

A Brain-Friendly Guide

Head First Design Patterns

Avoid those
embarrassing
coupling mistakes

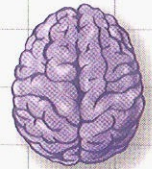


Learn why everything
your friends know about
Factory pattern is
probably
wrong



Discover the secrets
of the Patterns Guru

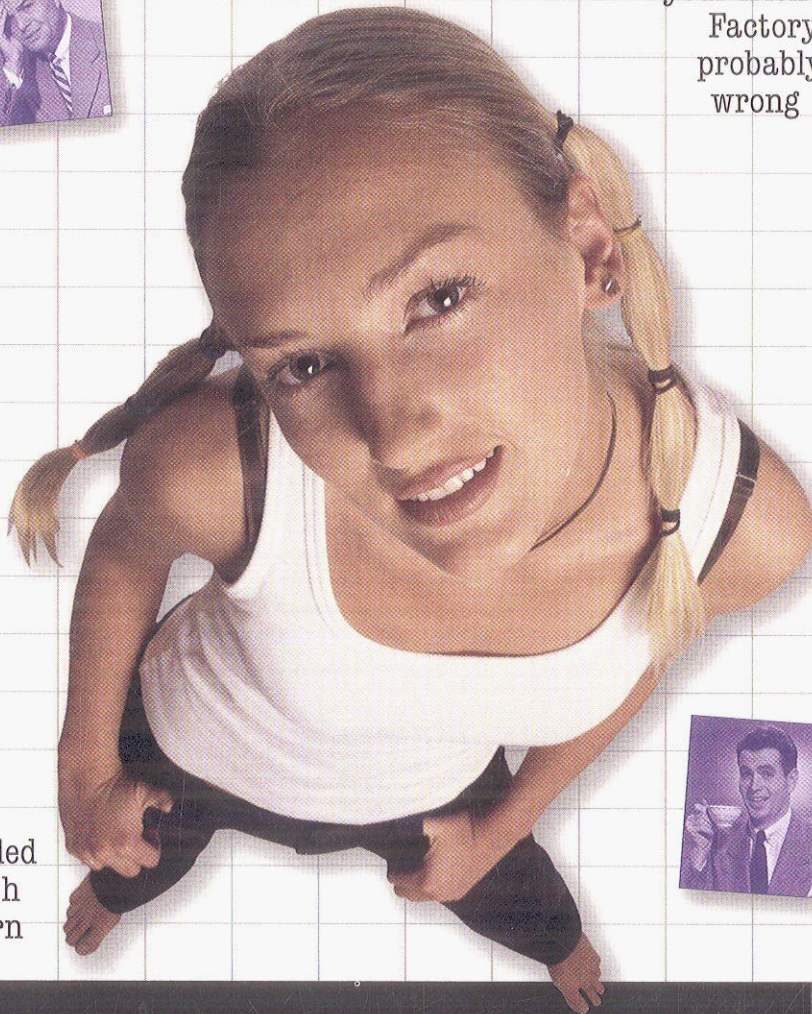
Load the patterns
that matter straight
into your brain



Find out how
Starbuzz Coffee doubled
their stock price with
the Decorator pattern



See why Jim's
love life improved
when he cut down
his inheritance



O'REILLY®

Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

Table of Contents (summary)

	Intro	xxv
1	Welcome to Design Patterns: <i>an introduction</i>	1
2	Keeping your Objects in the know: <i>the Observer Pattern</i>	37
3	Decorating Objects: <i>the Decorator Pattern</i>	79
4	Baking with OO goodness: <i>the Factory Pattern</i>	109
5	One of a Kind Objects: <i>the Singleton Pattern</i>	169
6	Encapsulating Invocation: <i>the Command Pattern</i>	191
7	Being Adaptive: <i>the Adapter and Facade Patterns</i>	235
8	Encapsulating Algorithms: <i>the Template Method Pattern</i>	275
9	Well-managed Collections: <i>the Iterator and Composite Patterns</i>	315
10	The State of Things: <i>the State Pattern</i>	385
11	Controlling Object Access: <i>the Proxy Pattern</i>	429
12	Patterns of Patterns: <i>Compound Patterns</i>	499
13	Patterns in the Real World: <i>Better Living with Patterns</i>	577
14	Appendix: <i>Leftover Patterns</i>	611

Table of Contents (the real thing)

Intro

Your brain on Design Patterns. Here you are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how *do* you trick your brain into thinking that your life depends on knowing Design Patterns?

