Your Brain on Design Patterns

Head First Design Patterns

Avoid those embarrassing coupling mistakes

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

Discover the secrets of the Patterns Guru

Load the patterns that matter straight into your brain

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

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

Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

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 tres '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 co-author of Design Patterns

"'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 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 the Freemans."

— 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 the Freemans have out run him. Seriously...this is one of the funniest and smartest books on software design I've ever read."

— Aaron LaBerge, VP Technology, ESPN.com

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, co-editor 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 and Elisabeth Freeman 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 X-Box, 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

"Just the right tone for the geeked-out, casual-cool guru coder in all of us. The right reference for practical development strategies—gets my brain going without having to slog through a bunch of tired, stale professor-speak."

— Travis Kalanick, Founder of Scour and Red Swoosh Member of the MIT TR100

"This book combines good humors, 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 the *Head First* approach

"Java technology is everywhere—in mobile phones, cars, cameras, printers, games, PDAs, ATMs, smart cards, gas pumps, sports stadiums, medical devices, Web cams, servers, you name it. If you develop software and haven't learned Java, it's definitely time to dive in—Head First."

— Scott McNealy, Sun Microsystems Chairman, President and CEO

"It's fast, irreverent, fun, and engaging. Be careful—you might actually learn something!"

— Ken Arnold, former Senior Engineer at Sun Microsystems
Co-author (with James Gosling, creator of Java),
"The Java Programming Language"

"Head First Java is like Monty Python meets the gang of four... the text is broken up so well by puzzles and stories, quizzes and examples, that you cover ground like no computer book before."

— Douglas Rowe, Columbia Java Users Group

" 'Head First Java'... gives new meaning to their marketing phrase `There's an O Reilly for that.` I picked this up because several others I respect had described it in terms like 'revolutionary' and a described a radically different approach to the textbook. They were (are) right... In typical O'Reilly fashion, they've taken a scientific and well considered approach. The result is funny, irreverent, topical, interactive, and brilliant...Reading this book is like sitting in the speakers lounge at a view conference, learning from – and laughing with – peers... If you want to UNDERSTAND Java, go buy this book."

— Andrew Pollack, www.thenorth.com

"If you want to *learn* Java, look no further: welcome to the first GUI-based technical book! This perfectly-executed, ground-breaking format delivers benefits other Java texts simply can't... Prepare yourself for a truly remarkable ride through Java land."

— Neil R. Bauman, Captain & CEO, Geek Cruises (www.GeekCruises.com)

What a fantastic way to learn!!! I CAN NOT PUT THIS BOOK DOWN!!! My 3 year old woke up at 1:40 a.m. this morning, and I put him back to bed with book in hand and a flashlight so I could continue to read for about another hour.

— Ross Goldberg

"This stuff is so fricking good it makes me wanna WEEP! I'm stunned."

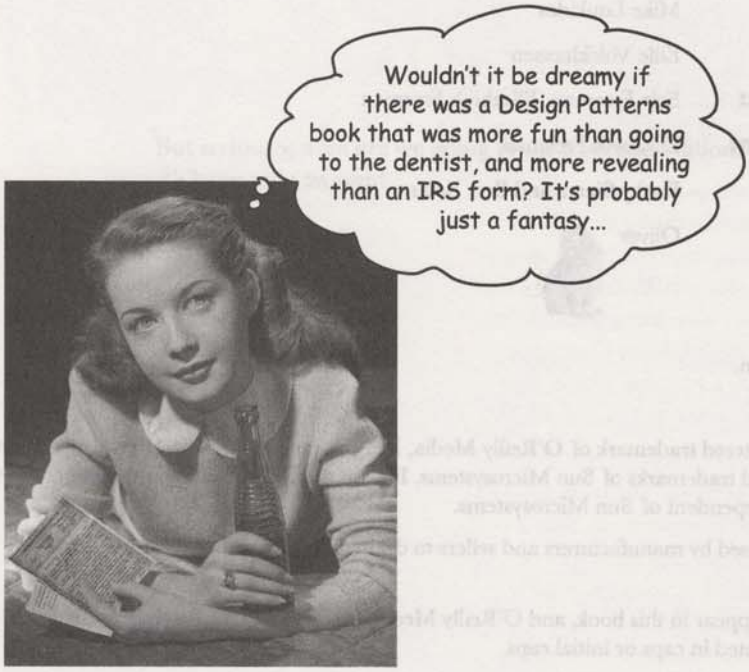
— Floyd Jones, Senior Technical Writer/Poolboy, BEA

Other related books from O'Reilly

Head First Java
Head First EJB
Head First Servlets & JSP
Learning Java
Java in a Nutshell
Java Enterprise in a Nutshell
Java Examples in a Nutshell
Java Cookbook
J2EE Design Patterns

Be watching for more books in the Head First series!

Head First Design Patterns



Eric Freeman
Elisabeth Freeman

with
Kathy Sierra
Bert Bates

O'REILLY®

Beijing • Cambridge • Köln • Paris • Sebastopol • Taipei • Tokyo

Head First Design Patterns

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

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

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 (safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor: Mike Loukides

Cover Designer: Ellie Volckhausen

Pattern Wranglers: Eric Freeman, Elisabeth Freeman

Facade Decoration: Elisabeth Freeman

Strategy: Kathy Sierra and Bert Bates

Observer: Oliver

Printing History:

October 2004: First Edition.

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.

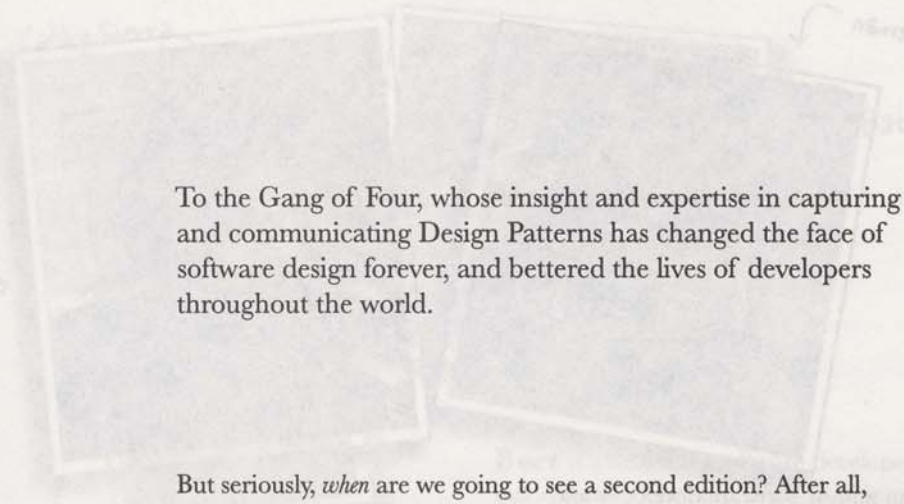
In other words, if you use anything in *Head First Design Patterns* to, say, run a nuclear power plant, you're on your own. We do, however, encourage you to use the DJ View app.

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: 0-596-00712-4

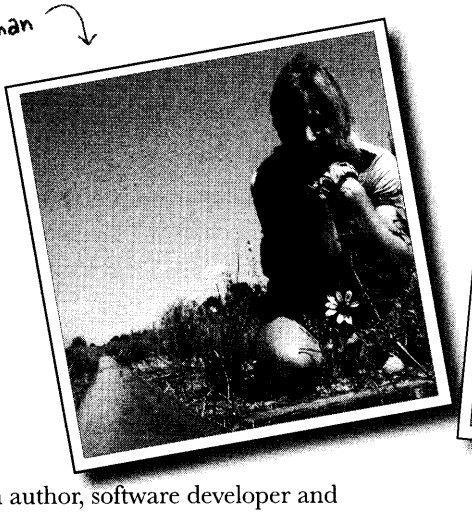
[M]

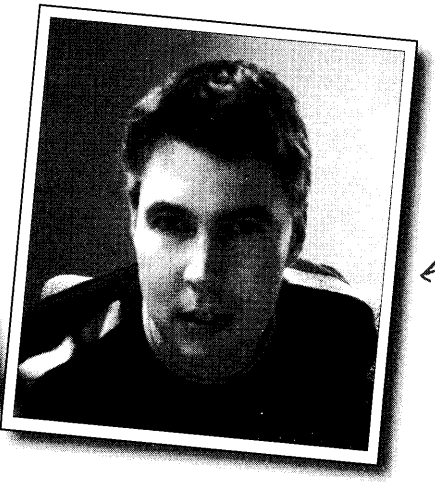


To the Gang of Four, whose insight and expertise in capturing and communicating Design Patterns has changed the face of software design forever, and bettered the lives of developers throughout the world.

But seriously, *when* are we going to see a second edition? After all, it's been only *ten years*!

Authors/Developers of Head First Design Patterns





Elisabeth is an author, software developer and digital artist. She's been involved with the Internet since the early days, having co-founded The Ada Project (TAP), an award winning web site for women in computing now adopted by the ACM. More recently Elisabeth lead research and development efforts in digital media at the Walt Disney Company where she co-invented Motion, a content system that delivers terabytes of video every day to Disney, ESPN and Movies.com users.

Elisabeth is a computer scientist at heart and holds graduate degrees in Computer Science from Yale University and Indiana University. She's worked in a variety of areas including visual languages, RSS syndication and Internet systems. She's also been an active advocate for women in computing, developing programs that encourage woman to enter the field. These days you'll find her sipping some Java or Cocoa on her Mac, although she dreams of a day when the whole world is using Scheme.

Elisabeth has loved hiking and the outdoors since her days growing up in Scotland. When she's outdoors her camera is never far. She's also an avid cyclist, vegetarian and animal lover.

You can send her email at beth@wickedlysmart.com

Eric is a computer scientist with a passion for media and software architectures. He just wrapped up four years at a dream job – directing Internet broadband and wireless efforts at Disney – and is now back to writing, creating cool software and hacking Java and Macs.

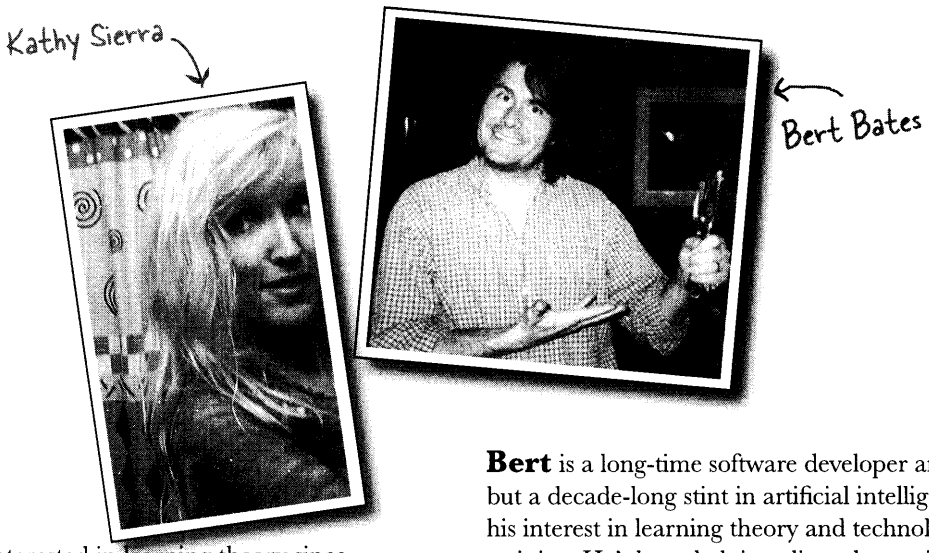
Eric spent a lot of the '90s working on alternatives to the desktop metaphor with David Gelernter (and they're both *still* asking the question "why do I have to give a file a name?"). Based on this work, Eric landed a Ph.D. at Yale University in '97. He also co-founded Mirror Worlds Technologies (now acquired) to create a commercial version of his thesis work, Lifestreams.

In a previous life, Eric built software for networks and supercomputers. You might know him from such books as *JavaSpaces Principles Patterns and Practice*. Eric has fond memories of implementing tuple-space systems on Thinking Machine CM-5s and creating some of the first Internet information systems for NASA in the late 80s.

Eric is currently living in the high desert near Santa Fe. When he's not writing text or code you'll find him spending more time tweaking than watching his home theater and trying to restoring a circa 1980s Dragon's Lair video game. He also wouldn't mind moonlighting as an electronica DJ.

Write to him at eric@wickedlysmart.com or visit his blog at http://www.ericfreeman.com

Creators of the Head First series
(and co-conspirators on this book)



Kathy has been interested in learning theory since her days as a game designer (she wrote games for Virgin, MGM, and Amblin'). She developed much of the Head First format while teaching New Media Authoring for UCLA Extension's Entertainment Studies program. More recently, she's been a master trainer for Sun Microsystems, teaching Sun's Java instructors how to teach the latest Java technologies, and developing several of Sun's certification exams. Together with Bert Bates, she has been actively using the Head First concepts to teach throusands of developers. Kathy is the founder of javaranch.com, which won a 2003 and 2004 Software Development magazine Jolt Cola Productivity Award. You might catch her teaching Java on the Java Jam Geek Cruise (geekcruises.com).

She recently moved from California to Colorado, where she's had to learn new words like, "ice scraper" and "fleece", but the lightning there is fantastic.

Likes: runing, skiing, skateboarding, playing with her Icelandic horse, and weird science. Dislikes: entropy.

You can find her on javaranch, or occasionally blogging on java.net. Write to her at kathy@wickedlysmart.com.

Bert is a long-time software developer and architect, but a decade-long stint in artificial intelligence drove his interest in learning theory and technology-based training. He's been helping clients becoming better programmers ever since. Recently, he's been heading up the development team for several of Sun's Java Certification exams.

He spent the first decade of his software career travelling the world to help broadcast clients like Radio New Zealand, the Weather Channel, and the Arts & Entertainment Network (A & E). One of his all-time favorite projects was building a full rail system simulation for Union Pacific Railroad.

Bert is a long-time, hopelessly addicted *go* player, and has been working on a *go* program for way too long. He's a fair guitar player and is now trying his hand at banjo.

Look for him on javaranch, on the IGS go server, or you can write to him at terrapin@wickedlysmart.com.

Table of Contents (summary)

Table of Contents (the real thing)

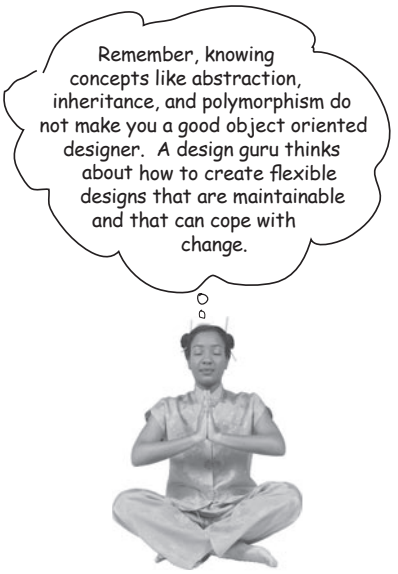
Intro

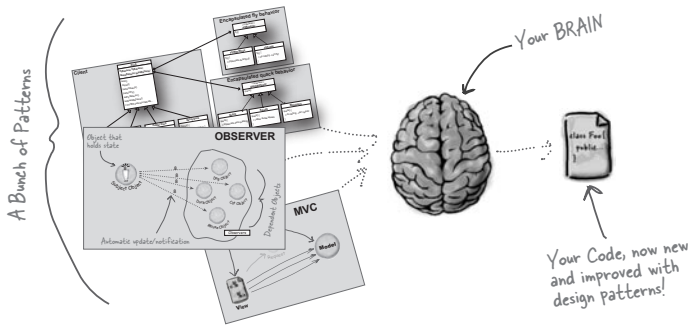
Your brain on Design Patterns. Here *you* are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is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

intro to Design Patterns

1 Welcome to Design Patterns

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



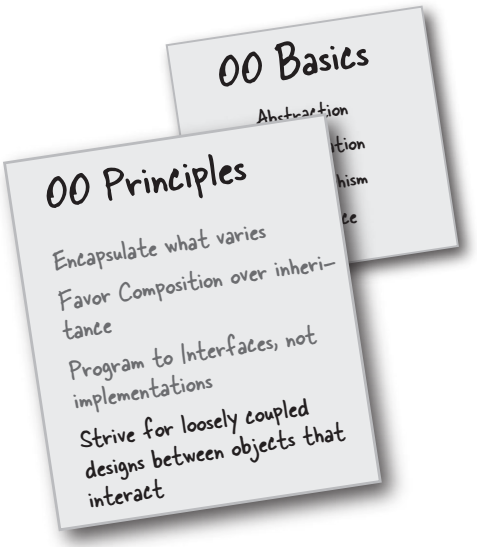


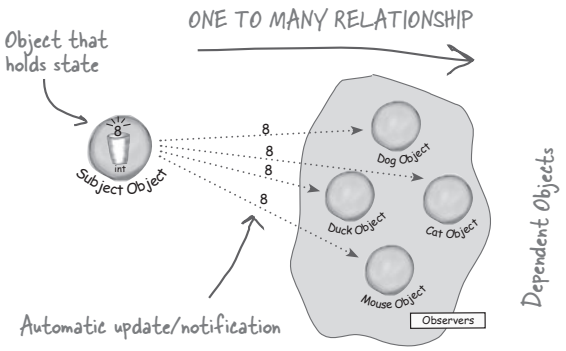
the Observer Pattern

2 Keeping your Objects in the Know

Don't miss out when something interesting happens!

