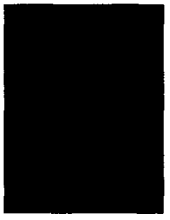
Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing Beta Edition 0.9.0

Armando Fox and David Patterson

August 22, 2012



Contents

1 Engineering Software is Different from Hardware **21**

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.

1.1 Introduction . . . 22
1.2 Product Lifetimes: Independent Products vs. Continuous Improvement . . . 22
1.3 Development Processes: Waterfall vs. Spiral vs. Agile . . . 23
1.4 Assurance: Testing and Formal Methods . . . 27
1.5 Productivity: Conciseness, Synthesis, Reuse, and Tools . . . 29
1.6 Software as a Service . . . 32
1.7 Service Oriented Architecture . . . 35
1.8 Cloud Computing . . . 38
1.9 Fallacies and Pitfalls . . . 40
1.10 Guided Tour of the Book . . . 41
1.11 How *NOT* to Read this Book . . . 44
1.12 Concluding Remarks: Engineering Software is More Than Programming . . . 46
1.13 To Learn More . . . 47
1.14 Suggested Projects . . . 48

2 SaaS Architecture **49**

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.

2.1 100,000 Feet: Client-Server Architecture . . . 50
2.2 50,000 Feet: Communication—HTTP and URIs . . . 52

2.3 10,000 feet: Representation—HTML and CSS . . . 56
2.4 5,000 Feet: 3-Tier Architecture & Horizontal Scaling . . . 59
2.5 1,000 Feet: Model-View-Controller Architecture . . . 62
2.6 500 Feet: Active Record for Models . . . 64
2.7 500 feet: Routes, Controllers, and REST . . . 66
2.8 500 feet: Template Views . . . 69
2.9 Fallacies and Pitfalls . . . 71
2.10 Concluding Remarks: Patterns, Architecture, and Long-Lived APIs . . . 72
2.11 To Learn More . . . 73
2.12 Suggested Projects . . . 74

3 Ruby for Java Programmers 75

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.

3.1 Overview and Three Pillars of Ruby . . . 76
3.2 Everything is an Object . . . 79
3.3 Every Operation is a Method Call . . . 81
3.4 Classes, Methods, and Inheritance . . . 84
3.5 All Programming is Metaprogramming . . . 87
3.6 Blocks: Iterators, Functional Idioms, and Closures . . . 90
3.7 Mix-ins and Duck Typing . . . 93
3.8 Make Your Own Iterators Using Yield . . . 95
3.9 Fallacies and Pitfalls . . . 98
3.10 Concluding Remarks: Idiomatic Language Use . . . 99
3.11 To Learn More . . . 100
3.12 Suggested Projects . . . 100

4 Rails From Zero to CRUD 105

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.

4.1 Rails Basics: From Zero to CRUD . . . 106
4.2 Databases and Migrations . . . 110
4.3 Models: Active Record Basics . . . 112
4.4 Controllers and Views . . . 117
4.5 Debugging: When Things Go Wrong . . . 123
4.6 Form Submission: New and Create . . . 126
4.7 Redirection and the Flash . . . 128

4.8 Finishing CRUD: Edit/Update and Destroy . . . 131
4.9 Fallacies and Pitfalls . . . 135
4.10 Concluding Remarks: Designing for SOA . . . 136
4.11 To Learn More . . . 137
4.12 Suggested Projects . . . 138

5 Validating Requirements: BDD and User Stories 141

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.

5.1 Introduction to Behavior-Driven Design and User Stories . . . 142
5.2 SMART User Stories . . . 144
5.3 Introducing Cucumber and Capybara . . . 146
5.4 Running Cucumber and Capybara . . . 148
5.5 Lo-Fi User Interface Sketches and Storyboards . . . 150
5.6 Enhancing Rotten Potatoes . . . 153
5.7 Explicit vs. Implicit and Imperative vs. Declarative Scenarios . . . 158
5.8 Fallacies and Pitfalls . . . 161
5.9 Concluding Remarks: Pros and Cons of BDD . . . 162
5.10 To Learn More . . . 164
5.11 Suggested Projects . . . 165

6 Verification: Test-Driven Development 167

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.

6.1 Background: A RESTful API and a Ruby Gem . . . 168
6.2 FIRST, TDD, and Getting Started With RSpec . . . 169
6.3 The TDD Cycle: Red-Green-Refactor . . . 172
6.4 More Controller Specs and Refactoring . . . 177
6.5 Fixtures and Factories . . . 180
6.6 TDD for the Model . . . 183
6.7 Stubbing the Internet . . . 187
6.8 Coverage Concepts and Unit vs. Integration Tests . . . 193
6.9 Other Testing Approaches and Terminology . . . 196
6.10 Fallacies and Pitfalls . . . 197
6.11 Concluding Remarks: TDD vs. Conventional Debugging . . . 199
6.12 To Learn More . . . 199
6.13 Suggested Projects . . . 200

7 Improving Productivity: DRY and Concise Rails 203

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.

