking (when sudo /etc/init.d/networking restart host computer is suspended and then resumed on a different network)

Reboot VM gracefully sudo /sbin/reboot

Reboot VM forcibly (if gracefully sudo /sbin/reboot -q

doesn't work)

Shutdown VM gracefully or
forcibly

Substitute shutdown for reboot in above 2 lines

Figure A.1: Some useful "housekeeping" commands that you can type at the Terminal prompt of your VM. Some of these are available via the GUI in the VM, but in the Unix world, it's best to understand the command-line equivalents in case the GUI itself is having problems.

ELABORATION: Free and Open Source Software

Linux was originally created by Finnish programmer Linus Torvalds for his own use. The GNU project was started by Richard Stallman, creator of the Emacs editor and founder of the Free Software Foundation (which stewards GNU), an illustrious developer with very strong opinions about the role of open source software. Both Linux and GNU are constantly being improved by contributions from thousands of collaborators worldwide; in fact, Linus later created Git to manage this large-scale collaboration. Despite the apparent lack of centralized authority in their development, the robustness of GNU and Linux compare favorably to proprietary software developed under a traditional centralized model. This phenomenon is explored in Eric Raymond's [*The Cathedral and the Bazaar*](#), which some consider the seminal manifesto of the Free and Open Source Software (FOSS) movement.

An alternative to developing on your own computer is Amazon Elastic Compute Cloud (EC2). We've prepared a VM image in Amazon Machine Image (AMI) format that has the same software preinstalled as the VirtualBox-compatible VM. The [book's web site](#) has a pointer to the AMI that corresponds to this version of the book, which is 0.9.0, and instructions for deploying and logging in to the VM on EC2. There are two caveats to remember if you use this approach. First, the default security configuration of an EC2 instance will allow you to login to it via ssh, but will not allow you to connect to a SaaS app running on it via HTTP. That is, you cannot run a browser on your local computer and point it at a SaaS app on your EC2 image. There are two ways to get around this. One is to change the default security settings to allow HTTP and HTTPS access, as described in [Amazon's EC2 documentation](#). The other is to run an [X server](#) on your local computer, such as [XMing](#) for Windows or [XQuartz](#) for Mac OS X, and allow the browser to run on the EC2 instance but display its graphics directly on your computer. The [VM information page](#) on the book's website explains how the ssh command line for logging into your instance should be changed if you want to try this option.

The X Window System originated at MIT in 1984 as a way to provide GUI facilities that would work across different operating systems and hardware. X11-based programs send graphical output to and receive user input from an X server program via TCP/IP. Confusingly, because the X server needs to draw on the screen and receive user input from the mouse and keyboard, it often runs on the client computer, with the X client running on a server computer in the cloud!

The second caveat of using EC2 involves what to do when your work session is over: you can either *Stop* or *Terminate* your instance. (Simply logging out of your instance doesn't shut it down; it will continue to run and possibly accumulate charges.) Both have the effect of deactivating your instance so it stops accumulating charges. However, stopping an instance will preserve the virtual hard disk associated with it,

so that any changes you made will persist the next time you restart the instance. During the intervening time, the virtual disk will be stored at Amazon and may incur usage charges once you exceed the Free tier (which as of February 2012 includes 30GB of storage for one year on Amazon Elastic Block Store). If you terminate the instance, the virtual hard disk is *destroyed forever* and any changes you made will *not* be present when you restart the instance. Whichever you use, be sure to use Git and a cloud-based Git backup such as GitHub or ProjectLocker, as described in Section [A.6](#), to back up your important work to the cloud!

You will save yourself a great deal of grief by working with an editor that supports syntax highlighting and automatic indentation for the language you use. You can either edit files directly on the VM, or use the VirtualBox “shared folders” feature to make some directories on your VM available as folders on your Mac or Windows PC so that you can run a native editor on your Mac or PC.