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

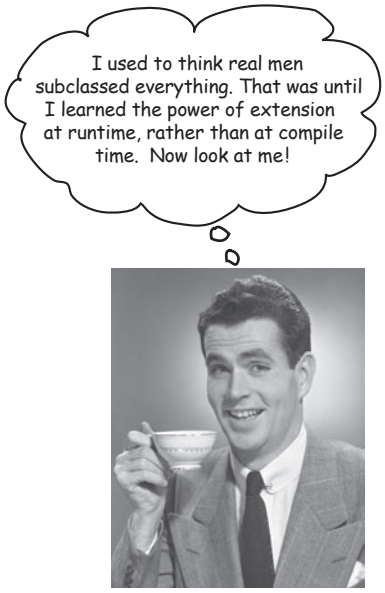




the Decorator Pattern

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.*

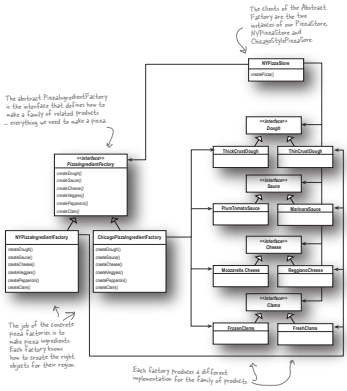


the Factory Pattern

4 Baking with OO Goodness

Get ready to cook some loosely coupled OO designs.

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



the Singleton Pattern

5 One of a Kind Objects

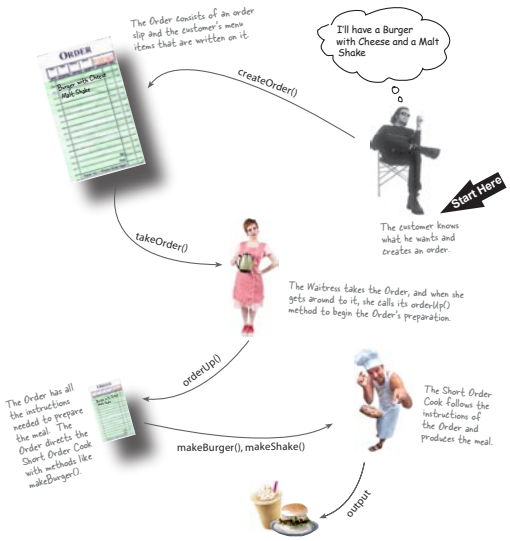
The Singleton Pattern: your ticket to creating one-of-a-kind objects, for which there is only one instance. You might be happy to know that of all patterns, the Singleton is the simplest in terms of its class diagram; in fact the diagram holds just a single class! But don't get too comfortable; despite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

the Command Pattern

6 Encapsulating Invocation

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

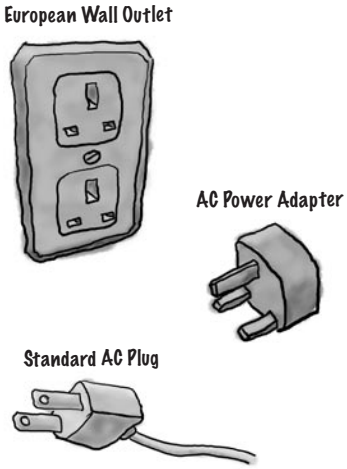
That's right, by encapsulating invocation we can crystallize pieces of computation so that the object invoking the computation doesn't need to worry about how to do things; it just uses our crystallized method to get it done. We can also do some wickedly smart things with these encapsulated method invocations,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

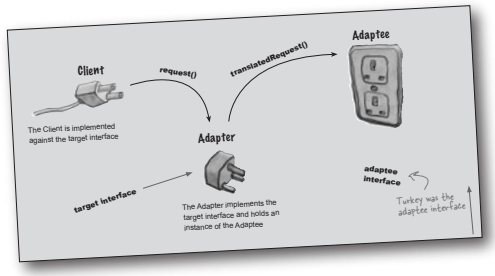


the Adapter and Facade Patterns

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.





the Template Method Pattern

8 Encapsulating Algorithms

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

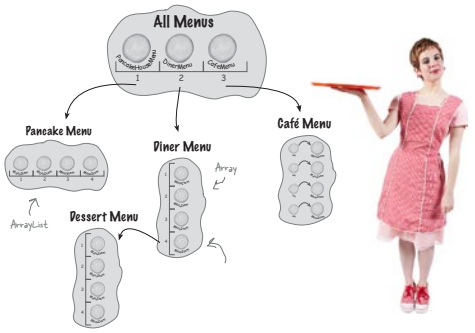
We're going to get down to encapsulating *pieces of algorithms* so that subclasses can hook themselves right into a computation anytime they want. We're even going to learn about a design principle inspired by Hollywood.

the Iterator and Composite Patterns

9 Well-Managed Collections

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

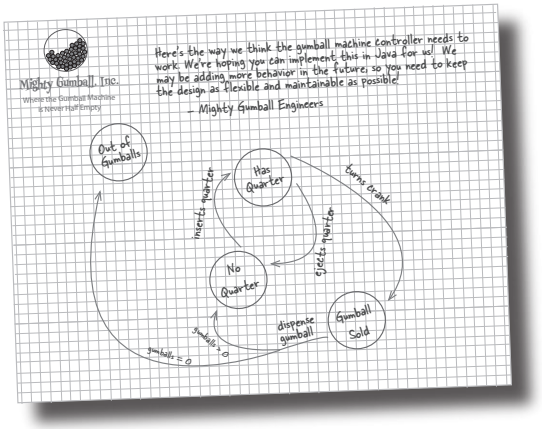
Put them in an Array, a Stack, a List, a Map, take your pick. Each has its own advantages and tradeoffs. But when your client wants to iterate over your objects, are you going to show him your implementation? We certainly hope not! That just *wouldn't* be professional. Don't worry—in this chapter you'll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



the State Pattern

10 The State of Things

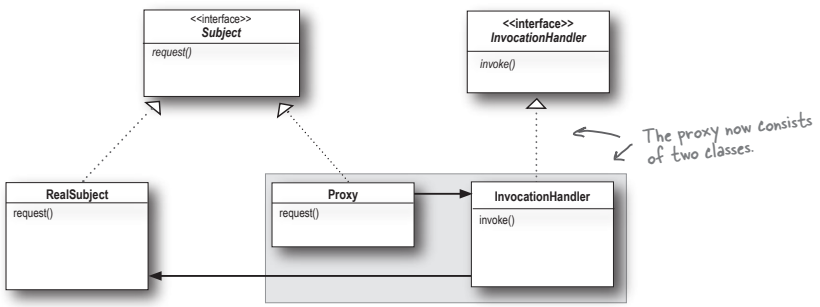
A little known fact: the Strategy and State Patterns were twins separated at birth. As you know, the Strategy Pattern went on to create a wildly successful business around interchangeable algorithms. State, however, took the perhaps more noble path of helping objects learn to control their behavior by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



the Proxy Pattern

11 Controlling Object Access

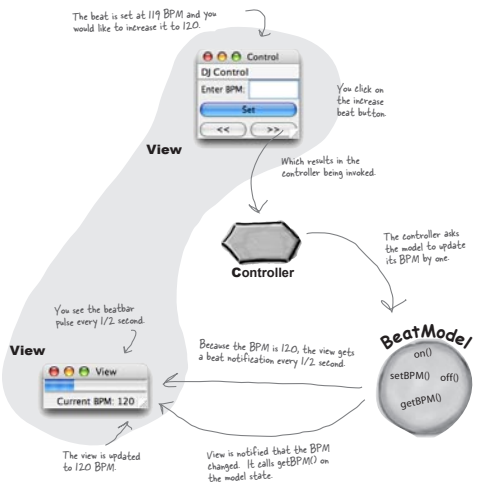
Ever play good cop, bad cop? You're the good cop and you provide all your services in a nice and friendly manner, but you don't want *everyone* asking you for services, so you have the bad cop *control access* to you. That's what proxies do: control and manage access. As you're going to see there are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



Compound Patterns

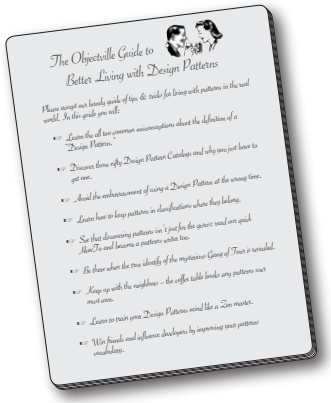
12 Patterns of Patterns

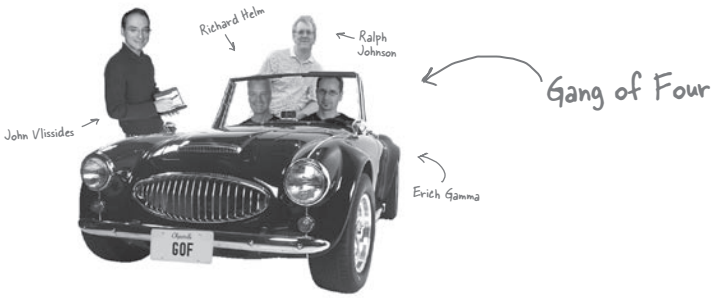
Who would have ever guessed that Patterns could work together? You've already witnessed the acrimonious Fireside Chats (and be thankful you didn't have to see the Pattern Death Match pages that the publisher forced us to remove from the book so we could avoid having to use a Parent's Advisory warning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



13 Patterns in the Real World

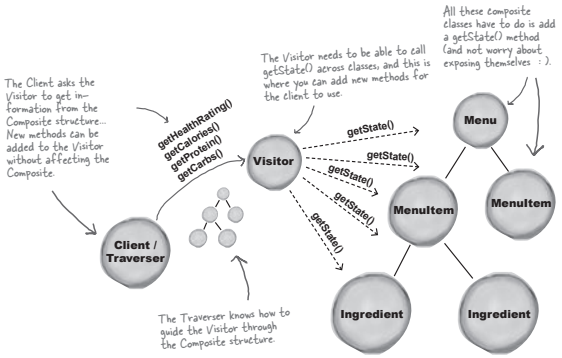
Ahhhh, now you're ready for a bright new world filled with Design Patterns. But, before you go opening all those new doors of opportunity we need to cover a few details that you'll encounter out in the real world—things get a little more complex *out there* than they are here in Objectville. Come along,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





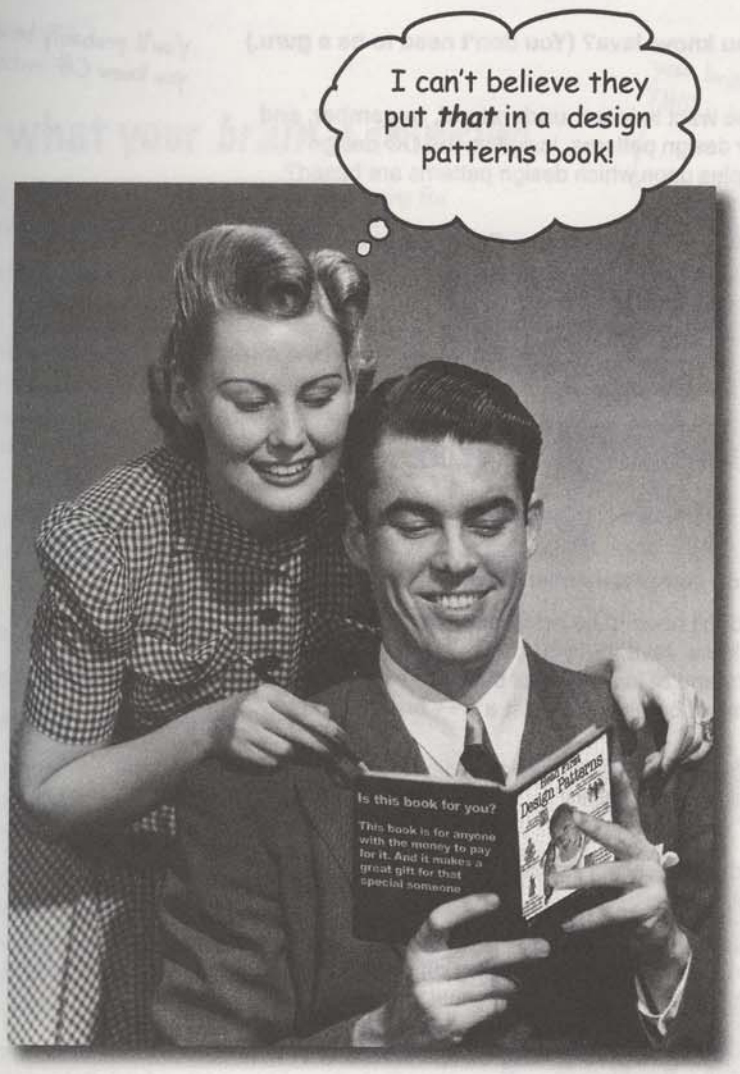
14 Appendix: Leftover Patterns

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



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.)**

   You'll probably be okay if you know C# instead.

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.

Who should probably back away from this book?

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

1. **Are you completely new to Java?**

   (You don't need to be advanced, and even if you *don't* know Java, but you know C#, you'll probably understand at least 80% of the code examples. You also *might* be okay with just a C++ background.)

2. Are you a kick-butt OO 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 Java components 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.*

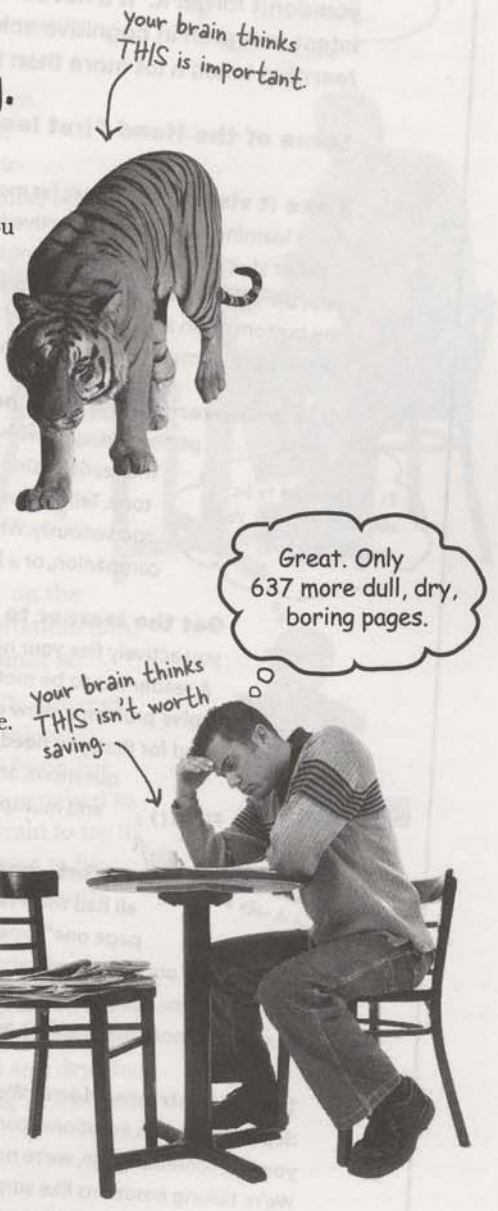
And that's how your brain knows...

