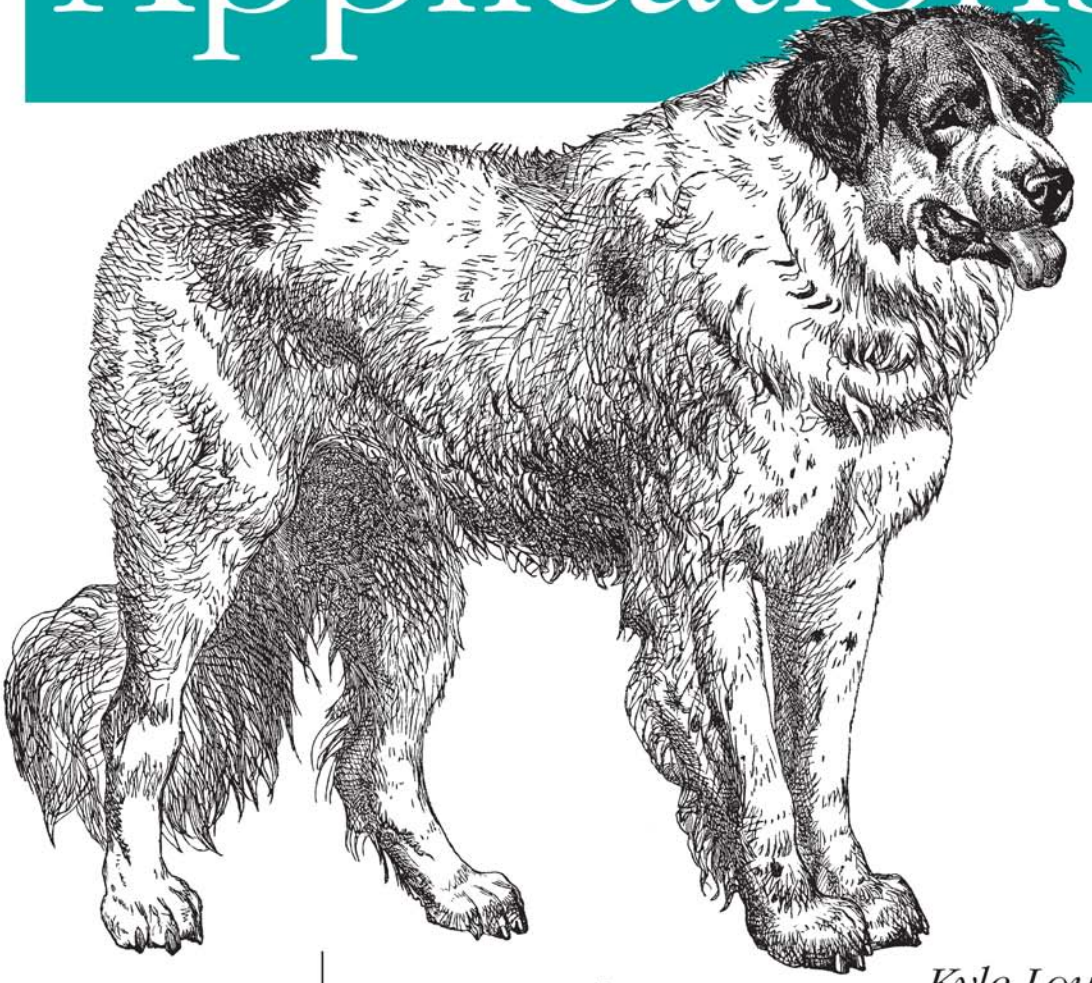
Producing Code That Can Grow and Thrive

Developing

Large Web Applications



O'REILLY® | YAHOO! PRESS

Kyle Loudon
Foreword by Nate Koechley

Developing Large Web Applications

How do you create a mission-critical site that provides exceptional performance while remaining flexible, adaptable, and reliable 24/7? Written by the manager of a UI group at Yahoo!, *Developing Large Web Applications* offers practical steps for building rock-solid applications that remain effective even as you add features, functions, and users. You'll learn how to develop large web applications with the extreme precision required for other types of software.

- Avoid common coding and maintenance headaches as small websites add more pages, code, and programmers
- Get comprehensive solutions for refining HTML, CSS, JavaScript, PHP, and Ajax for large-scale web applications
- Make changes in one place that ripple through all corresponding page elements
- Embrace the virtues of modularity, encapsulation, abstraction, and loosely coupled components
- Use tried-and-true techniques for managing data exchange, including forms and cookies
- Learn often-overlooked best practices in code management and software engineering
- Prepare your code to make performance enhancements and testing easier

"If you're ready to build a finely crafted large site, this is the book for you. Start today, because the world is waiting for your application."