Many [**Integrated Development Environments**](#) (IDEs) that support Ruby, including [Aptana](#), NetBeans and RubyMine, perform syntax highlighting, indentation and other useful tasks. While these IDEs also provide a GUI for other development-related tasks such as running tests, in this book we use command-line tools for these tasks for three reasons. First, unlike IDEs, the command line tools are the same across all platforms. Second, we place heavy emphasis in the book on automation to avoid mistakes and improve productivity; GUI tasks often cannot be automated, whereas command line tools can be composed into scripts, an approach central to the Unix philosophy. Third, understanding what tools are involved in each aspect of development helps roll back the “magic curtain” of IDE GUIs. We believe this is helpful when learning a new system because if something goes wrong while using the GUI, in order to find the problem you need some understanding of how the GUI actually does the tasks.

With this in mind, there are two ways to edit files on the VM. The first is to run an editor on the VM itself. We’ve preinstalled two popular editors on the VM. One is [vim](#), a lighter-weight editor that is customizable enough to include language-aware syntax highlighting and auto-indentation. Here’s a [collection of links](#) to tutorials and screencasts that cover this popular editor. The other is [Emacs](#), the granddaddy of customizable editors and one of the creations of the illustrious [Richard Stallman](#). The [canonical tutorial](#) is provided by the Free Software Foundation, though many others are available. We’ve included automatic support for editing Ruby and Rails apps in both vim and emacs on the VM.

vim stands for “*vi improved*,” since it began as a much-enhanced version of the early Unix editor [vi](#), written in 1976 by Unix legend, Sun co-founder and Berkeley alum [Bill Joy](#).

The second way to edit files is to edit natively on your Mac or Windows computer, which requires setting up the Shared Folders feature of VirtualBox as explained at <http://beta.saasbook.info/bookware-vm-instructions>. Free editors that support Ruby include [TextWrangler](#) for Mac OS X or [Notepad++](#) for Windows.

Don’t ever copy and paste code into or out of a word processor such as Microsoft Word or Pages. Many word processors helpfully convert regular quotes (“”) to “smart quotes,” convert sequences such as two hyphens (--) to em-dashes (—), and

perform other conversions that will make your code incorrect, cause syntax errors, and generally bring you grief. Don't do it.

Version control, also called source code control or software configuration management (SCM), is the process of keeping track of the history of changes to a set of files such that we can tell who made each change and when, reconstruct one or more files as they existed at some point in the past, or selectively combine changes made by different people. A version control system (VCS) is a tool that helps manage this process. For individual developers, SCM provides a timestamped and annotated history of changes to the project and an easy way to undo changes that introduce bugs. Chapter 9 discusses the many additional benefits of SCM for



small teams.

SCM or VCS? Confusingly, the abbreviations SCM and VCS are often used interchangeably.

We will be using Git for version control. Cloud-based Git hosting services like GitHub and ProjectLocker, while not required for Git, are highly desirable because they enable small teams to collaborate conveniently (as Chapter 9 describes) and give individual developers a place to back up their code. We include setup examples for both GitHub and ProjectLocker; once the initial setup is complete, the project management commands work the same way for both. This section covers the basics of Git, and the next two sections cover one-time setup instructions for GitHub and ProjectLocker.

Linus Torvalds invented Git to assist with version control on the Linux project. You should read this section even if you've used other VCSs like Subversion, as Git's conceptual model is quite different.

Like all version control systems, a key concept in Git is the project **repository**, usually shortened to **repo**, which holds the complete change history of some set of files that make up a project. To start using Git to track a project, you first cd to the project's top-level directory and use the command `git init`, which initializes an empty repo based in that directory. **Tracked files** are those that are a permanent part of the repo, so their revision information is maintained and they are backed up; `git add` is used to add a file to the set of tracked files. Not every project file needs to be tracked—for example, intermediate files created automatically as part of the development process, such as log files, are usually untracked.

Screencast [A.6.1](#) illustrates the basic Git workflow. When you start a new project, `git init` sets up the project's root directory as a Git repo. As you create files in your project, for each new file you use `git add filename` to cause the new file to be tracked by Git. When you reach a point where you're happy with the current state of the project, you **commit** the changes: Git prepares a list of all of the changes that will be part of this commit, and opens that list in an editor so you can add a descriptive comment. Which editor to use is determined by a **configuration setting**, as described below. Committing causes a snapshot of the tracked files to be recorded permanently along with the comments. This snapshot is assigned a **commit ID**, a 40-digit hexadecimal number that, surprisingly, is unique in the universe (not just within this Git repo, but across all repos); an example might be

1623f899bda026eb9273cd93c359326b47201f62. This commit ID is the canonical way to refer to the state of the project at that point in time, but as we'll see, Git provides more convenient ways to refer to a commit besides the cumbersome commit ID. One common way is to specify a prefix of the commit that is unique within this repo, such as 1623f8 for the example above.