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.

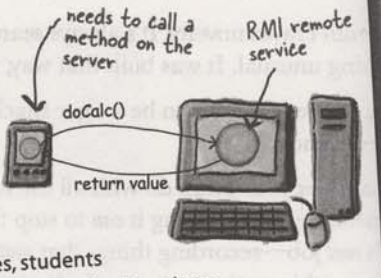
And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."



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

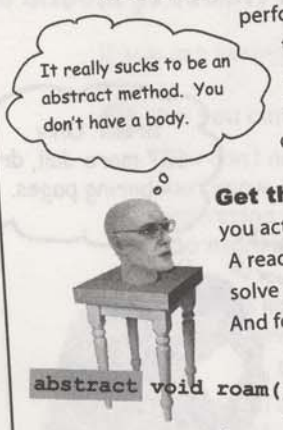
So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know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

Some of the Head First learning principles:

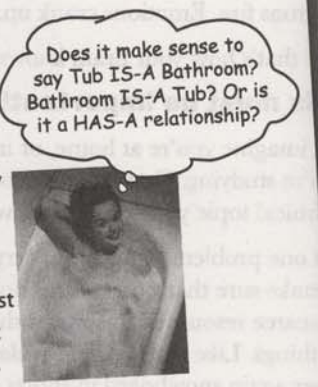


Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another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

Use a conversational and personalized style. In recent 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?



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, and thought-provoking questions, and activities that involve both sides of the brain, and multiple senses.



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.

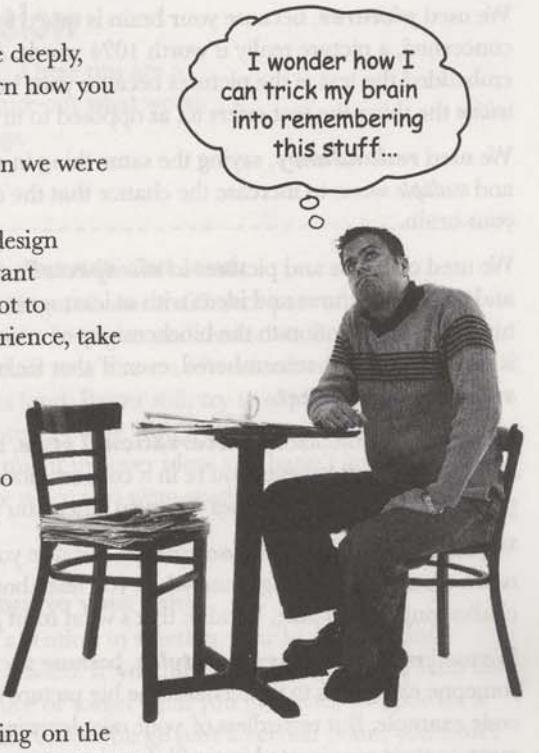
Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you *care* about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, 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."

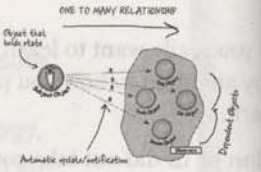
The faster way is to do ***anything that increases brain activity,*** especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting 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 1024 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 the text refers to, as opposed to in a caption or buried in the text somewhere.



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 40 ***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.

BULLET POINTS

We include 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 judgements.

We included ***challenges***, 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*.

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.

cut this out and stick it on your refridgerator.

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 side-bars—***they're part of the core content!*** Don't skip them.

5 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.

6 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.

7 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.

8 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.

9 *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.

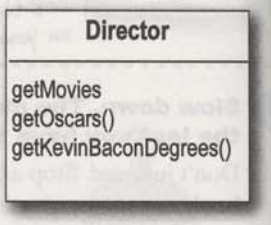
10 *Design* something!

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 faux-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), Sun's J2EE patterns, JSP 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 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 pattern 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 J2SE 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://www.wickedlysmart.com/headfirstdesignpatterns/code.html`

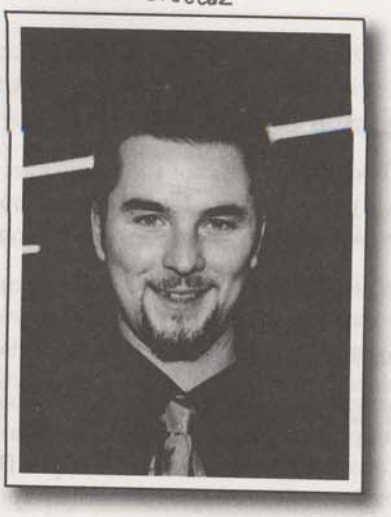
Also, for the sake of focusing on the learning side of the code, we did not put our classes into packages (in other words, they're all in the Java default package). We don't recommend this in the real world, and when you download the code examples from this book, you'll find that all classes *are* in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

The 'Brain Power' exercises don't have answers.

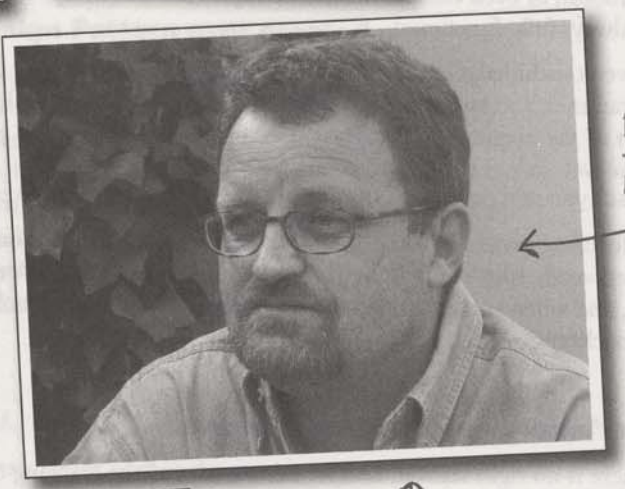
For some of them, there is no right answer, and for others, part of the learning experience of the Brain Power activities is for you to decide if and when your answers are right. In some of the Brain Power exercises you will find hints to point you in the right direction.

Tech Reviewers

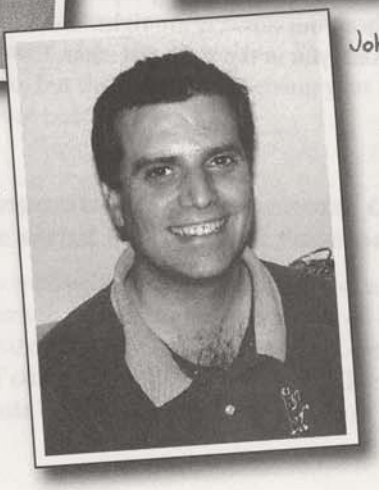

Jef Cumps


Valentin Crettaz


Barney Marispini


Fearless leader of the HFDP Extreme Review Team.


Ike Van Atta


Mark Spritzler

Johannes deJong


Dirk Schreckmann


Jason Menard

Philippe Maquet

In memory of Philippe Maquet

1960-2004

Your amazing technical expertise, relentless enthusiasm, and deep concern for the learner will inspire us always.

We will never forget you.

Acknowledgments

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**, to rock and roll star **Ellie Volkhausen** for her inspired cover design and also to **Colleen Gorman** for her hardcore copyedit. Finally, thanks to **Mike Hendrickson** for championing this Design Patterns 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.

Even more people*

From Eric and Elisabeth

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. Thanks to both of you for letting us enter your amazing world; we hope we've done Head First justice. Seriously, this has been amazing. Thanks for all your careful guidance, for pushing us to go forward and most of all, for trusting us (with your baby). You're both certainly "wickedly smart" and you're also the hippest 29 year olds we know. So... what's next?

A big thank you to **Mike Loukides** and **Mike Hendrickson**. 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.

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 & 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 indespensible resource for us in writing this book.

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. And thanks to Apple's **James Dempsey** for allowing us to use his *MVC song*.

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.

From Kathy and Bert

We'd like to thank Mike Hendrickson for finding Eric and Elisabeth... but we can't. Because of these two, we discovered (to our horror) that we aren't the *only* ones who can do a Head First book. ;) However, if readers want to *believe* that it's really Kathy and Bert who did the cool things in the book, well, who are *we* to set them straight?

*The large number of acknowledgments is because we're testing the theory that everyone mentioned in a book acknowledgment will buy at least one copy, probably more, what with relatives and everything. If you'd like to be in the acknowledgment of our *next* book, and you have a large family, write to us.

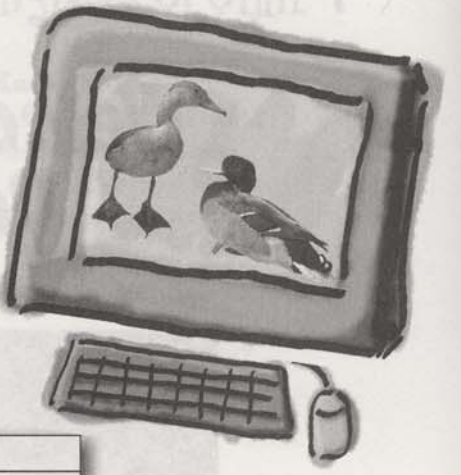
1 Intro to Design Patterns

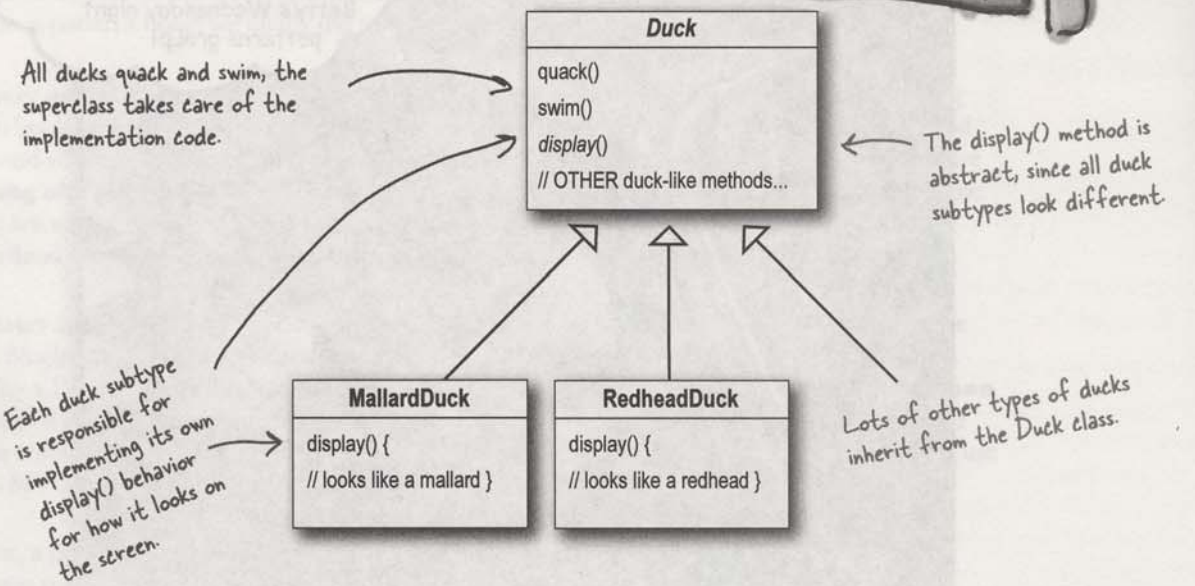
Welcome to Design Patterns

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 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.

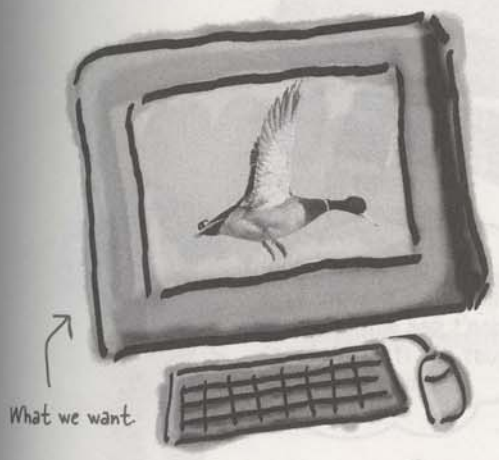


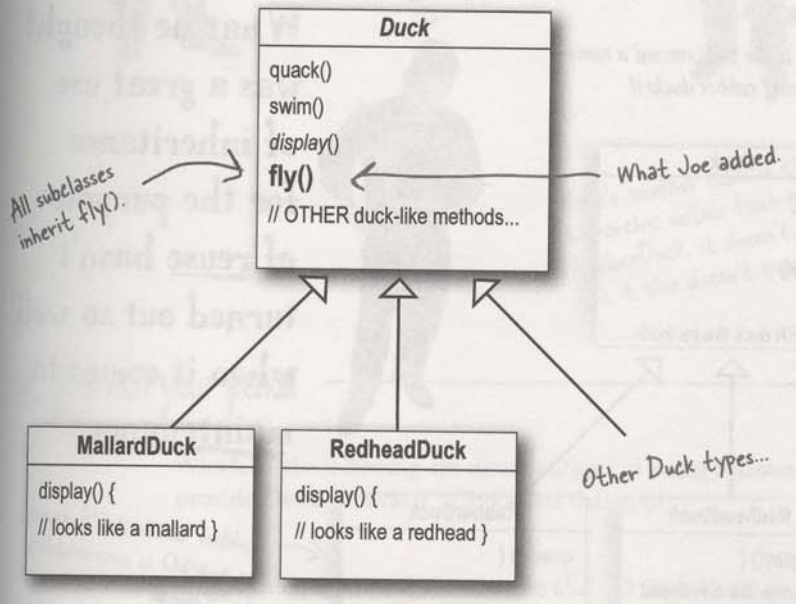


In the last year, the company has been under increasing pressure from competitors. After a week long off-site brainstorming session over golf, the company executives think it's time for a big innovation. They need something *really* impressive to show at the upcoming shareholders meeting in Maui *next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

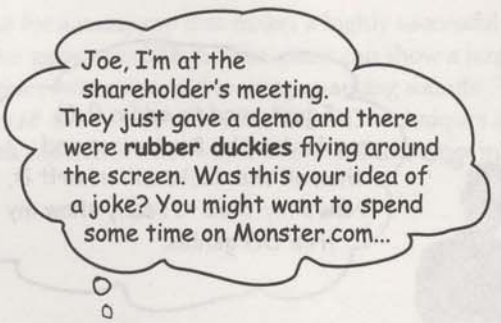
But now we need the ducks to FLY

The executives decided that flying ducks is just what the simulator needs to blow away the other duck sim 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...



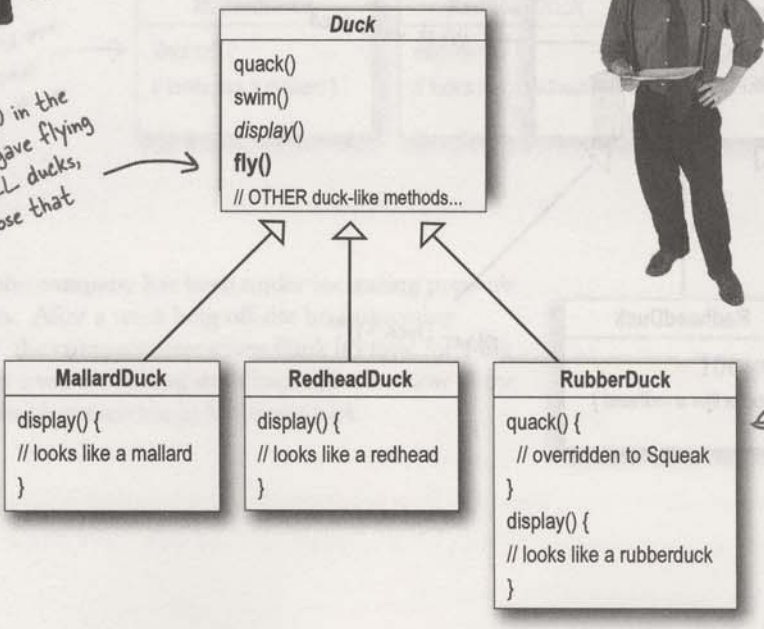
What happened?