—From the Foreword

Kyle Loudon leads a UI development group at Yahoo!. He previously worked on the user interface for the original Apple iPod, and led the UI group at Jeppesen DataPlan, a Boeing company involved in the development of a flight-planning system used by airlines around the world.

Familiarity with web development tools—including JavaScript, PHP, HTML, and CSS—is recommended.

O'REILLY®
oreilly.com

US $34.99 **CAN $43.99**
ISBN: 978-0-596-80302-5

Developing Large Web Applications

Developing Large Web Applications

Kyle Loudon
foreword by Nate Koechley

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Developing Large Web Applications
by Kyle Loudon

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Andy Oram
Production Editor: Sumita Mukherji
Copyeditor: Amy Thomson
Production Services: Newgen North America, Inc.

Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
March 2010: First Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-80302-5

[M]

1267035305

Table of Contents

Foreword ..... **xi**

Preface ..... **xiii**

1. The Tenets **1**
Managing Complexity 1
Modular Components 3
Achieving Modularity 3
Benefits of Modularity 4
Ten Tenets for Large Web Applications 4

2. Object Orientation **7**
The Fundamentals of OOP 8
Why Object Orientation? 9
UML Class Diagrams 9
Generalization 10
Association 10
Modeling a Web Page 11
Defining Page Types 11
Defining Module Types 11
Writing the Code 12
Achieving Modularity 14
Object-Oriented PHP 15
Classes and Interfaces 15
Inheritance in PHP 19
Object-Oriented JavaScript 22
Objects 22
Inheritance in JavaScript 25

3. Large-Scale HTML **27**
Modular HTML 28

A Bad Example: Using a Table and Presentation Markup 28
A Better Example: Using CSS 30
The Best Example: Semantically Meaningful HTML 31
Benefits of Good HTML 35
HTML Tags 37
Bad HTML Tags 37
Good HTML Tags 38
IDs, Classes, and Names 40
Conventions for Naming 41
XHTML 41
Benefits of XHTML 41
XHTML Guidelines 42
RDFa 45
RDFa Triples 45
Applying RDFa 46
HTML 5 49

4. Large-Scale CSS 51
Modular CSS 52
Including CSS 52
Applying CSS 55
Specificity and Importance 57
Scoping with CSS 58
Standard Module Formats 63
Positioning Techniques 65
CSS Box Model 66
Document Flow 67
Relative Positioning 68
Absolute Positioning 68
Floating 70
Layouts and Containers 71
Example Layouts 72
Example Containers 80
Other Practices 82
Browser Reset CSS 83
Font Normalization 85

5. Large-Scale JavaScript 87
Modular JavaScript 88
Including JavaScript 88
Scoping with JavaScript 90
Working with the DOM 92
Common DOM Methods 92

Popular DOM Libraries 93
Working with Events 98
Event Handling Normalization 99
A Bad Example: Global Data in Event Handlers 99
A Good Example: Object Data in Event Handlers 100
Event-Driven Applications 101
Working with Animation 102
Motion Animation 102
Sizing Animation 103
Color Transition 104
An Example: Chained Selection Lists 105

6. Data Management 115
Dynamic Modules 116
Data Managers 117
Creating Data Managers 120
Extending Data Managers 121
Data Using SQL As a Source 123
An SQL Example 124
Data Using XML As a Source 127
An XML Example 127
Data from Web Services 131
Data in the JSON Format 132
Cookies and Forms 133
Managing Data in Cookies 133
Managing Data from Forms 134

7. Large-Scale PHP 135
Modular Web Pages 136
Generating Pages in PHP 136
Working with Pages 141
Public Interface for the Page Class 141
Abstract Interface for the Page Class 144
Implementation of the Page Class 147
Extending the Page Class 157
Working with Modules 162
Public Interface for the Module Class 162
Abstract Interface for the Module Class 163
Implementation of the Module Class 164
Extending the Module Class 165
An Example Module: Slideshow 165
Layouts and Containers 177
Special Considerations 180

Handling Module Variations 180
Multiple Instances of a Module 181
Dynamic JavaScript and CSS 182
Implementing Nested Modules 182

8. Large-Scale Ajax 185
In the Browser 186
Managing Connections 186
Using Ajax Libraries 189
On the Server 194
Exchange Formats 194
Server Proxies 197
Modular Ajax 198
MVC and Ajax 200
Using Ajax with MVC 201
Public Interface for the Model Object 206
Implementation of the Model Object 207
Public Interface for the View Object 209
Abstract Interface for the View Object 209
View Object Implementation 210
Public Interface for the Connect Object 210
Abstract Interface for the Connect Object 211
Implementation of the Connect Object 212
Controllers 214
An Example of Ajax with MVC: Accordion Lists 215