Who is this book for?	xxvi
We know what your brain is thinking	xxvii
Metacognition	xxix
Bend your brain into submission	xxxi
Technical reviewers	xxxiv
Acknowledgements	xxxv

1

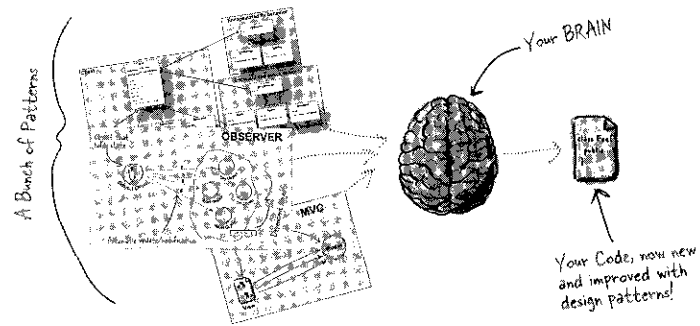
Welcome to Design Patterns

Someone has already solved your problems. In this chapter, you'll learn why (and how) you can exploit the wisdom and lessons learned by other developers who've been down the same design problem road and survived the trip. Before we're done, we'll look at the use and benefits of design patterns, look at some key OO design principles, and walk through an example of how one pattern works. The best way to use patterns is to *load your brain* with them and then *recognize places* in your designs and existing applications where you can *apply them*. Instead of *code reuse*, with patterns you get *experience reuse*.

Remember, knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.



The SimUDuck app	2
Joe thinks about inheritance...	5
How about an interface?	6
The one constant in software development	8
Separating what changes from what stays the same	10
Designing the Duck Behaviors	11
Testing the Duck code	18
Setting behavior dynamically	20
The Big Picture on encapsulated behaviors	22
IIAS-A can be better than IS-A	23
The Strategy Pattern	24
The power of a shared pattern vocabulary	28
How do I use Design Patterns?	29
Tools for your Design Toolbox	32
Exercise Solutions	34



the Observer Pattern

2

Keeping your Objects in the Know

Don't miss out when something interesting happens!

We've got a pattern that keeps your objects in the know when something they might care about happens. Objects can even decide at runtime whether they want to be kept informed. The Observer Pattern is one of the most heavily used patterns in the JDK, and it's incredibly useful. Before we're done, we'll also look at one to many relationships and loose coupling (yeah, that's right, we said coupling). With Observer, you'll be the life of the Patterns Party.

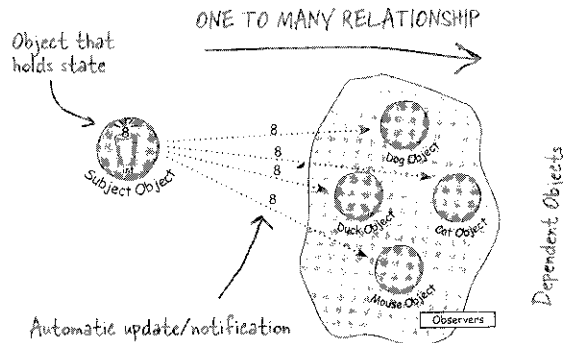
OO Basics

Abstraction
Encapsulation
Polymorphism

OO Principles

- Encapsulate what varies
- Favor Composition over inheritance
- Program to Interfaces, not implementations
- Strive for loosely coupled designs between objects that interact

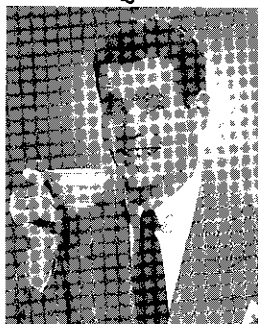
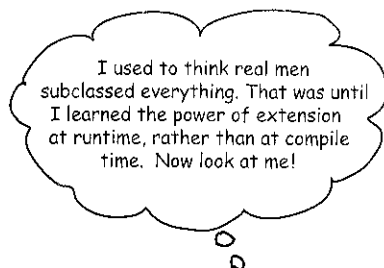
The Weather Monitoring application	39
Meet the Observer Pattern	44
Publishers + Subscribers = Observer Pattern	45
Five minute drama: a subject for observation	48
The Observer Pattern defined	51
The power of Loose Coupling	53
Designing the Weather Station	56
Implementing the Weather Station	57
Using Java's built-in Observer Pattern	64
The dark side of java.util.Observable	71
Tools for your Design Toolbox	74
Exercise Solutions	78