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)!

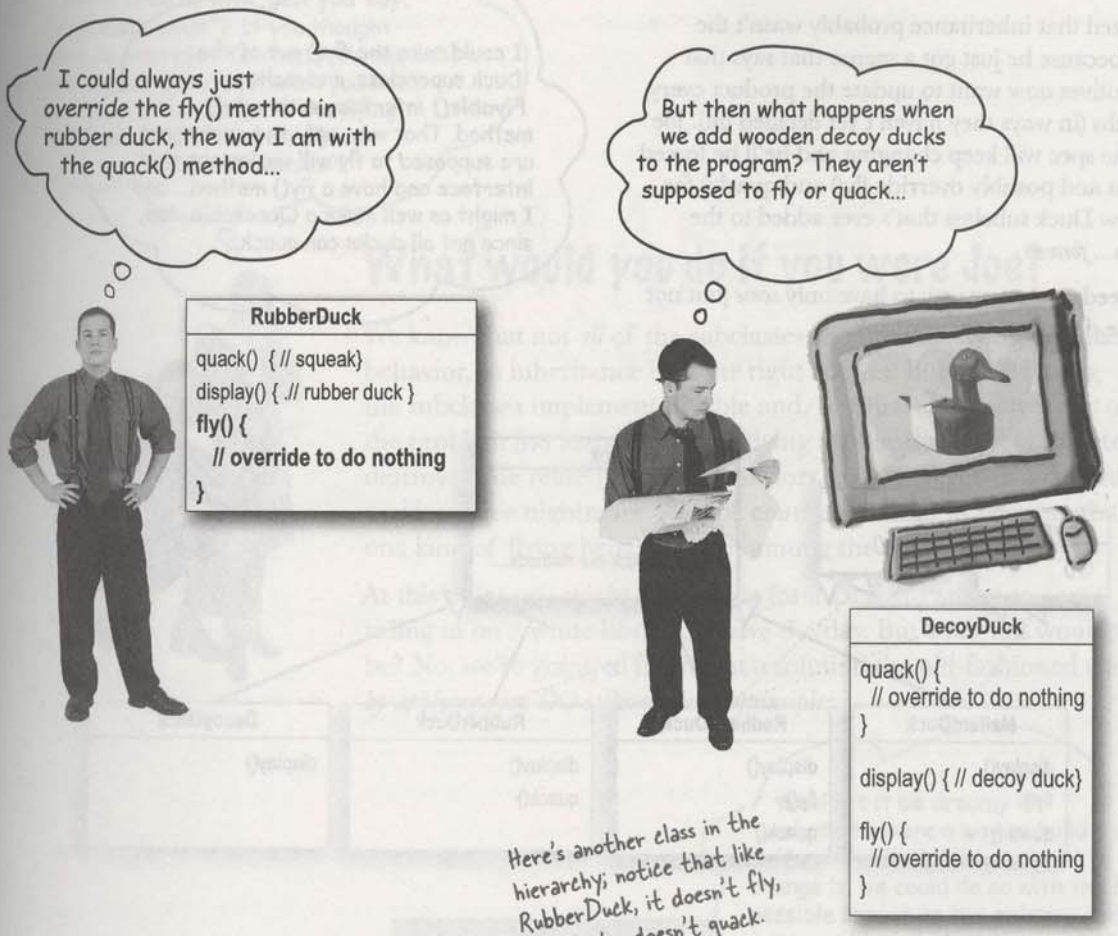
What he thought was a great use of inheritance for the purpose of reuse hasn't turned out so well when it comes to maintenance.

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



Rubber ducks don't quack, so quack() is overrridden to "Squeak".

Joe thinks about inheritance...



Sharpen your pencil

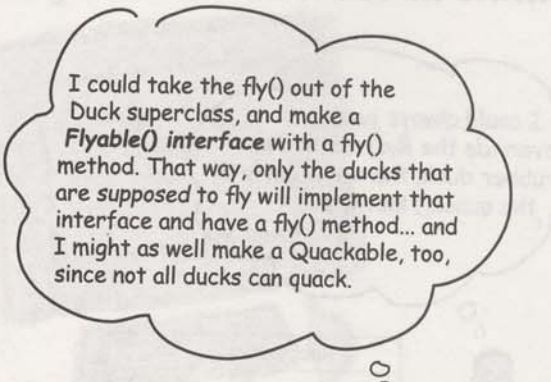
Which of the following are disadvantages of using *inheritance* to provide Duck behavior? (Choose all that apply.)

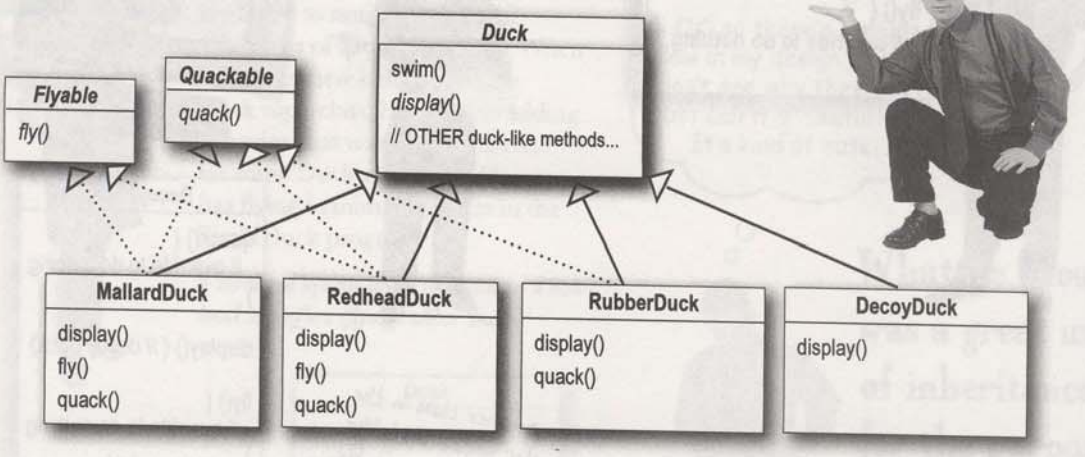
- ❑ A. Code is duplicated across subclasses.
- ❑ B. Runtime behavior changes are difficult.
- ❑ C. We can't make ducks dance.
- ❑ D. Hard to gain knowledge of all duck behaviors.
- ❑ E. Ducks can't fly and quack at the same time.
- ❑ F. Changes can unintentionally affect other ducks.

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.





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 only 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?

(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).

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 have no implementation code, so no code reuse. And that means that whenever you need to modify a behavior, you're 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.

Our first of many design principles. We'll spend more time on these thruoghout 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!

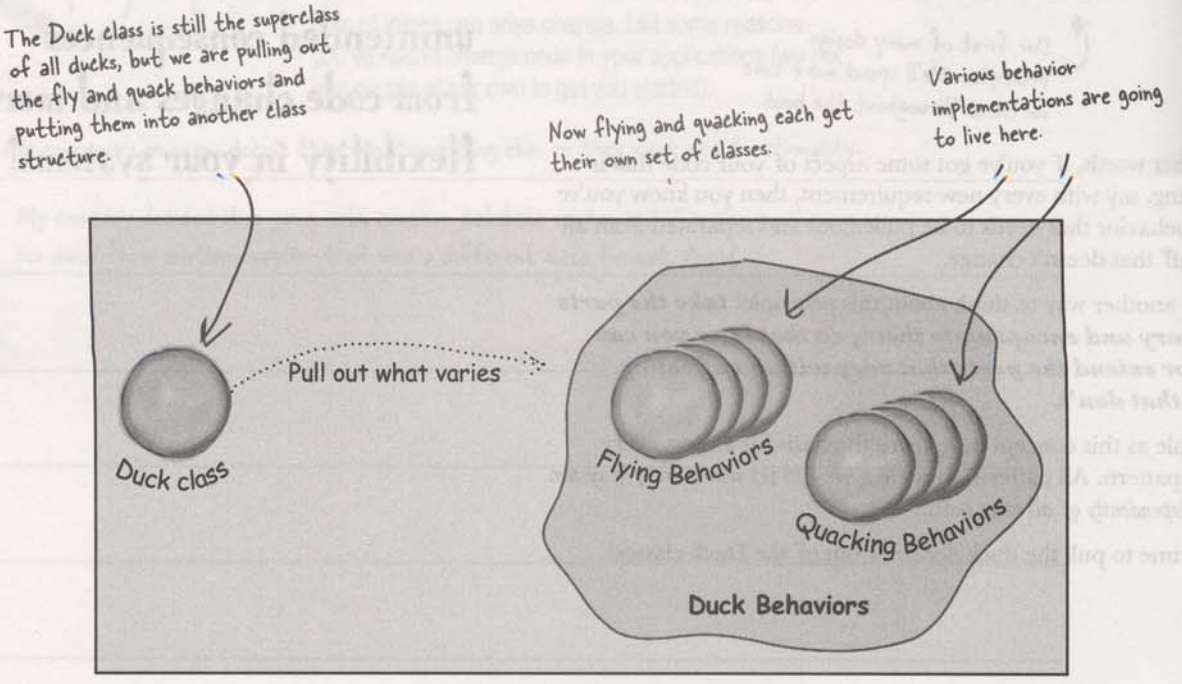
Separating what changes from what stays the same

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 their 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, say, *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.

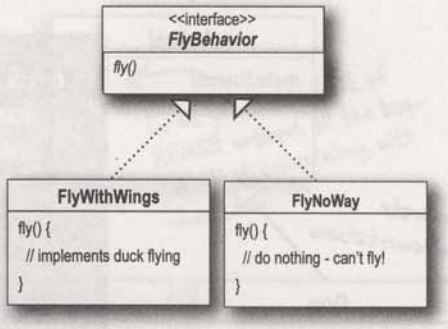
So this time it won't be the *Duck* classes that will implement the flying and quacking interfaces. Instead, we'll make a set of classes whose entire reason for living is to represent a behavior (for example, "squeaking"), and it's the *behavior* class, rather than the Duck class, that will implement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

This is in contrast to the way we were doing things before, where a behavior either came 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 out 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.



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

The word *interface* is overloaded here. There's the *concept* of interface, but there's also the Java construct **`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:

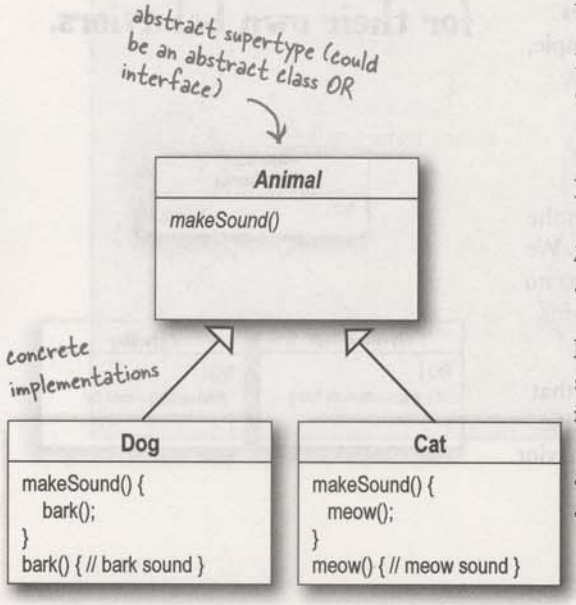
```
Animal animal = new Dog();
animal.makeSound();
```

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

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

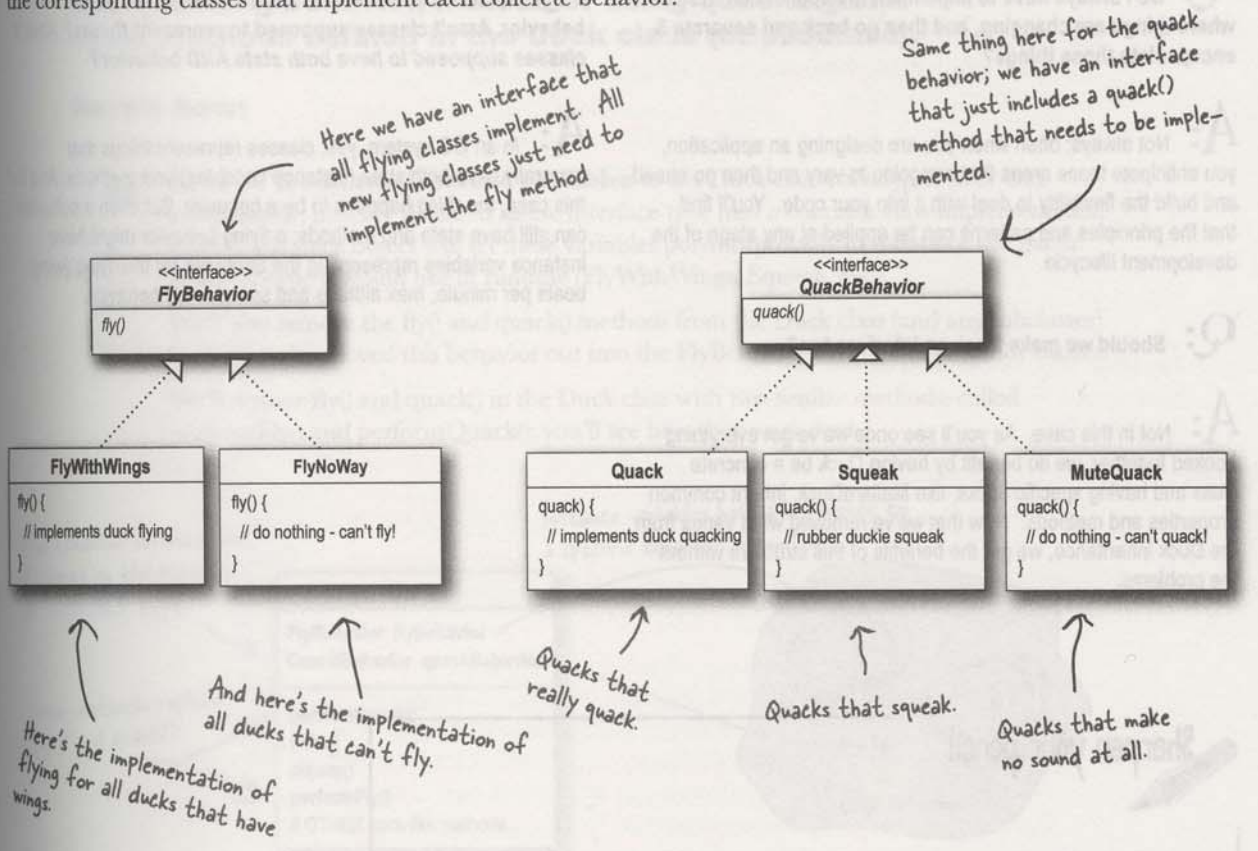
```
a = getAnimal();
animal.makeSound();
```

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:



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 and separate & 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 be a concrete class 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 and 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 Behavior

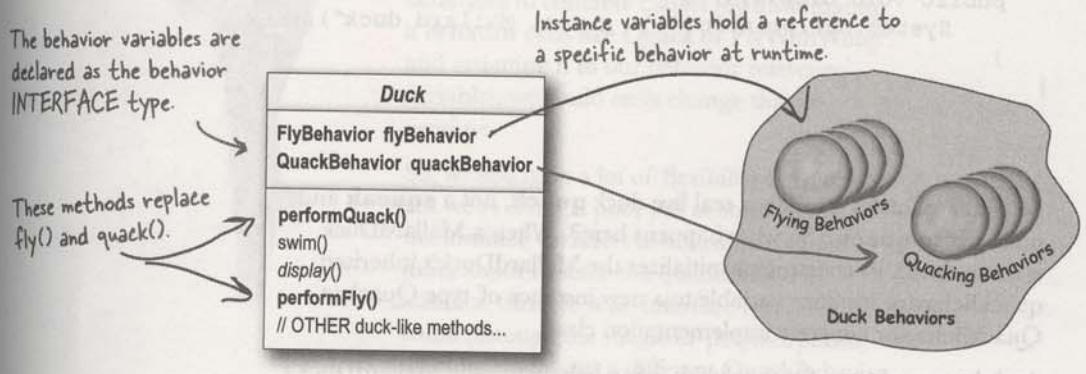
The key is that a Duck will now *delegate* its flying and quacking behavior, instead of using quacking and flying methods defined in the Duck class (or subclass).