9. Performance 221
Caching Opportunities 222
Caching CSS and JavaScript 222
Caching Modules 227
Caching for Pages 231
Caching with Ajax 231
Using Expires Headers 233
Managing JavaScript 234
JavaScript Placement 234
JavaScript Minification 234
Removing Duplicates 235
Distribution of Assets 237
Content Delivery Networks 237
Minimizing DNS Lookups 237
Minimizing HTTP Requests 238
Control Over Site Metrics 241
Modular Testing 243

Using Test Data 243
Creating Test Data 245

10. Application Architecture 247
Thinking Modularly 247
Organizing Components 248
Sitewide Architecture 248
Section Architecture 254
Architecture for Pages 256
Architecture and Maintenance 258
Reorganizing Module Uses 258
Adding Module Variations 261
Making Widespread Changes 263
Changes in Data Sources 266
Exposing Modules Externally 268

Index 271

Foreword

As a little kid, I wondered if I would be big and strong when I grew up. There were a lot of aspects to growing well. Would I be healthy? Useful? Productive? Successful?

Websites start out small, too. But these humble sites share my childhood dreams. They want to help more people in more ways; they want to be durable and reliable; they want to be indispensable and to live forever. In short: they want to be large and successful.

But growing up is hard to do. Challenges accumulate and complexity snowballs.

> Expansion means complexity and complexity decay.
>
> —C. Northcote Parkinson

I've seen it. The inevitable challenges of growth in websites—data management, performance—become crippling if mishandled. Things you thought were straightforward, like HTML, start giving you headaches. From front to back, JavaScript to PHP, harmony is displaced by dissonance.

> Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.
>
> —Alan Perlis

I've worked hand-in-hand with Kyle on some of the Web's largest applications. I've watched him craft CSS systems to make sprawling sites skinable and design Ajax architectures that adapt to and enhance the sites. He emerges from the trenches on top every time. He's a perpetual teacher, and, like the best in any discipline, also a perpetual student. We all benefit from his expertise.

Kyle shares his genius and hard-won expertise in this valuable book that will prepare you and your application for scale and success. The book is well structured and readable, with memorable tenets supported by savvy insights, sound philosophy, and fully functioning code examples. Complexity is inevitable, but success rewards the prepared.

> The way to build a complex system that works is to build it from very simple systems that work.
>
> —Kevin Kelly

During this book's deft tour of the complete web application stack, Kyle, the perfect guide, converts lines of explanatory code from one context into insightful tips in another. Build big by thinking small. Build new by thinking old. Manage scope. Boost signal and reduce noise. Resist breakage...these things are easy to rattle off, but it takes an author like Kyle, and a book like this, to make them practical and real.

If you're ready to build a finely crafted large site, this is the book for you. Learn what it takes, because today's compromise is tomorrow's constraint. Start today, because the world is waiting for your application.

Grow large and prosper.

—Nate Koechley
San Francisco, January 2010

Preface

It's been a while since I first worked on a book with O'Reilly in 1997. That book was a practical guide to data structures and algorithms, a subject that, for the most part, had been defined many years before by some of the early giants of computer science (Dijkstra, Hoare, Knuth, to name a few). By comparison, I've been able to witness the rapid evolution of the subject of this book from the front lines, and I have had the good fortune to help refine it myself while working as a web developer at one of the largest web applications in the world, Yahoo!.

Web developers have a fascinating role. We work just as closely with user experience designers as with engineers, and sometimes we're the designers, too. In many ways, we are guardians of the user experience as a web design goes from its mockup to its implementation. But we also have to write exceptionally good code that performs well in the challenging environment of web browsers. Today, more than ever, engineers recognize that web development must be carried out with the same rigor as other types of software development.

This book presents a number of techniques for applying established practices of good software engineering to web development—that is, development primarily using the disparate technologies of HTML, CSS, JavaScript, and server-side scripting languages. Whereas there are many books on how to use languages, how to use libraries, and how to approach software engineering, this is the first book to codify many of the techniques it presents. These techniques will make the components of your own web applications more reusable, maintainable, and reliable.

Audience

The primary audience for this book is software developers and managers interested in large web applications; however, you'll find that the techniques in this book are equally useful for web applications of any size. Although it's especially important to follow good development practices in large web applications, smaller web applications benefit from many of the same techniques, too.