3

Decorating Objects

Just call this chapter “Design Eye for the Inheritance Guy.” We'll re-examine the typical overuse of inheritance and you'll learn how to decorate your classes at runtime using a form of object composition. Why? Once you know the techniques of decorating, you'll be able to give your (or someone else's) objects new responsibilities *without making any code changes to the underlying classes.*



Welcome to Starbuzz Coffee	80
The Open-Closed Principle	86
Meet the Decorator Pattern	88
Constructing a Drink Order with Decorators	89
The Decorator Pattern Defined	91
Decorating our Beverages	92
Writing the Starbuzz code	95
Real World Decorators: Java I/O	100
Writing your own Java I/O Decorator	102
Tools for your Design Toolbox	105
Exercise Solutions	106

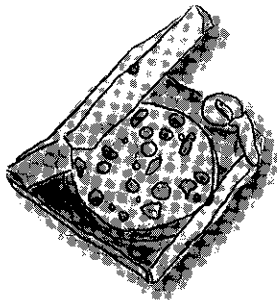
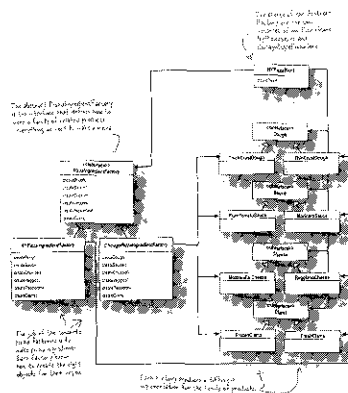
the Factory Pattern

4

Baking with OO Goodness

Get ready to cook some loosely coupled OO designs.

There is more to making objects than just using the `new` operator. You'll learn that instantiation is an activity that shouldn't always be done in public and can often lead to *coupling problems*. And you don't want *that*, do you? Find out how Factory Patterns can help save you from embarrassing dependencies.



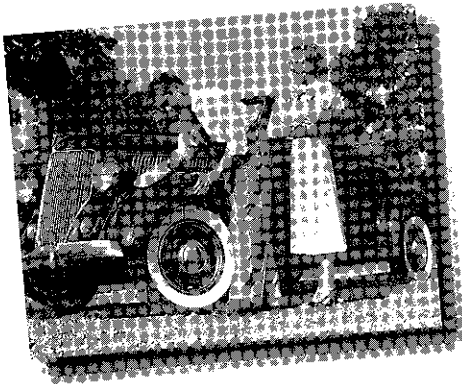
When you see “new”, think “concrete”	110
Objectville Pizza	112
Encapsulating object creation	114
Building a simple pizza factory	115
The Simple Factory defined	117
A Framework for the pizza store	120
Allowing the subclasses to decide	121
Let's make a PizzaStore	123
Declaring a factory method	125
Meet the Factory Method Pattern	131
Parallel class hierarchies	132
Factory Method Pattern defined	134
A very dependent PizzaStore	137
Looking at object dependencies	138
The Dependency Inversion Principle	139
Meanwhile, back at the PizzaStore...	144
Families of ingredients...	145
Building our ingredient factories	146
Looking at the Abstract Factory	153
Behind the scenes	154
Abstract Factory Pattern defined	156
Factory Method and Abstract Factory compared	160
Tools for your Design Toolbox	162
Exercise Solutions	164

the Singleton Pattern

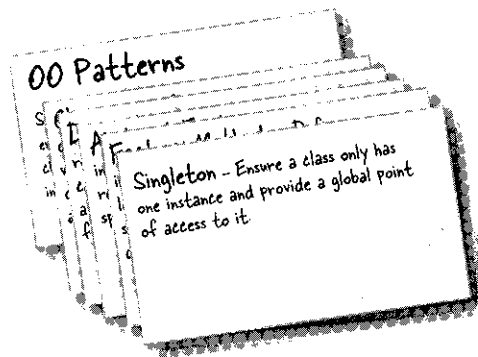
5

One of a Kind Objects

The Singleton Pattern: your ticket to creating one-of-a-kind objects, for which there is only one instance. You might be happy to know that of all patterns, the Singleton is the simplest in terms of its class diagram; in fact the diagram holds just a single class! But don't get too comfortable; despite its simplicity from a class design perspective, we'll encounter quite a few bumps and potholes in its implementation. So buckle up—this one's not as simple as it seems...