7.1 DRYing Out MVC: Partials, Validations and Filters . . . 204
7.2 Single Sign-On and Third-Party Authentication . . . 210
7.3 Associations and Foreign Keys . . . 215
7.4 Through-Associations . . . 220
7.5 RESTful Routes for Associations . . . 222
7.6 Composing Queries With Reusable Scopes . . . 225
7.7 Fallacies and Pitfalls . . . 225
7.8 Concluding Remarks: Languages, Productivity, and Beauty . . . 226
7.9 To Learn More . . . 227
7.10 Suggested Projects . . . 228

8 Legacy Software, Refactoring, and Agile Methods 229

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.

8.1 What Makes Code "Legacy" and How Can Agile Help? . . . 231
8.2 Exploring a Legacy Codebase . . . 234
8.3 Establishing Ground Truth With Characterization Tests . . . 238
8.4 Metrics, Code Smells, and SOFA . . . 240
8.5 Method-Level Refactoring: Replacing Dependencies With Seams . . . 244
8.6 Fallacies and Pitfalls . . . 249
8.7 Concluding Remarks: Continuous Refactoring . . . 250
8.8 To Learn More . . . 251
8.9 Suggested Projects . . . 253

9 Working In Teams vs. Individually **255**

Programming is now primarily a team sport, and this chapter covers techniques that can help teams succeed. Everyone on the team must to 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.

9.1 It Takes a Team: Two-Pizza and Scrum . . . 256
9.2 Points, Velocity, and Pivotal Tracker . . . 258
9.3 Pair Programming . . . 260
9.4 Design Reviews and Code Reviews . . . 262
9.5 Version Control for the Two-Pizza Team: Merge Conflicts . . . 264
9.6 Using Branches Effectively . . . 267
9.7 Reporting and Fixing Bugs: The Five R's . . . 271
9.8 Fallacies and Pitfalls . . . 273
9.9 Concluding Remarks: Teams, Collaboration, and Four Decades of Version Control . . . 274
9.10 To Learn More . . . 275
9.11 Suggested Projects . . . 276

10 SOLID Design Patterns for SaaS **279**

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 . . . 280
10.2 Just Enough UML . . . 284
10.3 Single Responsibility Principle . . . 287
10.4 Open/Closed Principle . . . 289
10.5 Liskov Substitution Principle . . . 293
10.6 Dependency Injection Principle . . . 295
10.7 Demeter Principle . . . 299
10.8 Fallacies and Pitfalls . . . 302
10.9 Concluding Remarks: Is Agile Design an Oxymoron? . . . 303
IO.IOTO Learn More . . . 304
10.11 Suggested Projects . . . 306

11 Enhancing SaaS With JavaScript **307**

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.

11.1 JavaScript: The Big Picture . . . 308
11.2 Client-Side JavaScript for Ruby Programmers . . . 310
11.3 Functions and Prototype Inheritance . . . 314
11.4 The Document Object Model and jQuery . . . 319
11.5 Events and Callbacks . . . 321
11.6 AJAX: Asynchronous JavaScript And XML . . . 327
11.7 Fallacies and Pitfalls . . . 332
11.8 Concluding Remarks: JavaScript Past, Present and Future . . . 334
11.9 To Learn More . . . 336
ll.lOSuggested Projects . . . 337

12 Performance, Upgrades, Practical Security **339**

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.

12.1 From Development to Deployment . . . 340
12.2 Quantifying Availability and Responsiveness . . . 342
12.3 Continuous Integration and Continuous Deployment . . . 344
12.4 Upgrades and Feature Flags . . . 346
12.5 Monitoring and Finding Bottlenecks . . . 350
12.6 Improving Rendering and Database Performance With Caching . . . 352
12.7 Avoiding Abusive Queries . . . 356
12.8 Defending Customer Data . . . 359
12.9 Fallacies and Pitfalls . . . 363
12. lOConcluding Remarks: Performance, Security, and Leaky Abstractions . . . 365
12.1 lTo Learn More . . . 365
12.12Suggested Projects . . . 369

13 Looking Backwards and Looking Forwards **371**

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.

13.1 Perspectives on SaaS and SOA . . . 372
13.2 Looking Backwards . . . 373
13.3 Looking Forwards . . . 373
13.4 Evaluating the Book in the Classroom . . . 376
13.5 Last Words . . . 379

13.6 To Learn More 380

Using the Bookware **381**
A.1 Alpha Edition Guidance 382
A.2 Overview of the Bookware 382
A.3 Using the Bookware VM With VirtualBox 383
A.4 Using the Bookware VM on Amazon Elastic Compute Cloud 384
A.5 Working With Code: Editors and Unix Survival Skills 385
A.6 Getting Started With Git for Version Control 386
A.7 Getting Started With GitHub or ProjectLocker 387
A.8 Deploying to the Cloud Using Heroku 391
A.9 Fallacies and Pitfalls 393
A.10 To Learn More 394