To get the most out of this book, you should already be very familiar with HTML, CSS, and JavaScript; this book does not teach these languages, although it covers many interesting aspects about them. This book uses PHP as the scripting language for server-side examples. Many readers will have a good understanding of PHP as well, but even those who don't should find the examples easy to follow. PHP is known for its flexibility, ubiquity, and ease of use, so it works well. Most examples can be translated to other server-side scripting languages fairly easily, if you desire.

Organization of This Book

This book is organized into three types of material: background (e.g., Object Orientation in Chapter 2), techniques associated with specific languages (e.g., Large-Scale HTML in Chapter 3, Large-Scale CSS in Chapter 4, Large-Scale JavaScript in Chapter 5, and Large-Scale PHP in Chapter 7), and techniques related to other aspects of development (e.g., Data Management in Chapter 6, Large-Scale Ajax in Chapter 8, Performance in Chapter 9, and Application Architecture in Chapter 10). Each chapter begins with a tenet presented from Chapter 1. These tenets act as assertions about the topic for each chapter to provide a concisely articulated direction.

Throughout the book, there are numerous examples in real code to demonstrate many of the techniques presented. Some of the numbered examples work together to create larger, more complete examples that extend across multiple chapters. While the focus of this book is not on teaching the specific languages addressed, the examples do demonstrate a number of aspects of each language that will help make you more proficient with each as you master them.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, filenames, and Unix utilities.

`Constant width`
: Indicates command-line options, variables and other code elements, HTML tags, the contents of files, and the output from commands.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values.

This icon signifies a tip, suggestion, or general note.

There are some other conventions to be aware of in this book:

`...`
: Indicates something that is missing (for you to fill in) in a line of code or a path (e.g., `require_once(.../navbar.inc);`).

`<?php ... ?>`
: Wraps PHP examples that contain the complete code for a file. Most PHP examples don't have this, because they show only a code snippet.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Developing Large Web Applications*, by Kyle Loudon. Copyright Yahoo!, Inc., 978-0-596-80302-5."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

We'd Like to Hear From You

Every example has in this book has been tested on various platforms, but occasionally you may encounter problems. The information in this book has also been verified at each step of the production process. However, mistakes and oversights can occur and we will gratefully receive details of any you find, as well as any suggestions you would like to make for future editions. You can contact the author and editors at:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596803025

You can find additional information about this book, including electronic versions of the examples at:

http://kyleloudon.com

To comment or ask technical questions about this book, send email to the following address, mentioning its ISBN number (978-0-596-80302-5):

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

Acknowledgments

This book is the result of having worked with outstanding people both at O'Reilly and in many projects leading up to the book. For this, I offer my heartfelt thanks.

First, I thank my editor at O'Reilly, Andy Oram. Andy and I worked together on my first book with O'Reilly, and I had hoped for a long time that working on another book together would not be a matter of if, but when. Having finished this book, I hope that another book will be a matter of when again. Andy inspires me with his ability to always find ways to make things better. His insights appear in one form or another on nearly every page of this book. Andy also kept our project moving along while being patient and understanding of the struggle that writers doing other jobs constantly face.

I also extend my sincere thanks to the entire production team at O'Reilly, who constantly impress me with their ability to handle the numerous aspects of production so smoothly. The ease with which it all seems to take place belies the work that it really requires. I would also like to thank Amy Thomson, my copyeditor, for having worked under such tight time constraints at the end of the book.

I send my heartfelt thanks to Nate Koechley for writing the foreword. Nate was one of my earliest colleagues at Yahoo! to turn me on to the truly awesome potential of web development. Much of what I've tried to capture in this book came from ideas that Nate worked passionately to instill at Yahoo! and across the Web. I couldn't have asked for a more fitting person to write the foreword.

I am grateful to have had outstanding technical reviewers for this book as well. Christoph Dorn, Steve Griffith, and Nate Koechley each provided an impressive level of detail and thought in their reviews. The book benefited greatly from their comments.

I would also like to acknowledge the influence of my many colleagues at Yahoo! and other projects before this. I especially thank Bryce Kujala and Vy Phan, who helped refine many of the ideas in the book by putting them to the test in practice early on. I'm also grateful to the exceptional user experience designers with whom I've had the honor to work closest: Veronica Gaspari, Cathy Tiritoglu, and Sasha Verhage.

Finally, I thank Shala, my wife, for her encouragement on another book project; my parents, Marc and Judy, for their support from afar; Shala's parents, Elias and Maria, for their frequent assistance at a moment's notice; and Julian, who has been my late-night companion—just too young to know it yet.