Here's how:

1. **First we'll add two instance variables** to the Duck class called *flyBehavior* and *quackBehavior*, that are declared as the interface type (not a concrete class implementation type). Each duck object will set these variables polymorphically to reference the *specific* behavior type it would like at runtime (FlyWithWings, Squeak, etc.).

   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 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 allows 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 it 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.

So MallardDuck's quack is a real live duck **quack**, not a **squeak** and not a **mute quack**. So what happens here? 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 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, but 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 Sqeak.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!

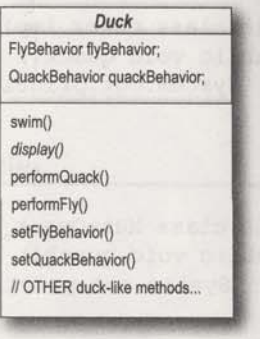
```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```

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 subclass, 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();
      }
}
```

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...voila! 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.

5 Run it!

```
File Edit Window Help Yabadabadoo
%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.

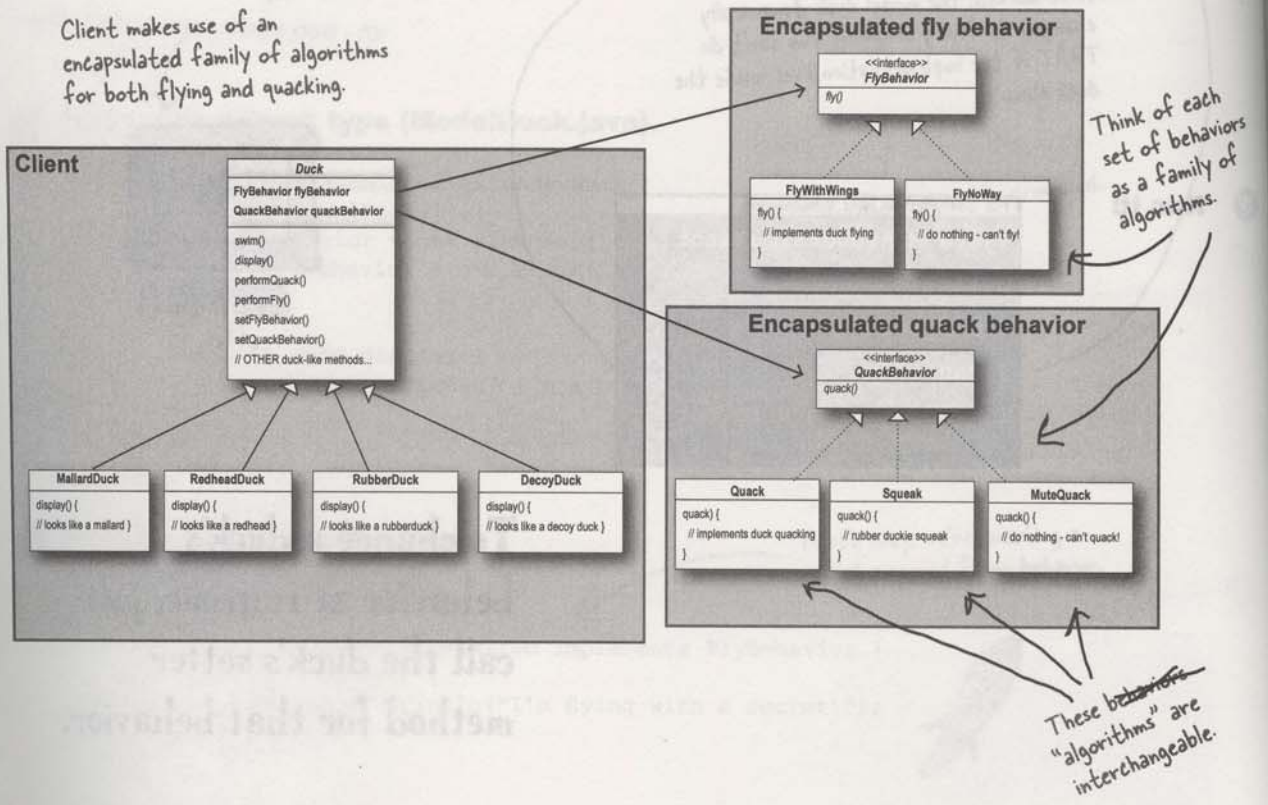
The Big Picture on encapsulated behaviors

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.



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.

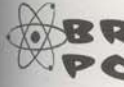
This is an important technique; in fact, we've been using our third design principle:

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.



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?

Master and Student...

Master: *Grasshopper, tell me what you have learned of the Object-Oriented ways.*

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

Master: *Grasshopper, continue...*

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

Master: *Grasshopper, is more time spent on code* ***before*** *or* ***after*** *development is complete?*

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

Master: *So Grasshopper, should effort go into reuse* ***above*** *maintaintability and extensibility?*

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

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

Now that we've made you take the long road to apply 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.

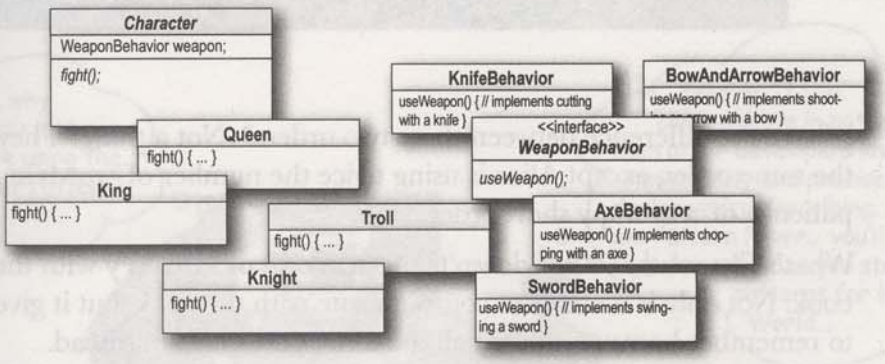
Design Puzzle

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

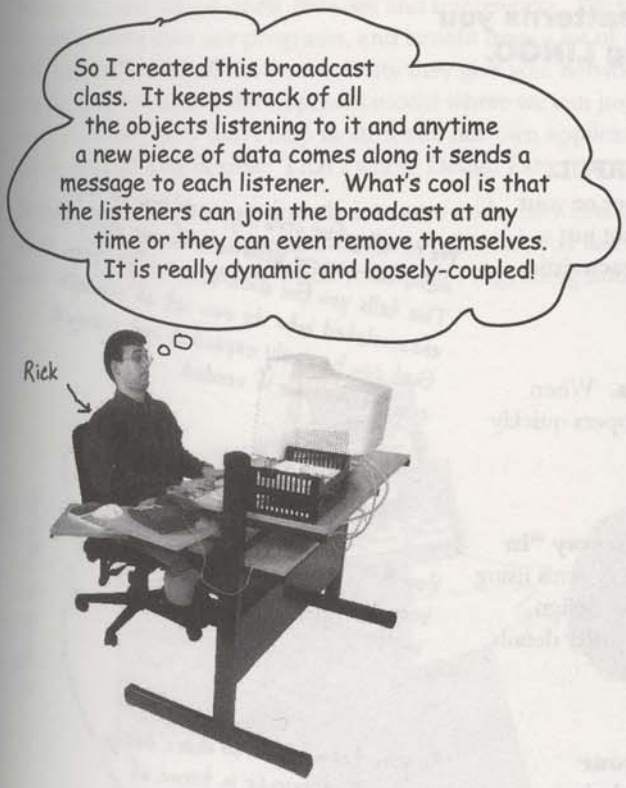
Give me a C.J. White, a black & white, a Jack Benny, a radio, a house boat, a coffee regular and burn one!

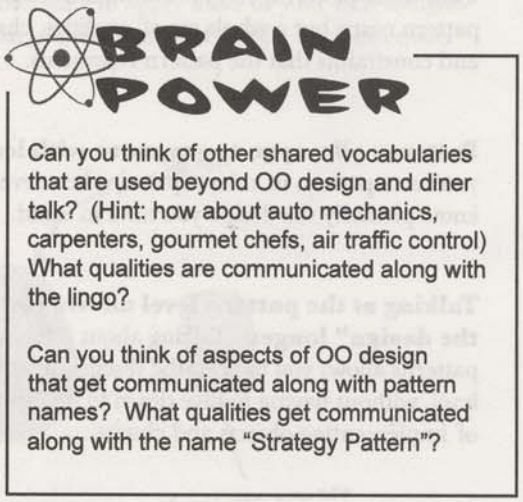
What's the difference between these two orders? Not a thing! They're both the same order, except Alice is using twice the number of words and trying the patience of a grumpy short order cook.

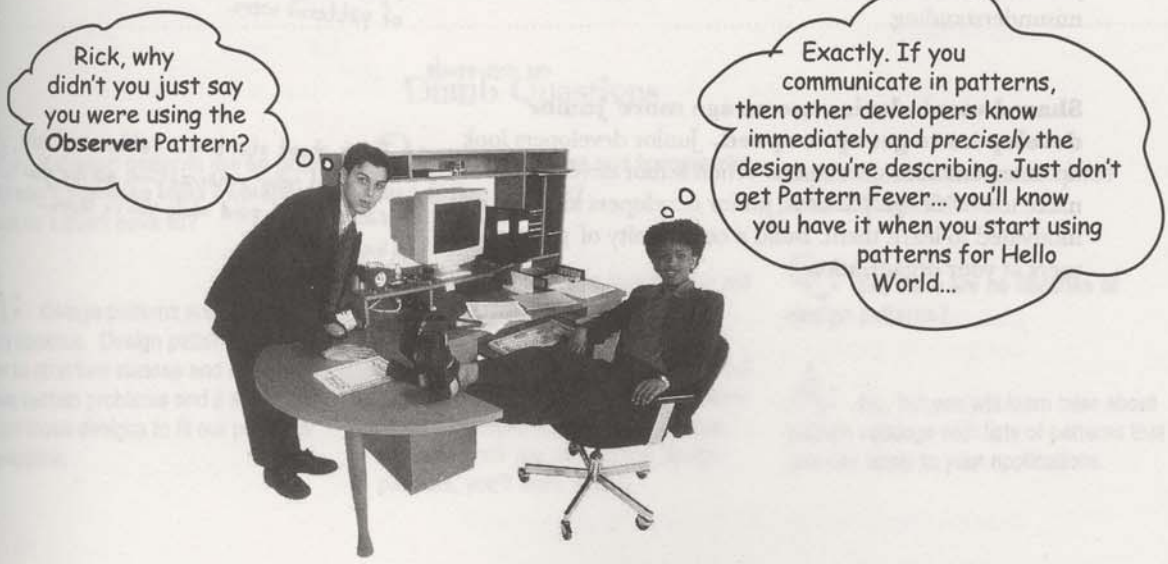
What's Flo got that Alice doesn't? **A shared vocabulary** with the short order cook. Not only is it easier to communicate with the cook, but it gives the cook less to remember because he's got all the diner patterns in his head.

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...







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 patterns users.

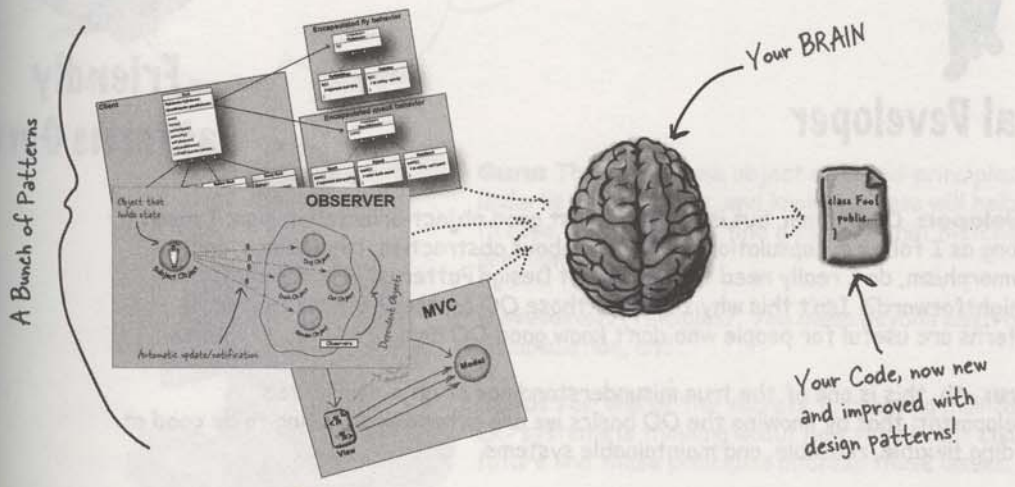
Shared vocabularies encourage more junior developers to get up to speed. Junior developers look up to experienced developers. When senior developers make use of design patterns, junior developers also become motivated to learn them. Build a community of pattern users at your organization.

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 towards 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 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 of jungle spaghetti code.



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?

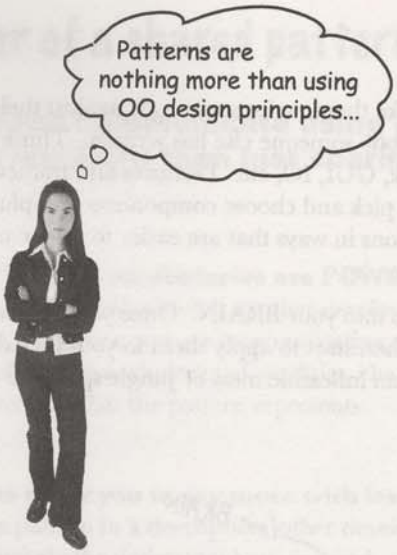
A: Design patterns are higher level than libraries. Design patterns tell us how to structure classes and objects to solve certain problems and it is our job to adapt those designs to fit our particular application.

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

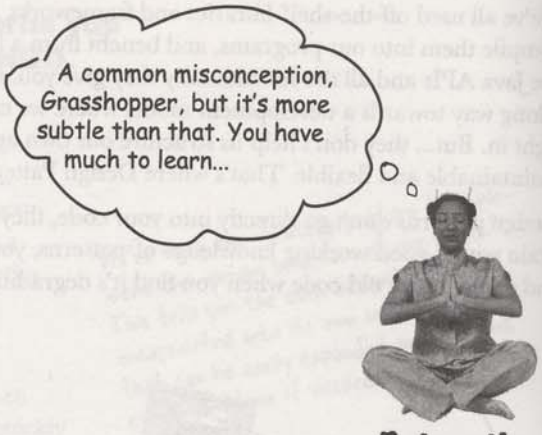
A: Frameworks and libraries are not design patterns; they provide specific implementations that we link into our code. Sometimes, however, libraries and frameworks make use of design patterns in their implementations. That's great, because once you understand design patterns, you'll more quickly understand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

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

A: No, but you will learn later about pattern 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 do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.

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

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

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 of using classes polymorphically, how inheritance is like design by contract, and how encapsulation works. If you are a little rusty on these, pull out your Head First Java and review, then skim this chapter again.

OO Principles

Encapsulate what varies.

Favor composition over inheritence.

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

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.

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

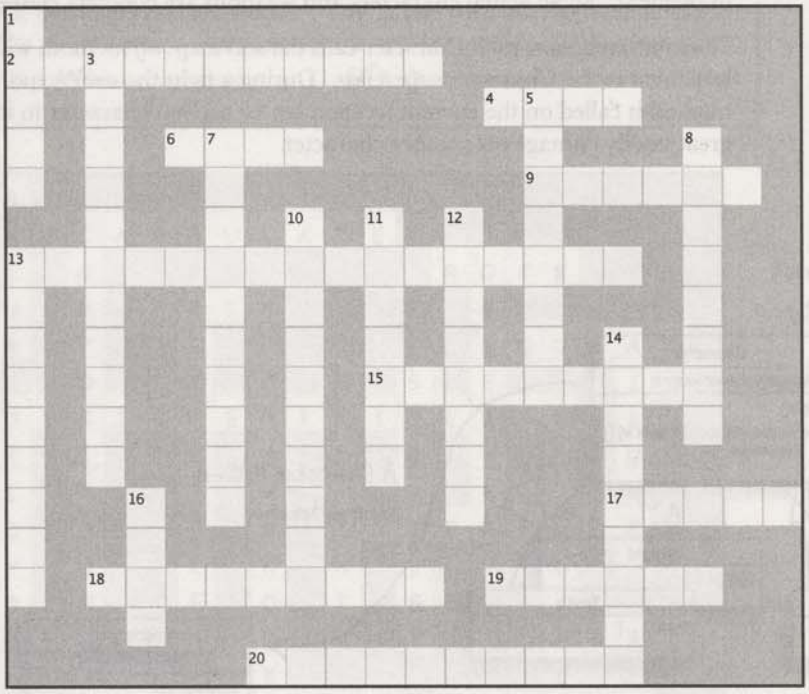