One and only one object	170
The Little Singleton	171
Dissecting the classic Singleton Pattern	173
Confessions of a Singleton	174
The Chocolate Factory	175
Singleton Pattern defined	177
Hershey, PA Houston , we have a problem...	178
BE the JVM	179
Dealing with multithreading	180
Singleton Q&A	184
Tools for your Design Toolbox	186
Exercise Solutions	188



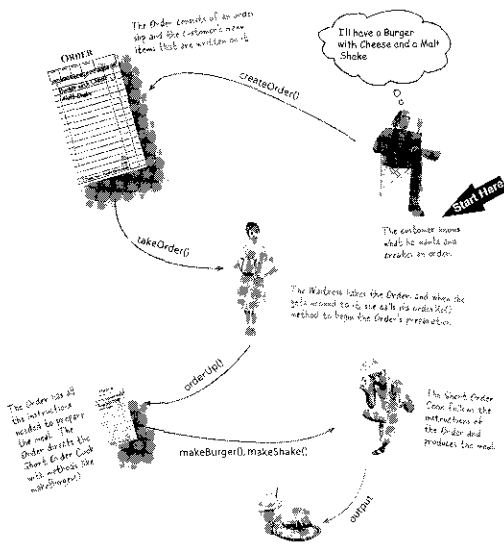
the Command Pattern

6

Encapsulating Invocation

In this chapter we take encapsulation to a whole new level: we're going to encapsulate *method invocation*.

That's right, by encapsulating invocation we can crystallize pieces of computation so that the object invoking the computation doesn't need to worry about how to do things; it just uses our crystallized method to get it done. We can also do some wickedly smart things with these encapsulated method invocations, like save them away for logging or reuse them to implement undo in our code.



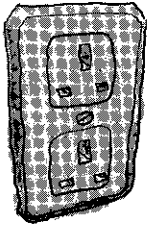
Home Automation or Bust	192
The Remote Control	193
Taking a look at the vendor classes	194
Meanwhile, back at the Diner...	197
Let's study the Diner interaction	198
The Objectville Diner Roles and Responsibilities	199
From the Diner to the Command Pattern	201
Our first command object	203
The Command Pattern defined	206
The Command Pattern and the Remote Control	208
Implementing the Remote Control	210
Putting the Remote Control through its paces	212
Time to write that documentation	215
Using state to implement Undo	220
Every remote needs a Party Mode!	224
Using a Macro Command	225
More uses of the Command Pattern: Queuing requests	228
More uses of the Command Pattern: Logging requests	229
Tools for your Design Toolbox	230
Exercise Solutions	232

the Adapter and Facade Patterns

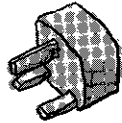
7 Being Adaptive

In this chapter we're going to attempt such impossible feats as putting a square peg in a round hole. Sound impossible? Not when we have Design Patterns. Remember the Decorator Pattern? We wrapped objects to give them new responsibilities. Now we're going to wrap some objects with a different purpose: to make their interfaces look like something they're not. Why would we do that? So we can adapt a design expecting one interface to a class that implements a different interface. That's not all, while we're at it we're going to look at another pattern that wraps objects to simplify their interface.

European Wall Outlet



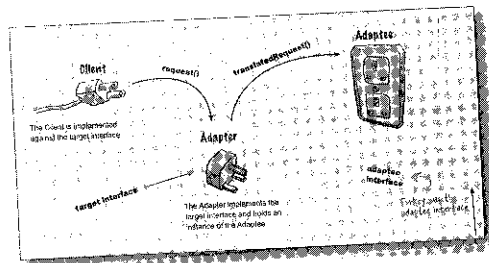
AC Power Adapter



Standard AC Plug



Adapters all around us	236
Object Oriented Adapters	237
The Adapter Pattern explained	241
Adapter Pattern defined	243
Object and Class Adapters	244
Tonight's talk: The Object Adapter and Class Adapter	247
Real World Adapters	248
Adapting an Enumeration to an Iterator	249
Tonight's talk: The Decorator Pattern and the Adapter Pattern	252
Home Sweet Home Theater	255
Lights, Camera, Facade!	258
Constructing your Home Theater Facade	261
Facade Pattern defined	264
The Principle of Least Knowledge	265
Tools for your Design Toolbox	270
Exercise Solutions	272



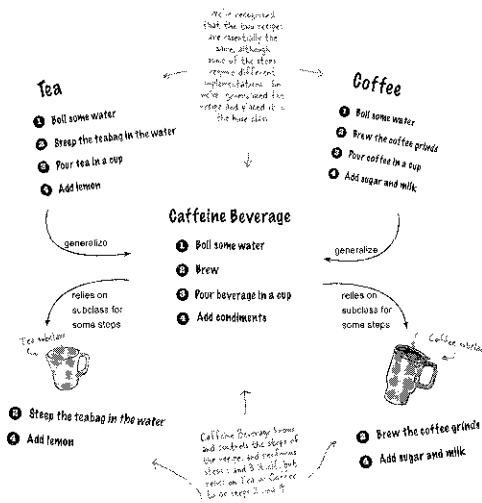
the Template Method Pattern



Encapsulating Algorithms

We've encapsulated object creation, method invocation, complex interfaces, ducks, pizzas... what could be next?

We're going to get down to encapsulating *pieces of algorithms* so that subclasses can hook themselves right into a computation anytime they want. We're even going to learn about a design principle inspired by Hollywood.



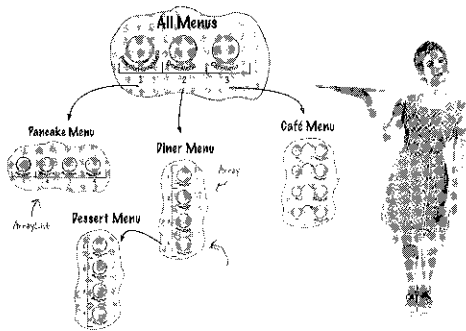
Whipping up some coffee and tea classes	277
Abstracting Coffee and Tea	280
Taking the design further	281
Abstracting prepareRecipe()	282
What have we done?	285
Meet the Template Method	286
Let's make some tea	287
What did the Template Method get us?	288
Template Method Pattern defined	289
Code up close	290
Hooked on Template Method...	292
Using the hook	293
Coffee? Tea? Nah, let's run the TestDrive	294
The Hollywood Principle	296
The Hollywood Principle and the Template Method	297
Template Methods in the Wild	299
Sorting with Template Method	300
We've got some ducks to sort	301
Comparing ducks and ducks	302
The making of the sorting duck machine	304
Swingin' with Frames	306
Applets	307
Tonight's talk: Template Method and Strategy	308
Tools for your Design Toolbox	311
Exercise Solutions	312

9

Well-Managed Collections

There are lots of ways to stuff objects into a collection.

Put them in an Array, a Stack, a List, a Map, take your pick. Each has its own advantages and tradeoffs. But when your client wants to iterate over your objects, are you going to show him your implementation? We certainly hope not! That just *wouldn't* be professional. Don't worry—in this chapter you'll see how you can let your clients *iterate* through your objects without ever seeing how you *store* your objects. You're also going to learn how to create some *super collections* of objects that can leap over some impressive data structures in a single bound. You're also going to learn a thing or two about object responsibility.



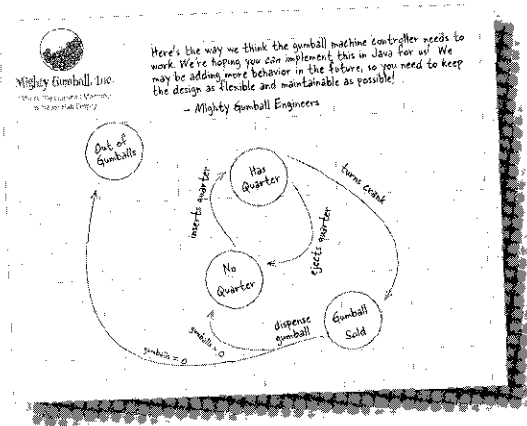
Objectville Diner and Pancake House merge	316
Comparing Menu implementations	318
Can we encapsulate the iteration?	323
Meet the Iterator Pattern	325
Adding an Iterator to DinerMenu	326
Looking at the design	331
Cleaning things up with java.util.Iterator	333
What does this get us?	335
Iterator Pattern defined	336
Single Responsibility	339
Iterators and Collections	348
Iterators and Collections in Java 5	349
Just when we thought it was safe...	353
The Composite Pattern defined	356
Designing Menus with Composite	359
Implementing the Composite Menu	362
Flashback to Iterator	368
The Null Iterator	372
The magic of Iterator & Composite together...	374
Tools for your Design Toolbox	380
Exercise Solutions	381

the State Pattern

10

The State of Things

A little known fact: the Strategy and State Patterns were twins separated at birth. As you know, the Strategy Pattern went on to create a wildly successful business around interchangeable algorithms. State, however, took the perhaps more noble path of helping objects learn to control their behavior by changing their internal state. He's often overheard telling his object clients, "just repeat after me, I'm good enough, I'm smart enough, and doggonit..."



How do we implement state?	387
State Machines 101	388
A first attempt at a state machine	390
You knew it was coming... a change request!	394
The messy STATE of things...	396
Defining the State interfaces and classes	399
Implementing our State Classes	401
Reworking the Gumball Machine	402
The State Pattern defined	410
State versus Strategy	411
State sanity check	417
We almost forgot!	420
Tools for your Design Toolbox	423
Exercise Solutions	424

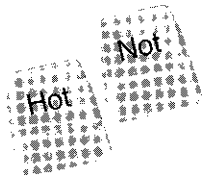


the Proxy Pattern

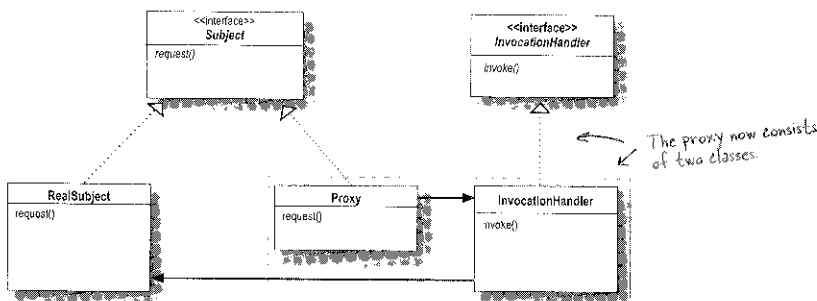
11

Controlling Object Access

Ever play good cop, bad cop? You're the good cop and you provide all your services in a nice and friendly manner, but you don't want *everyone* asking you for services, so you have the bad cop *control* access to you. That's what proxies do: control and manage access. As you're going to see there are *lots* of ways in which proxies stand in for the objects they proxy. Proxies have been known to haul entire method calls over the Internet for their proxied objects; they've also been known to patiently stand in the place for some pretty lazy objects.



Monitoring the gumball machines	430
The role of the 'remote proxy'	434
RMI detour	437
GumballMachine remote proxy	450
Remote proxy behind the scenes	458
The Proxy Pattern defined	460
Get Ready for virtual proxy	462
Designing the CD cover virtual proxy	464
Virtual proxy behind the scenes	470
Using the Java API's proxy	474
Five minute drama: protecting subjects	478
Creating a dynamic proxy	479
The Proxy Zoo	488
Tools for your Design Toolbox	491
Exercise Solutions	492

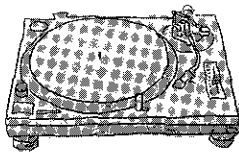
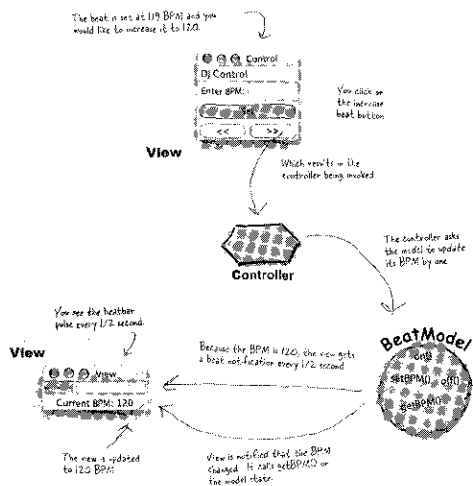


Compound Patterns

12

Patterns of Patterns

Who would have ever guessed that Patterns could work together? You've already witnessed the acrimonious Fireside Chats (and be thankful you didn't have to see the *Pattern Death Match* pages that the publisher forced us to remove from the book so we could avoid having to use a Parent's Advisory warning label), so who would have thought patterns can actually get along well together? Believe it or not, some of the most powerful OO designs use several patterns together. Get ready to take your pattern skills to the next level; it's time for Compound Patterns. Just be careful—your co-workers might kill you if you're struck with Pattern Fever.



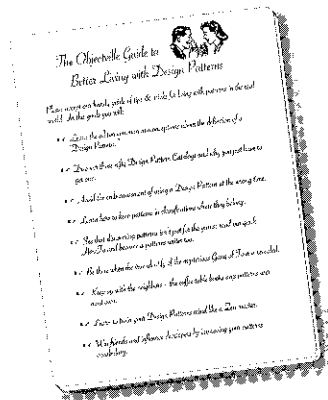
Compound Patterns	500
Duck reunion	501
Adding an adapter	504
Adding a decorator	506
Adding a factory	508
Adding a composite, and iterator	513
Adding an observer	516
Patterns summary	523
A duck's eye view: the class diagram	524
Model-View-Controller, the song	526
Design Patterns are your key to the MVC	528
Looking at MVC through patterns-colored glasses	532
Using MVC to control the beat...	534
The Model	537
The View	539
The Controller	542
Exploring strategy	545
Adapting the model	546
Now we're ready for a HeartController	547
MVC and the Web	549
Design Patterns and Model 2	557
Tools for your Design Toolbox	560
Exercise Solutions	561

Better Living with Patterns

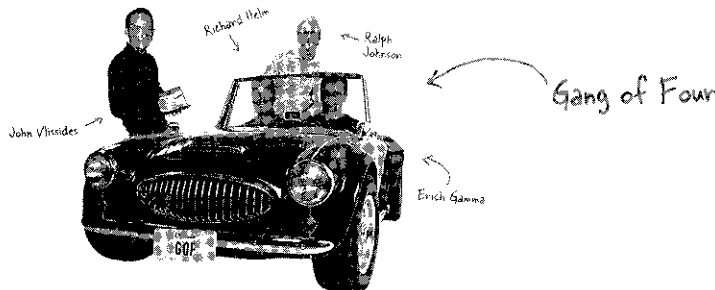
13

Patterns in the Real World

Ahhhh, now you're ready for a bright new world filled with Design Patterns. But, before you go opening all those new doors of opportunity we need to cover a few details that you'll encounter out in the real world—things get a little more complex *out there* than they are here in Objectville. Come along, we've got a nice guide to help you through the transition...

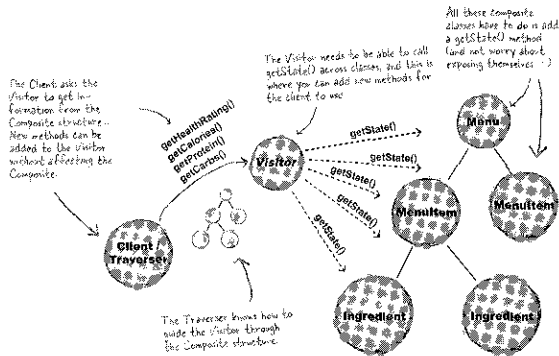


Your Objectville guide	578
Design Pattern defined	579
Looking more closely at the Design Pattern definition	581
May the force be with you	582
Pattern catalogs	583
How to create patterns	586
So you wanna be a Design Patterns writer?	587
Organizing Design Patterns	589
Thinking in patterns	594
Your mind on patterns	597
Don't forget the power of the shared vocabulary	599
Top five ways to share your vocabulary	600
Cruisin' Objectville with the Gang of Four	601
Your journey has just begun...	602
Other Design Pattern resources	603
The Patterns Zoo	604
Annihilating evil with Anti-Patterns	606
Tools for your Design Toolbox	608
Leaving Objectville...	609



14 Appendix: Leftover Patterns

Not everyone can be the most popular. A lot has changed in the last 10 years. Since *Design Patterns: Elements of Reusable Object-Oriented Software* first came out, developers have applied these patterns thousands of times. The patterns we summarize in this appendix are full-fledged, card-carrying, official GoF patterns, but aren't always used as often as the patterns we've explored so far. But these patterns are awesome in their own right, and if your situation calls for them, you should apply them with your head held high. Our goal in this appendix is to give you a high level idea of what these patterns are all about.



Bridge	612
Builder	614
Chain of Responsibility	616
Flyweight	618
Interpreter	620
Mediator	622
Memento	624
Prototype	626
Visitor	628



Index