The [SHA-1 algorithm](#) is used to compute the 40-digit one-way hash of a representation of the entire tree representing the project at that point in time.

To specify that Git should use the vim editor to let you make your changes, you would say `git config --global core.editor 'vim'`. It doesn't matter what directory you're in when you do this, since `--global` specifies that this option should apply to *all* your Git operations in *all* repos. (Most Git configuration variables can also be set on a per-repo basis.) Other useful values for this particular setting are '`mate -w`' for the TextMate editor on MacOS, '`edit -w`' for TextWrangler on MacOS, and the rather unwieldy

`"'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"` for Windows. In all cases, the various quote marks are necessary to prevent spaces from dividing up the name of the editor into multiple command-line arguments.

Unlike MacOS, the Windows shell (command prompt) diverges from Unix conventions, so it's often awkward to get Unix tools to work properly in Windows.

Screencast A.6.1: [Basic Git flow for a single developer](#)

In this simple workflow, `git init` is used to start tracking a project with Git, `git add` and `git commit` are used to add and commit two files. One file is then modified, and when `git status` shows that a tracked file has some changes, `git diff` is used to preview the changes that would be committed. Finally `git commit` is used again to commit the new changes, and `git diff` is used to show the differences between the two committed versions of one of the files, showing that `git diff` can either compare two commits of a file or compare the current state of a file with some previous commit.

It's important to remember that while `git commit` permanently records a snapshot of the current repo state that can be reconstructed at any time in the future, it does *not* create a backup copy of the repo anywhere else, nor make your changes accessible to fellow developers. The next section describes how to use a cloud-based Git hosting service for those purposes.

ELABORATION: Add, commit, and the Git index

The simplified explanation of Git above omits discussion of the *index*, a staging area for changes to be committed. `git add` is used not only to add a new file to the project, but also to stage an existing file's state for committing. So if Alice modifies *existing* file `foo.rb`, she would normally need to explicitly `git add foo.rb` to cause her change to be committed on the next `git commit`. The reason for separating the steps is that `git add` snapshots the file immediately, so even if the commit occurs later, the version that is committed corresponds to the file's state at the time `git add` was used. We simplified the discussion by using `-a` option to `git commit`, which means "commit *all* current changes to tracked files, whether or not `git add` was used to add them." (`git add` is still necessary to add a new file.)

A variety of cloud-based Git hosting services exist. We recommend and give instructions for GitHub and ProjectLocker, both of which offer a free account plan. GitHub gives you as many projects (repos) as you want, but all are publicly readable; ProjectLocker restricts the total amount of data but allows private repos that are off-limits to everyone except people you designate as collaborators. Both services allow you to lift their restrictions when using a paid plan, and your instructor may have a coupon or code that allows you to temporarily upgrade to a paid plan at no charge for the duration of the course.

Whichever you use, the basic concept is the same. You set up a **secret key** that identifies you to the cloud service as an account holder, and configure Git to know this key. You only need one key no matter how many different accounts or repos you have, so you only need to do the key setup once. You can then **push** your repo to the cloud-based service, creating a remote copy of the repo there called a **remote**. Other developers can, with your permission, both push their own changes and **pull** yours and others' changes from that remote.

You'll need to do the following two steps regardless of which service you use:



<http://pastebin.com/tp7ZqYyR>

```
1 cd ~/.ssh  
2 ssh-keygen -t rsa -C "yao@acm.org"  
3 # when prompted for 'File in which to save key', press Return to accept default  
4 # when prompted for passphrase, press Return to leave blank  
5 chmod 0600 *
```

Figure A.2: Create an *ssh* (secure shell) keypair on Linux by replacing `yao@acm.org` with your preferred email address in these commands. After the second command, a message "The key fingerprint is..." indicates success. The last command changes the permissions on the key files so that only you can read them—a requirement for *ssh* to work properly. For the book exercises it's OK to leave the passphrase blank, but this [guide from GitHub](#) explains the extra steps needed to use passphrases and why you

should usually use them.

Tell Git your name and email address, so that in a multi-person project each commit can be tied to the committer:

<http://pastebin.com/pcD5SDs2>

```
1 git config --global user.name 'Andy Yao'  
2 git config --global user.email 'yao@acm.org'  
3
```

Create an *ssh* (secure shell) key pair that will identify you to the cloud hosting service using the commands shown in Figure [A.2](#).

To get started with GitHub:

If you haven't done so already, establish an account at [GitHub](#). You can select the free account, [request a student account](#) if you're a full time student, or ask your instructor if he or she has a code available that will give you a temporary Micro plan (allowing private repositories) without the monthly fee. Follow **only steps 4 and 5 of [Github's instructions for adding your public key](#)** to your GitHub account. (You've already done the subsequent steps about setting your name and email, and we won't need to set up a token, as the remaining steps describe.)

To create a GitHub repo that will be a remote of your existing project repo, fill out and submit the [New Repository](#) form and note the repo name you chose. A good choice is a name that matches the top-level directory of your project, such as `myrottenpotatoes`.

Back on your development machine (the VM), in a terminal window `cd` to the top level of your project's directory (where you previously typed `git init`) and type:

<http://pastebin.com/91JKCtHW>

```
1 git remote add origin git@github.com:myusername/myreponame.git  
2 git push origin master  
3
```

The first command tells Git that you're adding a new remote for your repo

located at GitHub, and that the short name `origin` will be used from now on to refer to that remote. (This name is conventional among Git users for reasons explained in Chapter 9.) The second command tells Git to *push* any changes from your local repo to the `origin` remote that aren't already there.

To get started with ProjectLocker:

Sign up for a [ProjectLocker](#) account if you haven't already. The free plan is sufficient for the work in this book.

When logged into ProjectLocker, in the left-hand navigation bar click Manage Public Keys, then click New Key.

In the New Public Key dialog, for Name enter a memorable name for this key (for example, "SaaSbook" might remind you that it's the key you set up while reading this book). For User Name, enter the email address you used when following the steps in Figure A.2. Lastly, display the public key file with the command `cat ~/.ssh/id_rsa.pub`, and carefully copy and paste the *entire contents* of the file into the Key box, and click Save Public Key.

To create a ProjectLocker repo that will be a remote of your existing project repo, click Add Project from the left-hand navbar and fill out the form, noting the project name you chose. A good choice is a name that matches the top-level directory of your project, such as `myrottenpotatoes`.

Click on the User Home link in the left-hand navbar and you should see a list of your ProjectLocker projects, including the one just created. Note the field "Your Git Location" associated with the project. For user `andrewyao` and project `minimax`, the location will look something like `git-andrewyao@p15.projectlocker.com:minimax.git`.

Back on your development machine (the VM), in a terminal window `cd` to the top level of your project's directory (where you previously typed `git init`) and type:

<http://pastebin.com/AYJ1hM3Y>

```
1 # replace 'andrewyao' with your ProjectLocker account name,  
2 # 'p15' with whatever appears after '@' in the My Git Location  
3 # of your project on ProjectLocker, and 'minimax' with the  
4 # name you chose for 'Add Project' on ProjectLocker  
5 git remote add origin git-andrewyao@p15.projectlocker.com:minimax.git  
6 git push origin master  
7
```

The first command tells Git that you’re adding a new remote for your repo located at ProjectLocker, and that the short name `origin` will be used from now on to refer to that remote. (This name is conventional among GitHub users for reasons explained in Chapter 9.) The second command tells Git to *push* any changes from your local repo to the `origin` remote that aren’t already there.

Confusingly, on return visits to ProjectLocker you cannot login from `projectlocker.com`, but only from portal.projectlocker.com.

The account setup and key management steps above only have to be done once. The process of creating a new repo and using `git remote` to add it must be done for each new project. Each time you use `git push` in a particular repo, you are propagating all changes to the repo since your last push to the remote, which has the nice side effect of keeping an up-to-date backup of your project.

Figure A.3 summarizes the basic Git commands introduced in this chapter, which should be enough to get you started as a solo developer. When you work in a team, you’ll need to use additional Git features and commands introduced in Chapter 9.

Command	What it does	When to use it
<code>git pull</code>	Fetch latest changes from other developers and merge into your repo	Each time you sit down to edit files in a team project
<code>git add <i>file</i></code>	Stage <i>file</i> for commit	When you add a new file that is not yet tracked
<code>git status</code>	See what changes are pending commit and what files are untracked	Before committing, to make sure no important files are listed as “untracked” (if so, use <code>git add</code> to track them)
<code>git diff <i>filename</i></code>	See the differences between the current version of a file and the last committed version	To see what you’ve changed, in case you break something. This command has many more options, some described in Chapter 9.
<code>git commit -a</code>	Commit changes to <i>all</i> (-a) tracked files; an editor window will open where you can type a commit message describing the changes being committed	When you’re at a stable point and want to snapshot the project state, in case you need to roll back to this point later

<code>git checkout <i>filename</i></code>	Reverts a file to the way it looked after its last commit. Warning: any changes you've made since that commit will be lost. This command has many more options, some described in Chapter 9.	When you need to "roll back" one or more files to a known-good version
<code>git push <i>remote-name</i></code>	Push changes in your repo to the remote named <i>remote-name</i> , which if omitted will default to origin if you set up your repo according to instructions in Section A.7	When you want your latest changes to become available to other developers, or to back up your changes to the cloud

Figure A.3: Common Git commands. Some of these commands may seem like arbitrary incantations because they are very specific cases of much more general and powerful commands, and many will make more sense as you learn more of Git's features.

New cloud computing technologies like Heroku make SaaS deployment easier than it's ever been. Create a free [Heroku](#) account if you haven't already; the free account provides enough functionality for the projects in this book. Heroku supports apps in many languages and frameworks. For Rails apps, Heroku provides a gem with the needed functionality, which is preinstalled in the bookware VM. Once you've created a Heroku account, login to your VM and type `heroku login` and `ssh-keygen` at a terminal window to setup a public key pair that will be used for Heroku deployments. You only need to do this step once.

The concepts in Chapter 3 are central to this discussion, so read that chapter first if you haven't already.

Chapter 3 describes the three environments (development, production, testing) defined by Rails; when you deploy to Heroku or any other platform, your deployed app will run in the production environment. There are two important differences between your development environment and Heroku's production environment. One is that Heroku uses the PostgreSQL database rather than SQLite3, so you will need to change your Gemfile to indicate that your app depends on the PostgreSQL connector gem in production, but on SQLite3 in development. The excerpt below shows this change.

Another possible choice for other production environments might be `gem mysql`, which connects to the popular open-source RDBMS MySQL.

<http://pastebin.com/gi4GE8CQ>

```
1 # for Heroku, replace "gem 'sqlite3'" in your Gemfile with this:
2   group :development, :test do
3     # if you already have a 'group :development,:test' block in
your
4   # Gemfile, you can just move the line "gem 'sqlite3'" into it.
5
6   gem 'sqlite3' # use SQLite only in development and testing
7 end
8 group :production do
9   gem 'pg' # use PostgreSQL in production (Heroku)
9 end
```



Second, starting with Rails 3.1, Rails provides facilities that generate CSS from a higher-level format called [Sass](#) and generate JavaScript from a higher-level language called [CoffeeScript](#). To tell Heroku that it's OK to directly serve CSS and JavaScript files when no corresponding Sass and CoffeeScript files are available, you must change the following line in config/environments/production.rb:

<http://pastebin.com/yfWZ020B>

```
1 # in config/environments/production.rb:
2
# BEFORE - what 'rails new' generates for production.rb (1.18)
:
3 config.assets.compile = false
4 # AFTER - this is what you should change it to:
5 config.assets.compile = true
```

As always, since you changed the Gemfile, run `bundle install --without production` to make sure your app's dependencies are satisfied. (`--without production` omits installing the gems inside `group :production`; we recommend this because installation of the pg gem will fail on your computer unless

you happen to have a full PostgreSQL installation.) Once these changes are made, and you're satisfied with the state of your app, you're ready to deploy.

Essentially, Heroku behaves like a Git remote (Section [A.7](#)), except that pushing to that remote has the side-effect of deploying your app.

Heroku deployments are always from the master Git branch. Branches are discussed in Chapter [9](#).

Make sure your app is running correctly and passing all your tests locally.

Remote debugging is always harder. Before you deploy, maximize your confidence in your local copy of the app!

Login to your VM, change to the root directory of the application you want to deploy, and be sure the master branch of your *local* copy of the Git repo is up-to-date, since that's the version that will be deployed.

```
heroku create --stack cedar
```

This command is only necessary the *first* time you deploy a given app. It creates a new Heroku application with a whimsical preassigned name that also determines the URI. For example, if the preassigned name is luminous-coconut-237, your app will be deployed at `http://luminous-coconut-237.herokuapp.com`. You can change your app's name by logging into your Heroku account and clicking My Apps.

```
git push heroku master
```

This command pushes the current head of your Git repo's master branch to Heroku, as if Heroku were a Git remote, and causes Heroku to run `bundle install` on its end to make sure all necessary gems are available and deploy your app. In this version of the `git push` command, we're pushing to the remote called `heroku` (rather than the default `origin`) and specify that we're pushing the master branch, since all Heroku deployments must occur from that branch.

```
heroku ps
```

This command checks the process status (`ps`) of your deployed app. The **State** column should say something like "Up for 10s" meaning that your app has been available for 10 seconds. You can also use `heroku logs` to display the log file of your app, a useful technique if something goes wrong in production that worked fine in development.

```
heroku run rake db:migrate
```

If this is the first deployment of this app, this command will cause its database to be created. If this is the deployment of a new release, it's best to run this command just in case there are pending migrations (if there aren't any, the command safely does nothing). Heroku also has instructions on how to [import the data from your development database](#) to your production database.

For subsequent deployments of the same app, skip step 3. Figure [A.4](#) summarizes how some of the useful commands you've been using in development mode can be applied to the deployed app on Heroku.

Local (development)

```
rails server  
rails console  
rake db:migrate  
more log/development.log heroku logs
```

Heroku (production)

```
git push heroku master  
heroku run console  
heroku run rake db:migrate
```

Figure A.4: How to get the functionality of some useful development-mode commands for the deployed version of your app on Heroku.

ELABORATION: Production best practices

In this streamlined introduction, we're omitting two best practices that [Heroku recommends](#) for "hardening" your app in production. First, our Heroku deployment still uses WEBrick as the presentation tier; Heroku recommends using the streamlined thin webserver for better performance. Second, since subtle differences between SQLite3 and PostgreSQL functionality may cause migration-related problems as your database schemas get more complex, Heroku advises using PostgreSQL in both development and production, which would require installing and configuring PostgreSQL on your VM or other development computer. In general, it's a good idea to keep your development and production environments as similar as possible to avoid hard-to-debug problems in which something works in the development environment but fails in the production environment.

 **Pitfall: Making check-ins (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` to "cherry pick" a subset of changed files to include in a commit: add the specific files you want, and *omit* the `-a` flag to `git commit`.

 **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 your file won't be tracked or backed up. Before you do a commit or a push, use `git status` to see the list of Untracked Files, and `git add` any files in that list that *should* be tracked. You can use the `.gitignore` file to avoid being warned about files you never want to track, such as binary files or temporary files.

 **Pitfall: Confusing commit with push.** `git commit` captures a snapshot of the staged changes in *your* copy of a repo, but no one else will see those changes until you use `git push` to propagate them to other repo(s) such as the origin.

 **Pitfall: Forgetting to reset VM networking when your host computer moves.** Remember that your VM relies on the networking facilities of your host computer. If your host computer moves to a new network, for example if you suspend it at home and wake it up at work, that's like unplugging and reconnecting your host computer's network cable. The VM must therefore also have its (virtual) network

cable disconnected and reconnected, which you can do using the Devices menu in VirtualBox.



Pitfall: Hidden assumptions that differ between development and production environments. Chapter 3 explains how Bundler and the Gemfile automate the management of your app's dependencies on external libraries and how migrations automate making changes to your database. In this appendix we showed how 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!

A.10 To Learn More

- The [Git Community Book](#) is a good online reference that can also be downloaded as a PDF file.