One down, many to go!

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.

Let's give your right brain something to do.

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



Across

2. _____ what varies
4. Design patterns _____
6. Java IO, Networking, Sound
9. Rubberducks make a
13. Bartender thought they were called
15. Program to this, not an implementation
17. Patterns go into your _____
18. Learn from the other guy's
19. Development constant
20. Patterns give us a shared _____

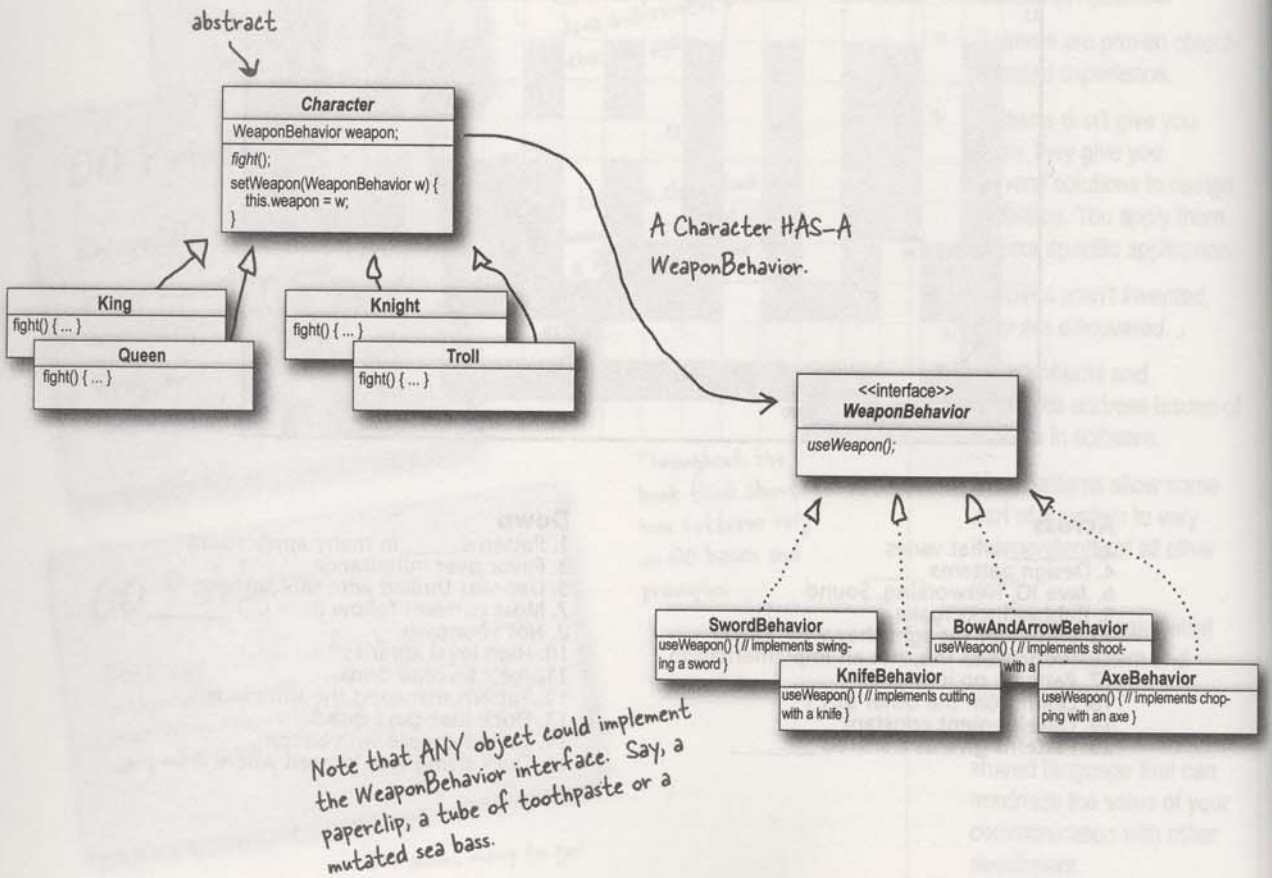
Down

1. Patterns ____ in many applications
3. Favor over inheritance
5. Dan was thrilled with this pattern
7. Most patterns follow from OO _____
8. Not your own
10. High level libraries
11. Joe's favorite drink
12. Pattern that fixed the simulator
13. Duck that can't quack
14. Grilled cheese with bacon
16. Duck demo was located where

Design Puzzle Solution

Character is the abstract class for all the other characters (King, Queen, Knight and Troll) while Weapon is an interface that all weapons 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.

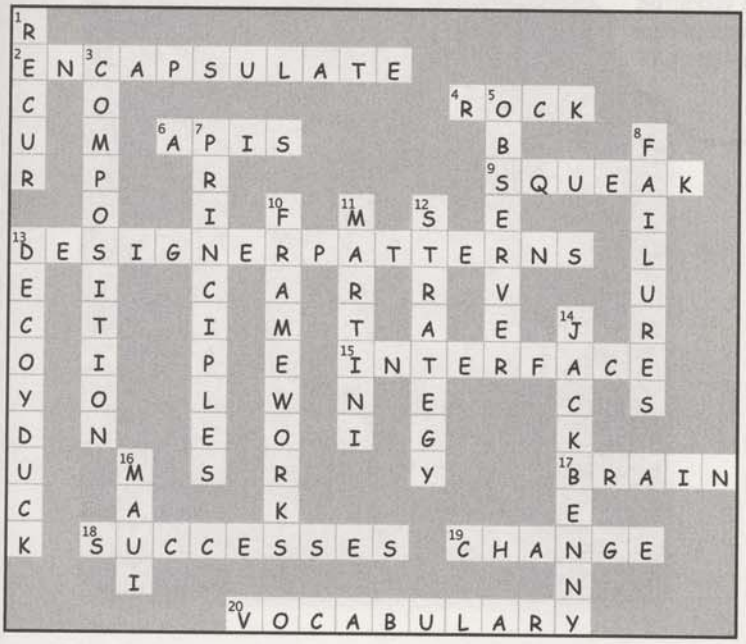


Solutions

Sharpen your pencil

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

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



Sharpen your pencil

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?

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

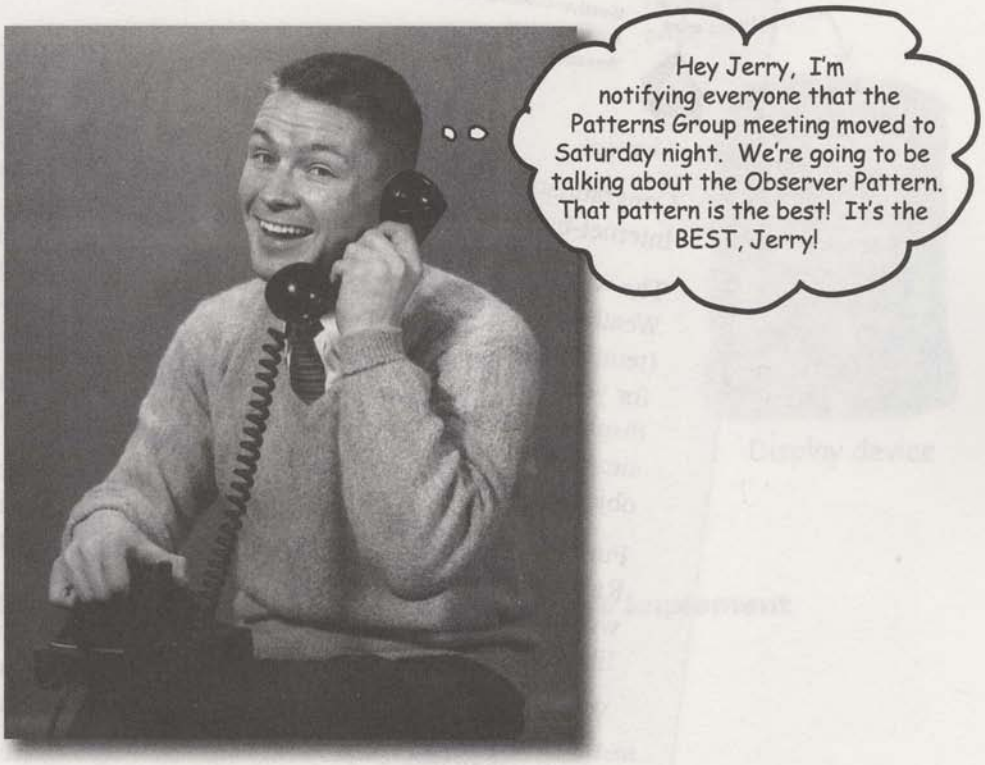
My company decided it is going with another database vendor and it is also purchasing its data from another supplier that uses a different data format. Argh!

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.

2 the Observer Pattern

Keeping your Objects in the know



Don't miss out when something interesting happens! We've got a pattern that keeps your objects in the know when something they might care about happens. Objects can even decide at runtime whether they want to be kept informed. The Observer Pattern is one of the most heavily used patterns in the JDK, and it's incredibly useful. Before we're done, we'll also look at one to many relationships and loose coupling (yeah, that's right, we said coupling). With Observer, you'll be the life of the Patterns Party.

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 patent pending WeatherData object, which tracks current weather conditions (temperature, humidity, and barometric pressure). We'd like for 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 release an API so that other developers can write their own weather displays and plug them right in. We'd like for you to supply that API!

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,

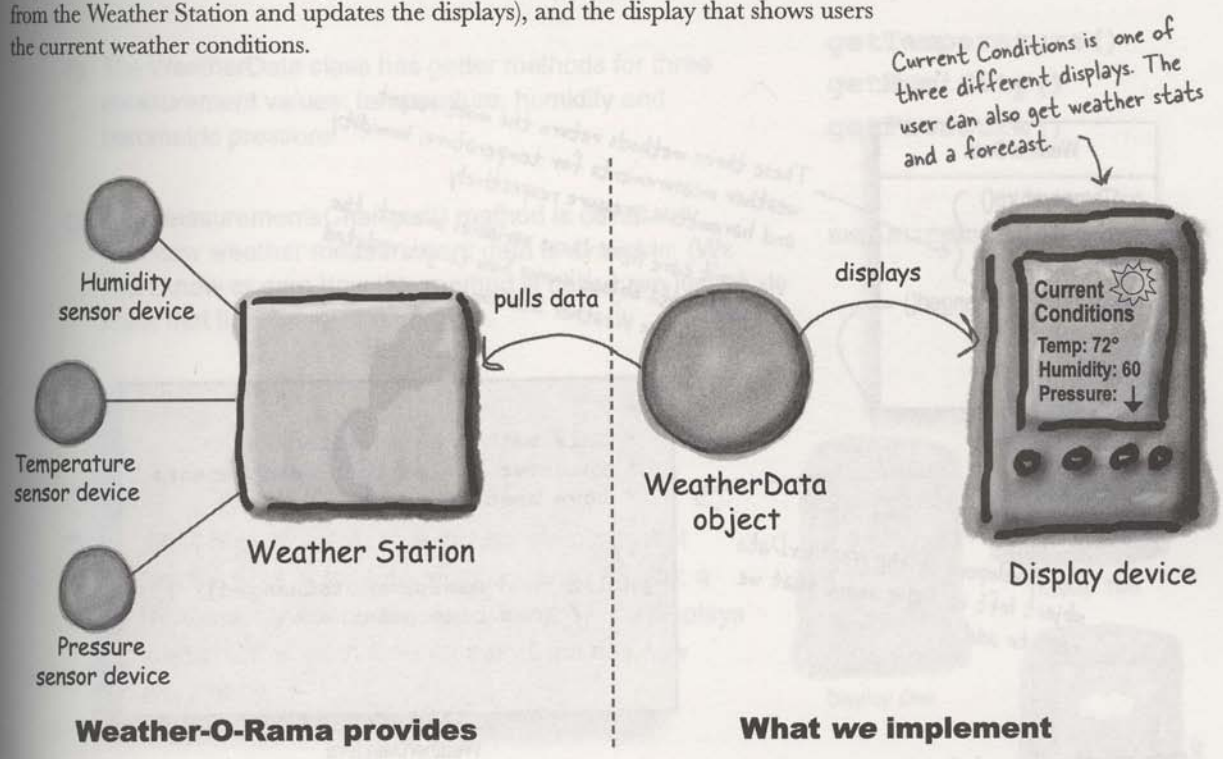
Johnny Hurricane

Johnny Hurricane, CEO

P.S. We are overnighting the WeatherData source files to you.

The Weather Monitoring application overview

The three players in the system are 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.

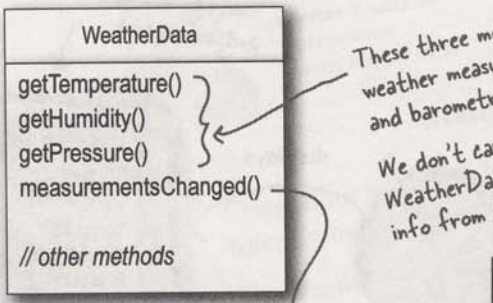


The WeatherData object knows how to talk to the physical Weather Station, to get updated data. The WeatherData object then updates its displays for the three different display elements: Current Conditions (shows temperature, humidity, and pressure), Weather Statistics, and a simple forecast.

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

As promised, the next morning the WeatherData source files arrive. Peeking inside the code, things look pretty straightforward:



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

We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

The developers of the WeatherData object left us a clue about what we need to add...

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

WeatherData.java

Remember, this Current Conditions is just ONE of three different display screens.

Display device

Our job is to implement measurementsChanged() so that it updates the three displays for current conditions, weather stats, and forecast.

What do we know so far?

The spec from Weather-O-Rama wasn't all that clear, but we have to figure out what we need to do. So, what do we know so far?

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

```
getTemperature()
getHumidity()
getPressure()
```

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

```
measurementsChanged()
```

- We 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 each time WeatherData has new measurements.

Display One

Display Two

Display Three

- The system must be expandable—other developers can create new custom display elements and users can add or remove as many display elements as they want to the application. Currently, we know about only the initial *three* display types (current conditions, statistics and forecast).

Future displays

Taking a first, misguided SWAG at the Weather Station

Here's a first implementation possibility—we'll take the hint from the Weather-O-Rama developers and add our code to the measurementsChanged() method:

```
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
}
```

Grab the most recent measuremets by calling the WeatherData's getter methods (already implemented).

Now update the displays...

Call each display element to update its display, 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 element we need to alter code.
- ❑ C. We have no way to add (or remove) display elements at run time.
- ❑ 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.

Definition of SWAG: Scientific Wild A** Guess

What's wrong with our implementation?

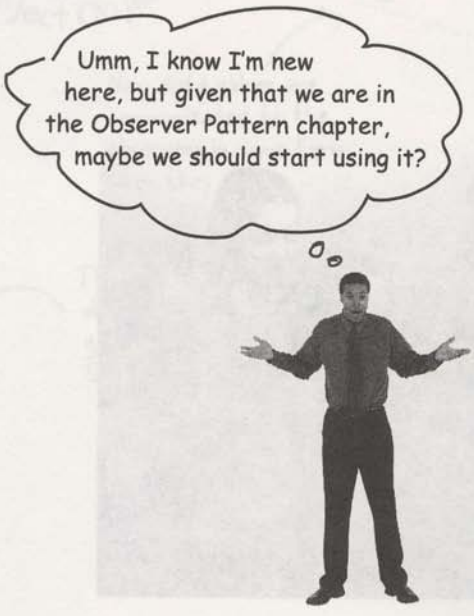
Think back to all those Chapter 1 concepts and principles...