CHAPTER 1

The Tenets

As applications on the Web become larger and larger, how can web developers manage the complexity? In many ways, we need to turn to some of the same good practices used in other types of software development. Generally speaking, these practices are not yet pervasive in web development—that is, in software development primarily using HTML, CSS, JavaScript, and various server-side scripting languages (we'll use PHP for the server-side scripting in this book, but the same principles apply to many other languages). Furthermore, the uniqueness of these technologies poses a challenge for developers trying to apply good practices in a cohesive way.

One of the themes that you'll see repeated in this book is the importance of extending modular development practices to web development. This book presents concrete, practical techniques to achieve modularity in large web applications. In the process, we'll explore many of the finer aspects of HTML, CSS, JavaScript, and PHP. You'll find that most of the techniques are relatively simple to apply, and none rely on the use of specific frameworks. That said, it's important to realize that they don't preclude you from using various frameworks, either; to the contrary, these techniques create a better landscape in which to use many frameworks. As a case in point, we'll look at several examples that utilize the Yahoo! User Interface (YUI) JavaScript library.

At the outset, it's important to establish why the techniques that we're going to explore in this book are especially useful for web developers working on large web applications. We'll begin by looking at some of the factors that contribute to the complexity of many large web applications. Then we'll explore how modularity plays an important role in managing this complexity. Last, we'll examine a list of tenets that will guide our discussions throughout the rest of the book.

Managing Complexity

If you consider how different the Internet is today from just 10 years ago, it's clear how complicated web applications have become and just how quickly the changes have taken place. Far too often, this complexity makes large web applications difficult to

maintain, less reliable, and more costly over their lifetimes. Let's examine some factors that contribute to the complexity of many large web applications. Typically, large web applications have the following characteristics:

Continuous availability
: Most large web applications must be running 24/7. In addition, response times have to be fast at any moment, even at peak load times. Web developers need to write code that is especially robust.

Large user base
: Large web applications usually have large numbers of users. This necessitates management of a large number of simultaneous connections or layers of caching. Web developers often need to write code to manage these situations.

Piece-by-piece delivery
: Whereas many types of software are distributed as complete units, web applications have many parts delivered page by page, or connection by connection via Ajax. As a result, large web applications operate within an environment effectively shared by a huge number of users.

Diversity
: It's hard to think of a business or service that doesn't have at least some sort of web interface. For example, we see financial applications, ticketing sites, applications that organize massive amounts of data (e.g., search engines), media systems (e.g., news sites), and the list goes on. Web developers need to write code that may be reused in unexpected places.

Longevity
: The largest web applications today, even those that have been around many years, are just at the beginning of their lifetimes. Web developers need to write code under the assumption that it will have to stand up to years of changes and maintenance.

Multiple environments
: The Web is a fast-changing landscape littered with old browsers and other devices that can be difficult to support. Users access large web applications from all types of environments and with screens of wildly different sizes (including mobile devices). Web developers must write code that can handle the numerous idiosyncrasies that result from this.

Real-time updates
: Large web applications are not static; they are constantly fluctuating applications for which changes are typically pushed to servers regularly. Web developers need to write code to address this potential for moving parts.

Over time, web developers often end up addressing complexity in large web applications via one-off fixes and tweaks as their applications reach various breaking points. But there is a better way. This book will show you how to address challenges like the ones above head-on from the start. Mitigating the complexity from these challenges can often be attributed to one or more byproducts of modularity that we'll examine in

a moment. For example, I stated above that large web applications need to be available all the time. From the perspective of the web developer, this is an issue of reliability, which modularity plays a key role in addressing.

Modular Components

In a large web application, to address complexity with modular components, or *modules*, you need to encapsulate everything the component needs within small, well-defined pieces of the application. This allows you to divide a large application into more manageable pieces that you can build with a specific focus and reuse wherever needed. In addition, you hide (or abstract) the implementation details of each module and take steps to ensure each module can operate more or less independently of the others. Even relatively small web applications can benefit from such modularity. After all, the small web applications of today are the Googles, Yahoo!s, and Amazons of tomorrow. Building web applications in a modular way at the start provides a solid foundation for future growth.

Modularity seems like a simple thing, but it can be deceptively difficult to attain in a cohesive manner across the HTML, CSS, JavaScript, and server-side scripts that web developers write for large web applications. Let's look more closely at the concept of modularity and some basic ideas to help you achieve it.

Achieving Modularity

You achieve modularity in large web applications, as in other types of software, through *encapsulation*, *abstraction*, and maintaining as much of a *loose coupling* as possible among an application's modules.

Encapsulation

Encapsulation is the process of bundling everything that a module requires within a single, cohesive unit. Modules for web applications need to encapsulate all the HTML, CSS, JavaScript, and PHP that they require. Chapter 7 shows how to accomplish this using object-oriented PHP. We'll also have to employ techniques in the HTML, CSS, and JavaScript itself for a module to support this. These techniques are presented in Chapters 3, 4, and 5, respectively.

Abstraction

Abstraction is the process of hiding details that should not be observable when working with a module from outside its implementation. Defining a good interface is the key to abstraction. Web applications present special challenges because HTML is not built for hiding information in the manner that many languages enforce and because CSS cuts across module boundaries, or *cascades*, and must therefore be managed rigorously. Chapter 7 shows how object-oriented PHP can be used to define interfaces that abstract

the details of working with sections of HTML and CSS. Data managers, presented in Chapter 6, are good examples of interfaces that abstract working with dynamic data managed by the backend.

Loose coupling

A loose coupling between modules means that one depends on the other as little as possible and only in clearly defined ways. In Chapter 2, you'll see that it's relatively easy to create a dependency graph that depicts the dependencies among objects in an object-oriented system. When object orientation is used to implement modules in a large web application, it's easier to notice how changes to one part of a system may affect another. The techniques presented for modular HTML and CSS in Chapters 3 and 4 also promote loose coupling.

Benefits of Modularity

Once you create modules that take advantage of encapsulation, abstraction, and loose coupling, you'll have better *reusability*, *maintainability*, and *reliability* across your entire web application. When a module is reusable, it's clear how to make it operate and do new things. You can move it from place to place in the application with the confidence that everything it needs to work properly will come with it, and that you won't experience unexpected consequences when you reuse the module. When a module is maintainable, it's clear which area of the application you need to change to affect certain features and how to make the changes. Modules that are easy to maintain are more reliable because they reduce the risk of side effects when changes take place.

Ten Tenets for Large Web Applications

With an eye toward modularity, we'll use the list of tenets in this section to guide our discussions in the chapters that follow. A tenet is a principle or belief that we will accept as true—that is, it's an assertion. A quick read-through of this list will provide you with a high-level understanding of where we'll be heading. Later, you may find this list to be a concise reminder of the concepts we've discussed. Each tenet is examined in the chapter with the corresponding number. For instance, Tenet 4 describes the robust use of CSS, and Chapter 4 shows how to apply this tenet when implementing your CSS.

Tenet 1
: Large web applications are built from modular components that are highly reusable, maintainable, and reliable.

Tenet 2
: The use of object orientation in JavaScript and server-side scripting languages improves reusability, maintainability, and reliability in large web applications by promoting modularity.

Tenet 3

Large-scale HTML is semantic, devoid of presentation elements other than those inherent in the information architecture, and pluggable into a wide variety of contexts in the form of easily identifiable sections.

Tenet 4

Large-scale CSS forms a layer of presentation that is separate from the information architecture, applied in a modular fashion, and free of side effects as we reuse modules in various contexts.

Tenet 5

Large-scale JavaScript forms a layer of behavior applied in a modular and object-oriented fashion that prevents side effects as we reuse modules in various contexts.

Tenet 6

Dynamic data exchanged between the user interface and the backend is managed through a clearly defined data interface. Pages define a single point for loading data and a single point for saving it.

Tenet 7

Pages are constructed from highly reusable modules that encapsulate everything required (e.g., HTML, CSS, JavaScript, and anything else) to make each module an independently functioning and cohesive unit that can be used in a variety of contexts across various pages.

Tenet 8

Large-scale Ajax is portable and modular, and it maintains a clear separation between data changes and updates to a presentation. Data exchange between the browser and server is managed through a clearly defined interface.

Tenet 9

Large-scale HTML, JavaScript, CSS, and PHP provide a good foundation on which to build large web applications that perform well. They also facilitate a good environment for capturing site metrics and testing.

Tenet 10

The organization of files on the server for a large web application reflects the architecture of the application itself, including clearly demarcated scopes in which each file will be used.

CHAPTER 2
Object Orientation

Object orientation has been at the heart of many types of software for years, but web developers using JavaScript and server-side scripting languages generally haven't been as fast to embrace it. In the early days, websites had fairly simple scripting needs, so object orientation wasn't crucial. Today, with the complexity of software on the Web, an understanding of object orientation is fundamental to approaching the development of large web applications with the same rigor as other types of software.

Throughout this book, we'll explore many examples that use object orientation in PHP and JavaScript, so it's worth spending a little time to examine its importance and how both languages address it. Both PHP and JavaScript have powerful support for object orientation, but each implements it in different ways. One of the fundamental differences is that PHP is a *class-based language*. In class-based languages, you declare and extend classes, which you then instantiate as objects wherever needed. This is object orientation as C++ and Java developers know it. On the other hand, JavaScript is a *prototype-based* (or *object-based*) *language*. In prototype-based languages, there are no classes; you create objects on the fly and derive new ones using existing objects as prototypes. Whatever the language, the following tenet (first described in Chapter 1) articulates what we expect to achieve with an object-oriented implementation:

> Tenet 2: The use of object orientation in JavaScript and server-side scripting languages improves reusability, maintainability, and reliability in large web applications by promoting modularity.

This chapter begins with an overview of object orientation and why it's a good approach in general. To this end, we'll examine the fact that one of the most important reasons for its popularity, often not easily articulated even by experienced developers, is that it helps narrow the semantic gap between real-world problems and the solutions that you build in software. Next, we'll look at an example of how to visualize a web page in an object-oriented way. Finally, we'll explore some of the details behind object orientation in PHP and JavaScript. We won't cover all aspects of object orientation for each, but we'll explore the core features with an emphasis on those used in this book.

The Fundamentals of OOP

The fundamental idea behind object-oriented programming (OOP) is to group together data (called *data members* in object-oriented parlance) with the operations that do things with it (called *methods*). In PHP, you define which data members and methods go together by placing them in a *class*, as shown in Example 2-1. In JavaScript, the details are little different (as you'll see later in the chapter), but the idea is the same.

Example 2-1. A simple class in PHP

```
class Account
{
   protected $balance;
   protected $minimum;

   public function __construct($amount, $lowest)
   {
      // Initialize the data members when a new object is instantiated.
      $this->balance = $amount;
      $this->minimum = $lowest;
   }

   public function deposit($amount)
   {
      $this->balance += $amount;
   }

   public function withdraw($amount)
   {
      $this->balance -= $amount;
   }
}
```

Note the use of `$this` in each method. It refers to the particular instance of the object that's currently being manipulated. This way, each account can have a different balance and minimum, and when a deposit or withdrawal is made, it will always affect the correct balance—the one for the account on which the deposit or withdrawal is invoked.

Once you have the data members and methods bundled together, you create an instance of the bundle as an *object* and invoke the methods to carry out various operations, as shown in Example 2-2. In this example, each object is a separate instance of the class, so after depositing $100.75 into `$account1`, that object has a balance of $600.75, while `$account2` still has a balance of $300.00. Of course, there is a lot more behind object orientation than this, but this example illustrates some of the fundamental ideas.

Example 2-2. Using an object in PHP

```
$account1 = new Account(500, 200);
$account2 = new Account(300, 200);

...

$account1->deposit(100.75);
```

Why Object Orientation?

To understand why object orientation is a good approach to software development, it helps to think about the following:

- What's the natural way in which people tend to think about problems and, as a result, build cognitive models to solve them?
- How can we draw visual models from our cognitive models in a standard way so that we can work with them?
- How can software developers write code to resemble as closely as possible these visual, and hence cognitive, models?
- How can we abstract problems in common ways to allow developers with diverse backgrounds to collaborate?

In programs that are not object-oriented (programs written using traditional procedural languages, for example), the models we create tend to be abstractions of the computer itself: functions are abstractions of processors, data structures are abstractions of memory, etc. Since most real-world problems don't look like a computer, this creates a disconnect. Object-oriented systems more closely resemble our cognitive models and the visual models drawn from them. As a result, object orientation fundamentally narrows the semantic gap between the natural way we think about problems and how we work with them on a computer.

To illustrate how object orientation narrows this gap, we'll need a visual model that we can translate into code. Let's look at a standard approach for drawing visual models from cognitive ones: the *Unified Modeling Language* (UML). Although UML defines nine types of diagrams to describe components and their interactions or relationships in systems, we'll focus on just one: the *class diagram*.

UML Class Diagrams

If someone were to ask you to draw a model of how various components of a banking system were related, you would most likely draw some combination of boxes or circles for the components, and lines or arrows to show their relationships. UML class diagrams standardize this rather natural way to depict things. Stated formally, a UML class diagram is a directed graph in which nodes (boxes) represent classes of objects and

edges (lines or arrows) represent relationships between the classes. Two examples of some of the relationships between classes are *generalization* and *association*.

Generalization

Generalization is the relationship between one class and a more general form of it. This type of relationship is sometimes called an "is-a" relationship because one class *is a* more specialized form of the general one. For example, a checking account is a more specialized form of a bank account. In UML, you show generalization by drawing a hollow arrow from a box labeled with the more specialized class to a box labeled with the more general class, as shown in Figure 2-1.

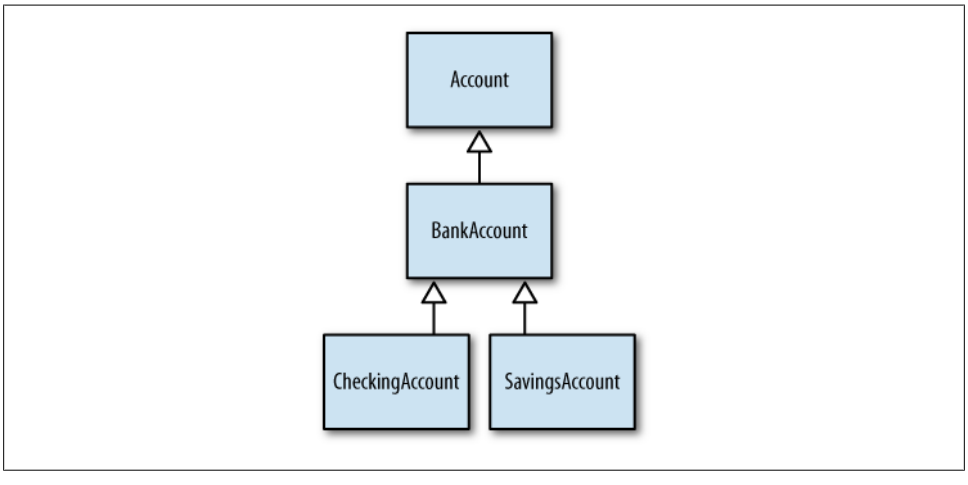


Figure 2-1. Classes related by generalization

Association

Association is the relationship between one class and a class of something it contains. This type of relationship is sometimes called a "has-a" relationship because one class *has a* member within it. For example, a customer at a bank has an account. In UML, you show an association by drawing an arrow from a box labeled with the containing class to a box labeled with the class of its member, as shown in Figure 2-2 (you can omit the arrowheads if both classes contain each other). You can also include a number of instances next to the arrowhead or use an asterisk (*) to indicate any number of instances, if you know these details.

Other items that you may include within the box itself are the operations the class may perform and data members associated with the class that don't aggregate any data themselves (e.g., an account balance that is simply a number). Figure 2-2 shows examples of these items as well.

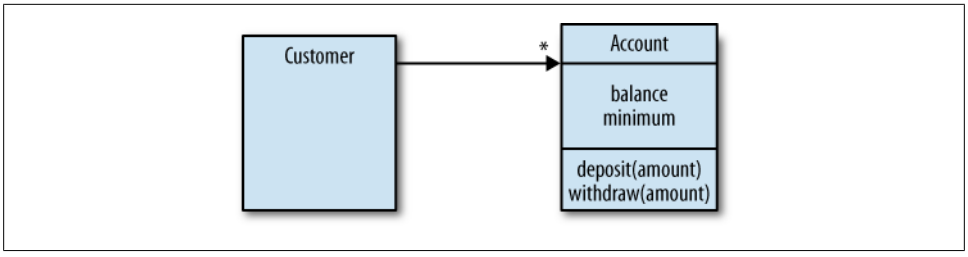


Figure 2-2. Classes related by association

Modeling a Web Page

Now let's turn to the class diagram for a web page, shown in Figure 2-3. This diagram captures the way we're going to think about web pages when we write some PHP code to build them in a modular fashion in Chapter 7. It doesn't capture all the details of the implementation in Chapter 7, but it shows enough to illustrate the resemblance between this type of model and its object-oriented code in Example 2-3. Our diagram has two fundamental entities: *pages* and *modules*.

Defining Page Types

The most general form of a page is represented by a class called `Page`, which represents the data and operations for web pages across all types of web applications. We also recognize the need for more specialized classes of pages on a specific website (e.g., `SitePage`), pages on certain sections of the site (e.g., `NewCarsPage`), and pages for very specific purposes (e.g., `NewCarSearchResultsPage`). The diagram further conveys the fundamental use of modules in pages. Every page is composed of modules, and a module conversely has an association with the page on which it resides. This way, it can add what it needs to the page.

To keep things simple for now, the diagram shows just one data member for `Page`, `js_module`, and one operation, `add_to_js`. In Chapter 7, you'll see that modules need a way to add their JavaScript to a page, and `add_to_js` provides a way to do this. Pages need someplace to collect the JavaScript, and the diagram shows the data member `js_module` for just this purpose.

Defining Module Types

The most general class of module is `Module`, which represents the data and operations common to all modules. We also recognize the need for more specialized classes of modules to implement modules for very specific purposes (e.g., `NavBar`), as well as layouts and containers, which are themselves more specific forms of `Layout`.