```
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);
}
```

Area of change, we need to encapsulate this.

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

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



We'll take a look at Observer, then come back and figure out how to apply it to the weather monitoring app.

Meet the Observer Pattern

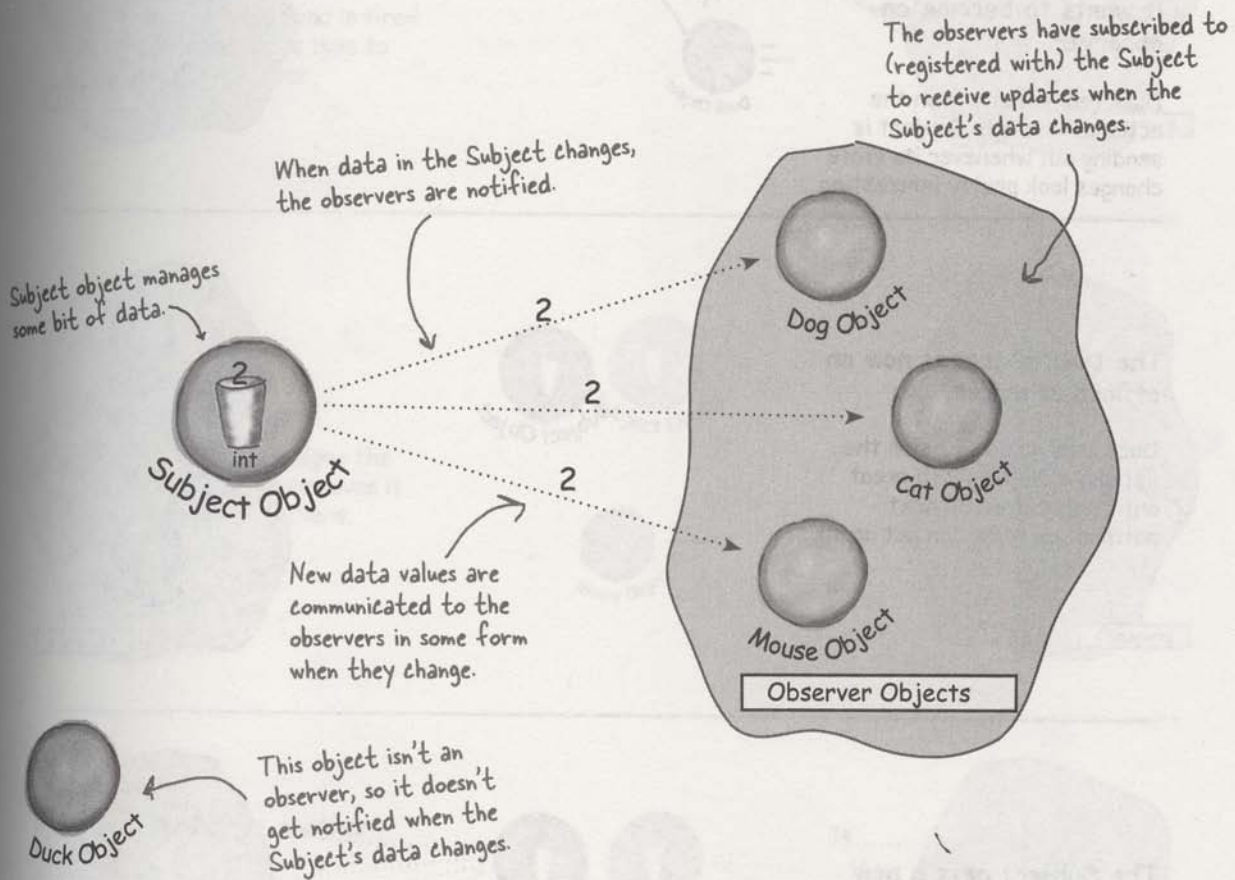
You know how newspaper or magazine subscriptions work:

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

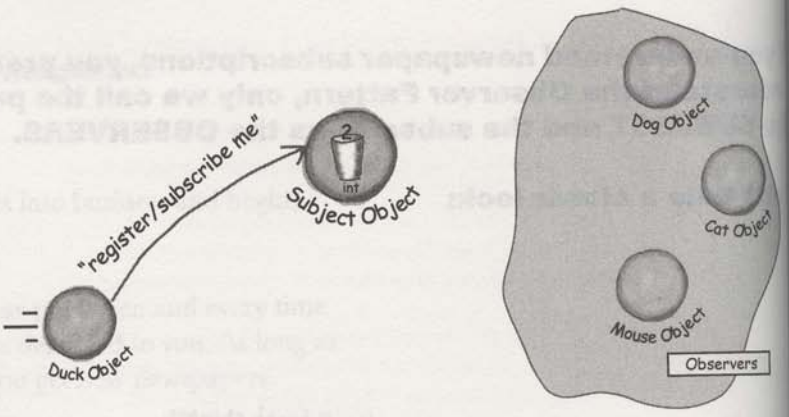
Let's take a closer look:



A day in the life of the Observer Pattern

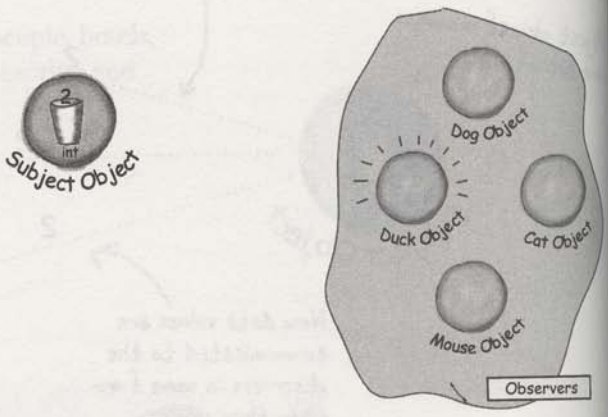
A Duck object comes along and tells the Subject that it wants to become an observer.

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



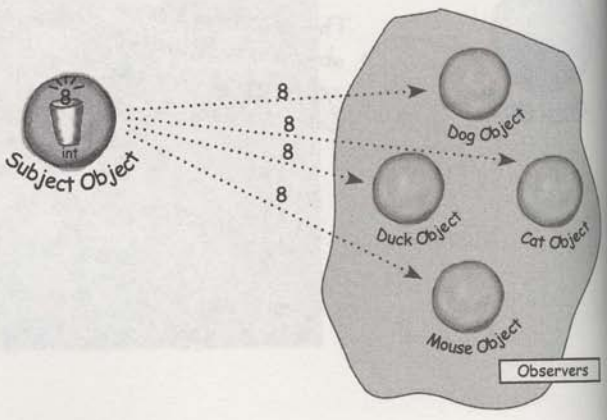
The Duck object is now an official observer.

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



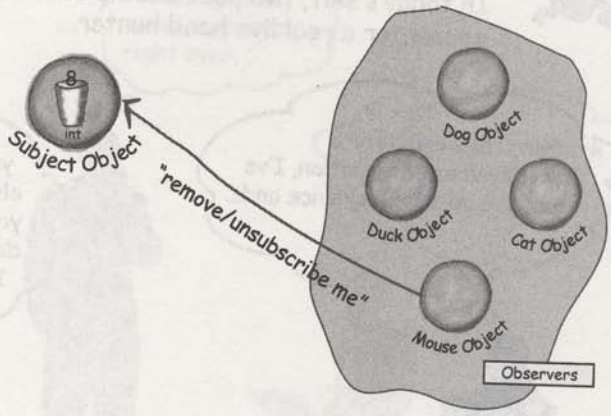
The Subject gets a new data value!

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



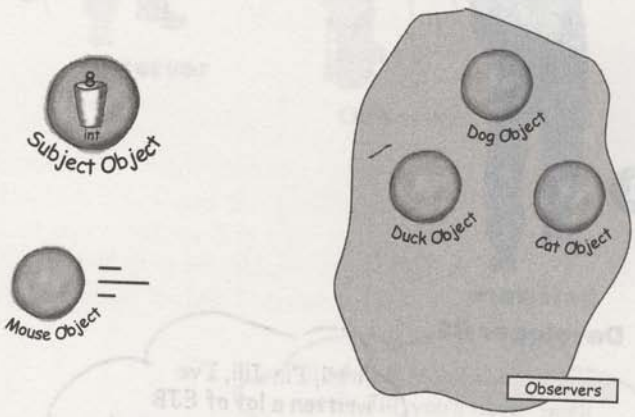
The Mouse object asks to be removed as an observer.

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



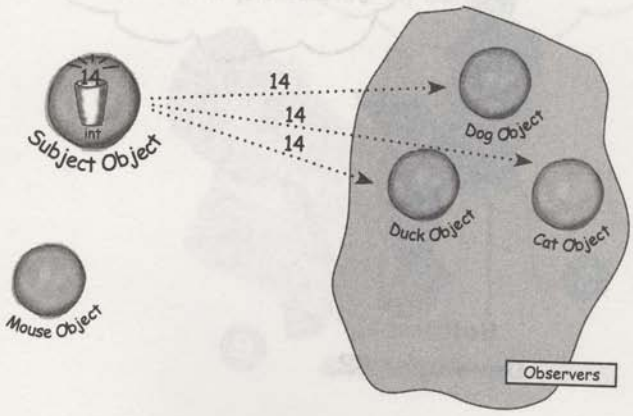
Mouse is outta here!

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



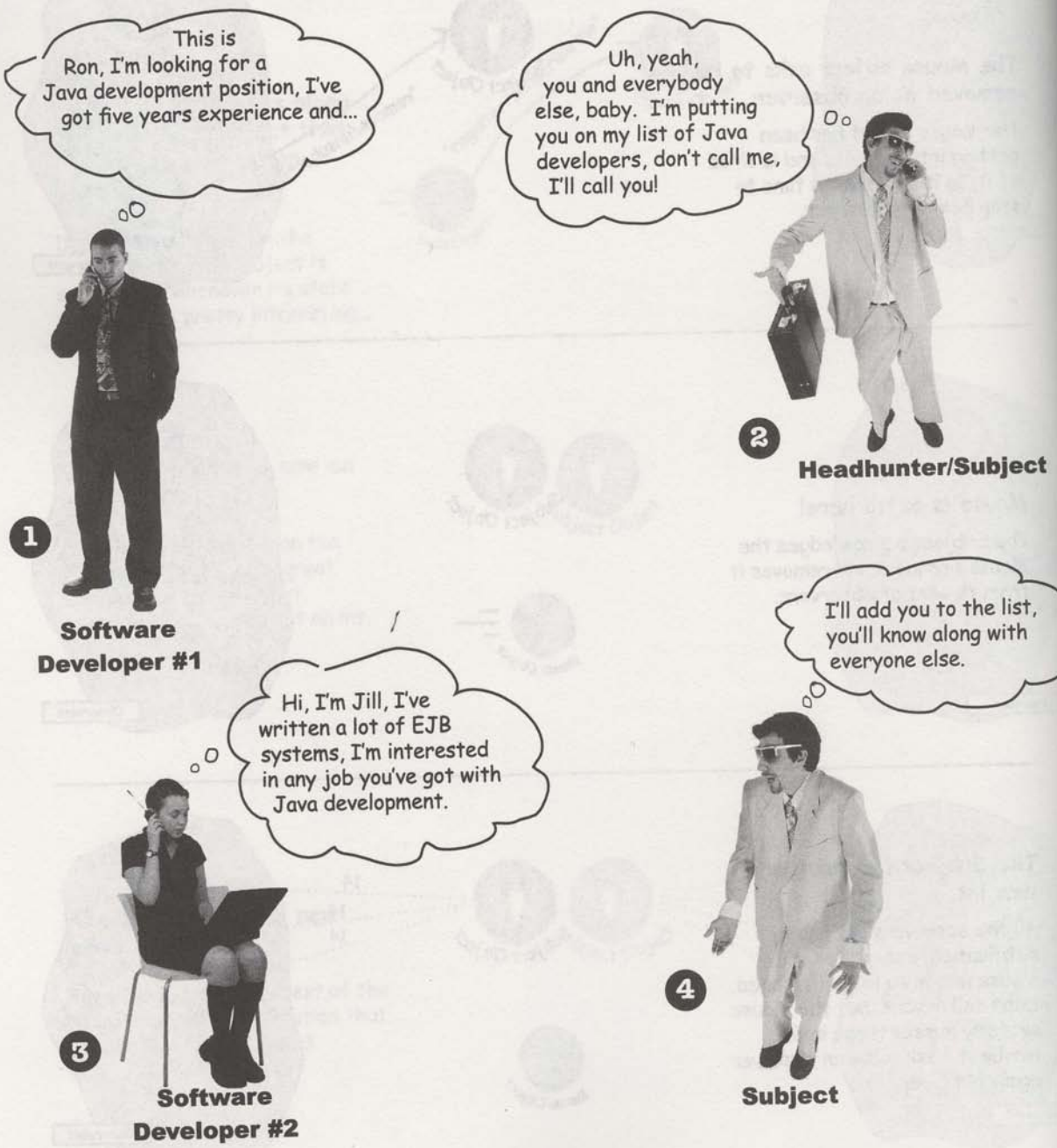
The Subject has another new int.

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 it'll ask to be an observer again some day.

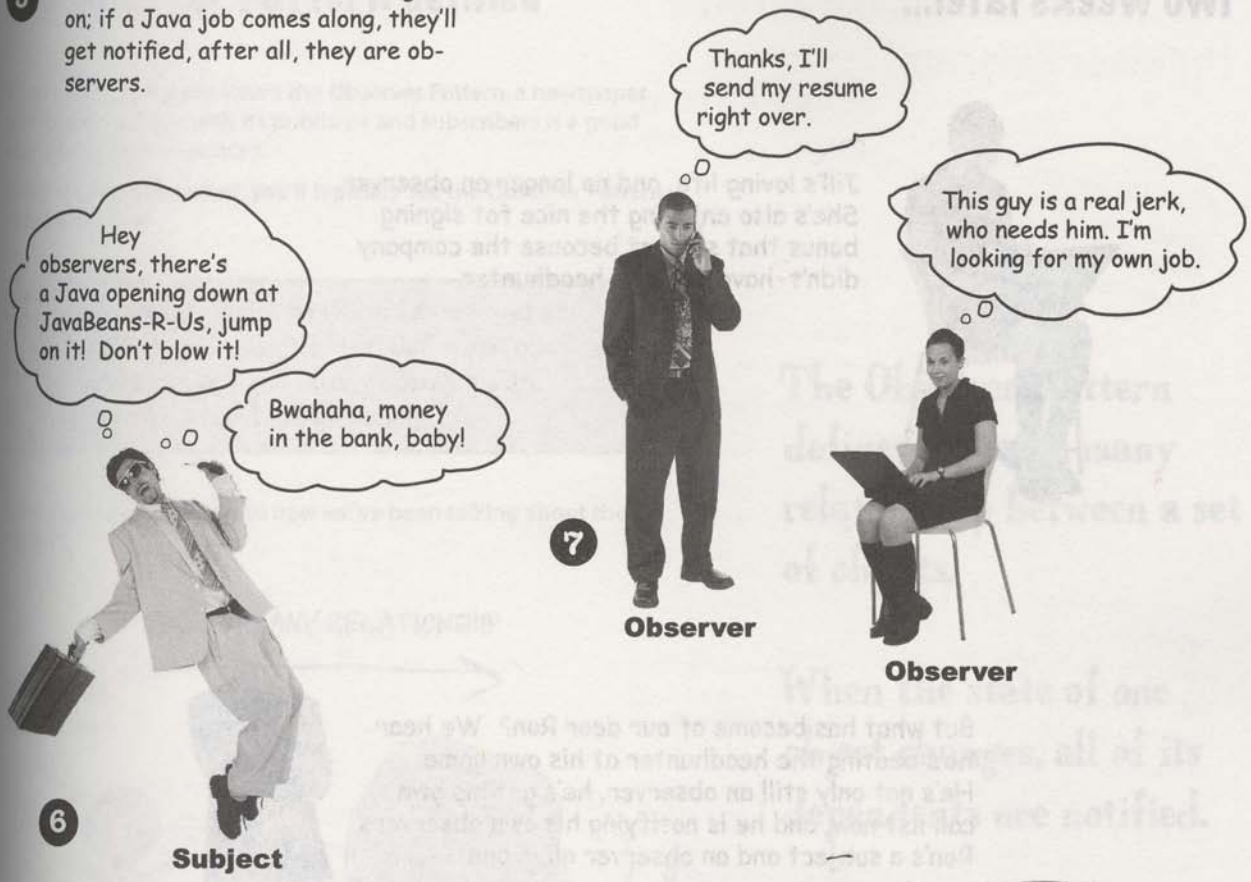


Five minute drama: a subject for observation

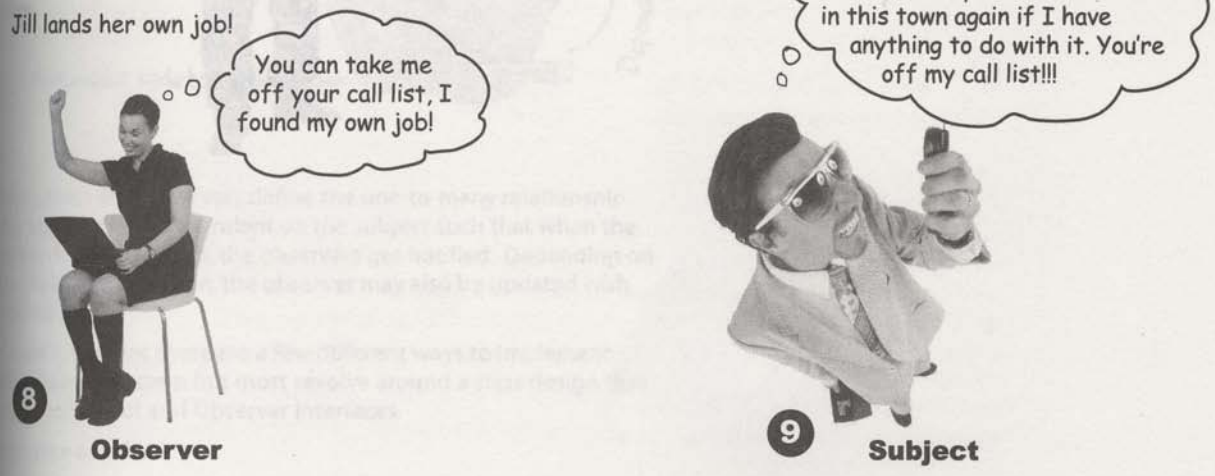
In today's skit, two post-bubble software developers encounter a real live head hunter...



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



Jill lands her own job!



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 Ron? We hear he's beating the headhunter at his own game. He's not only still an observer, he's got his own call list now, and he is notifying his own observers. Ron's a subject and an observer all in one.

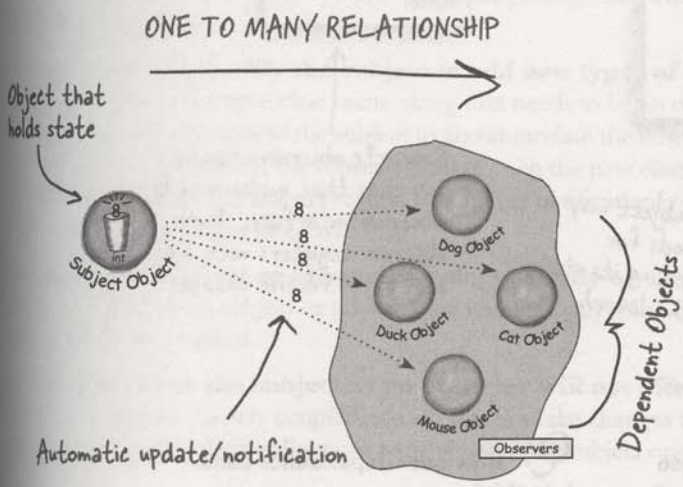
The Observer Pattern defined

When you're trying to picture the Observer Pattern, a newspaper subscription service 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.

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



The subject and observers define the one-to-many relationship. The observers are dependent on the subject such that when the subject's state changes, the observers get notified. Depending on the style of notification, the observer may also be updated with new values.

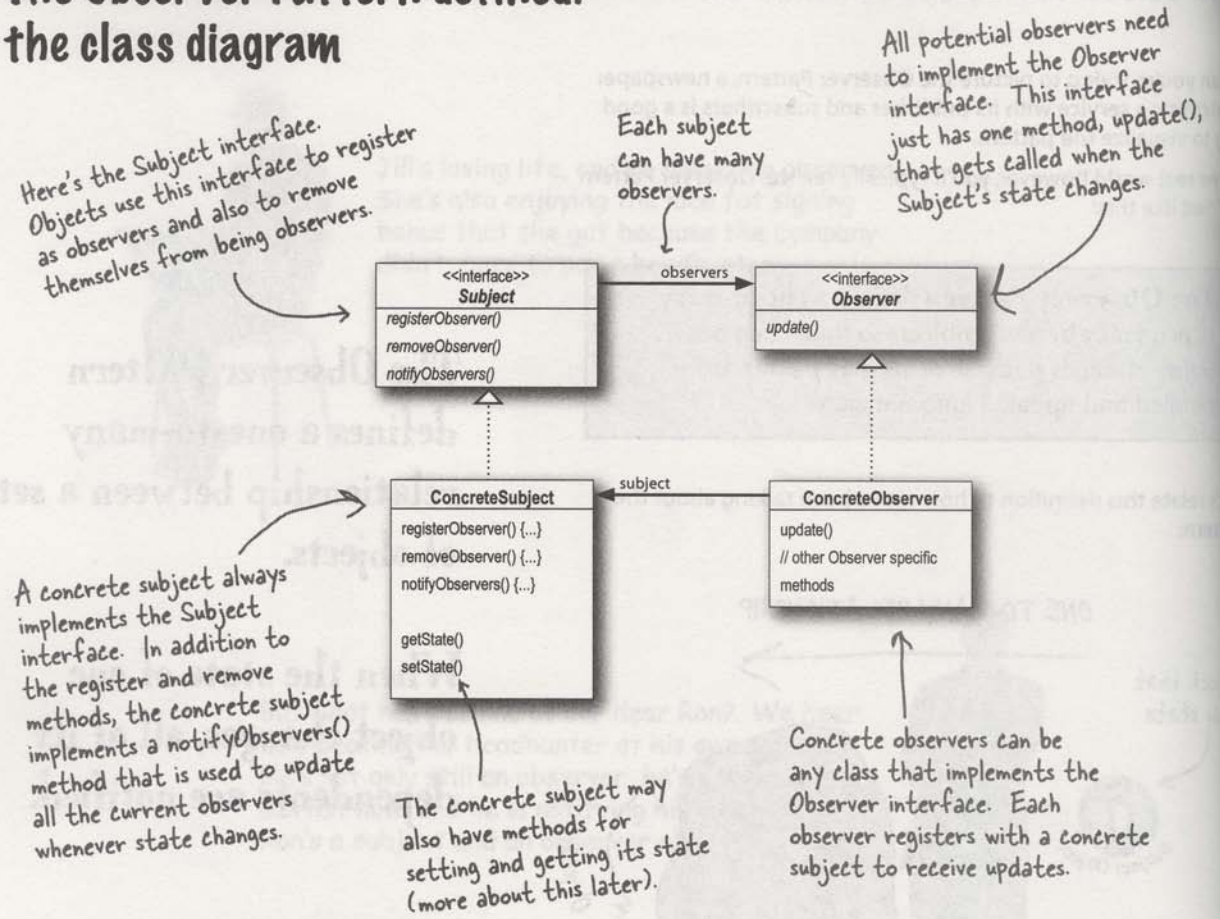
As you'll discover, there are a few different ways to implement the Observer Pattern but most revolve around a class design that includes Subject and Observer interfaces.

Let's take a look...

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.

The Observer Pattern defined: the class diagram



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.

The power of Loose Coupling

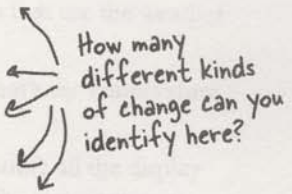
When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

The Observer Pattern provides an object design where subjects and observers are loosely coupled.

Why?

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.

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.

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

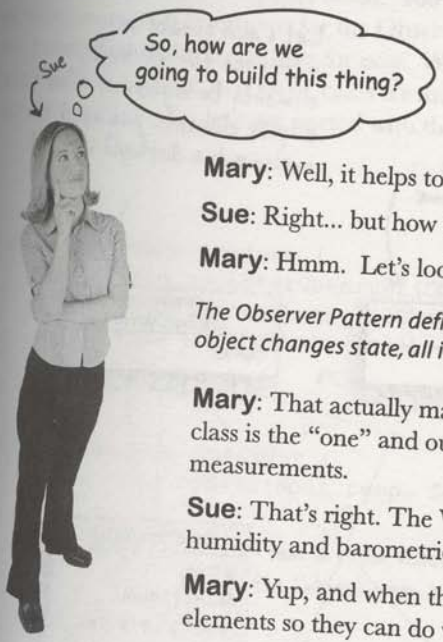
Sharpen your pencil

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

Back to the Weather Station project, your teammates have already started thinking through the problem...



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.

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.

Sue: Cool, I now 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?

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?

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.

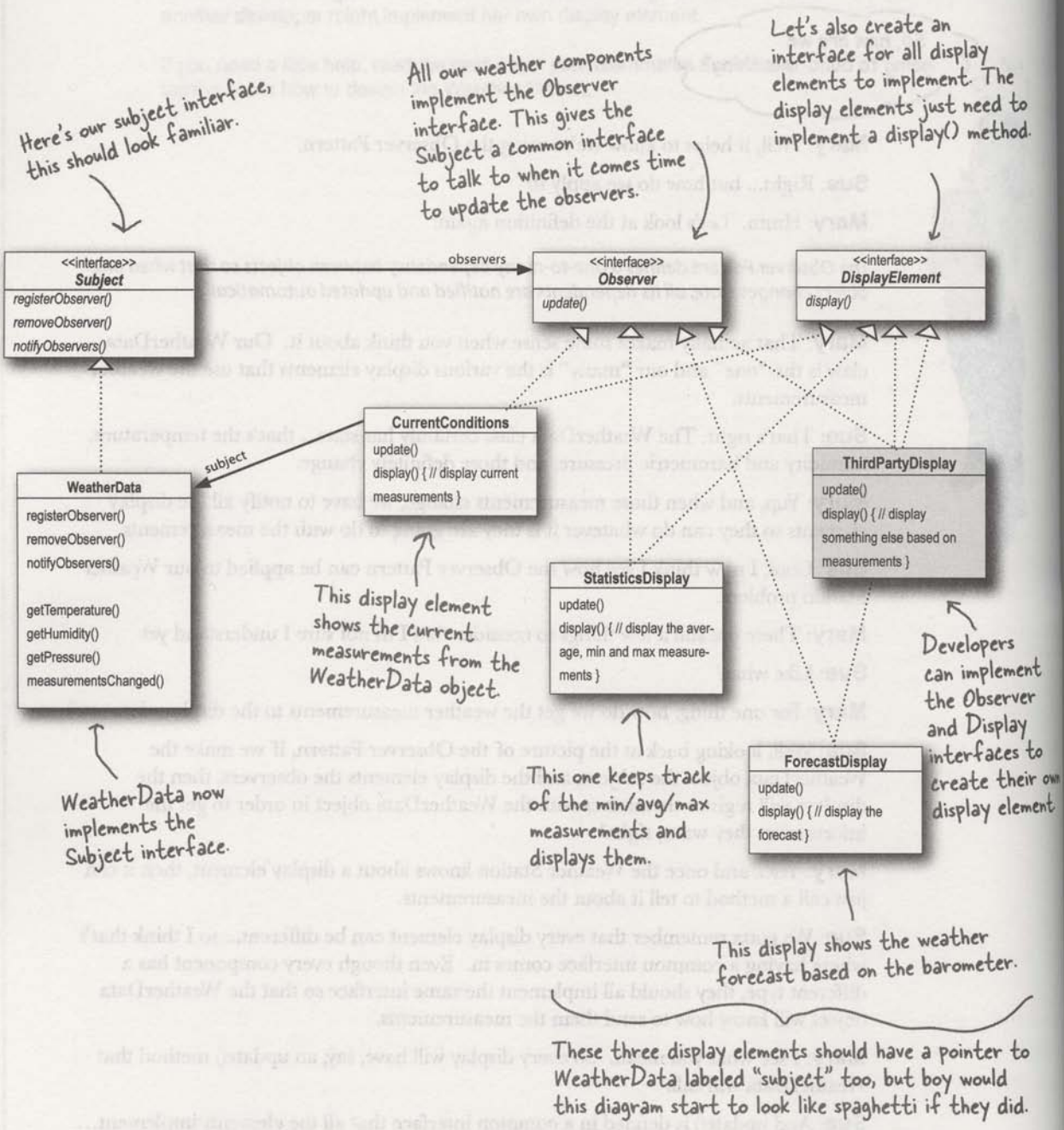
Sue: We gotta remember that every display element can be different... so I think that's where having a common interface comes in. Even though every component has a different type, they should all implement the same interface so that the WeatherData object will know how to send them the measurements.

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

We're going to start our implementation using the class diagram and following Mary and Sue's lead (from a few pages back). You'll see later in this chapter that Java provides some built-in support for the Observer pattern, however, we're going to get our hands dirty and roll our own for now. While in some cases you can make use of Java's built-in support, in a lot of cases it's more flexible to build your own (and it's not all that hard). So, let's get started 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.

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?

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 the wickedlysmart web site. You'll find the URL on page xxxiii in the Intro.

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

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

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            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 head first code directory. You'll see they are very similar.

This display implements Observer 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 Subject weatherData;

    public CurrentConditionsDisplay(Subject 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 are right, there are much better ways to design the way the data gets displayed. We are going to see this when we get to the model-view-controller pattern.

Q: Why did you store a reference to the 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. Here's our first attempt. We'll come back later in the book and make sure all the components are easily pluggable via a configuration file. For now here's how it all works:

```
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, 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 is actually feels). To compute the heat index, you take the temperature, T, and the relative humidity, RH, and use this formula:

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

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

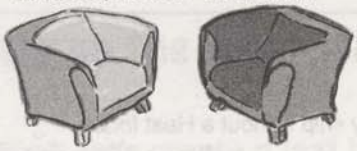
How does it work? You'd have to refer to *Head First Meteorology*, or try asking someone at the National Weather Service (or try a Google search).

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: **A Subject and Observer spar over the right way to get state information to the Observer.**

Subject

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.

Oh yeah, like what?

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

Well... I guess that might work. I'd have to open myself up even more 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.

Observer

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

Well yeah, but that's just a small part of who I am. Anyway, I 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 gets a little annoying at times...

Ok, wait just a minute here; first, we're not lazy, we just 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, I can see the advantages to doing it both ways. I have noticed that there is a built-in Java Observer Pattern that allows you to use either push or pull.

Great... maybe I'll get to see a good example of pull and change my mind.

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's 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, well 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.

Oh really? I think we're going to look at that next....

What, us agree on something? I guess there's always hope.

Using Java's built-in Observer Pattern

So far we've rolled our own code for the Observer Pattern, but Java has built-in support in several of its APIs. The most general is the Observer interface and the Observable class in the java.util package. These are quite similar to our Subject and Observer interface, but give you a lot of functionality out of the box. You can also implement either a push or pull style of *update to your observers, as you will see.*

To get a high level feel for java.util.Observer and java.util.Observable, check out this reworked OO design for the WeatherStation: