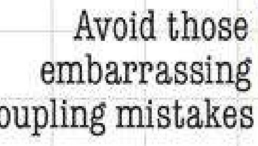
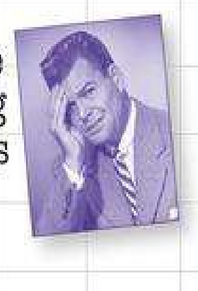
10th Anniversary
Updated for Java 8

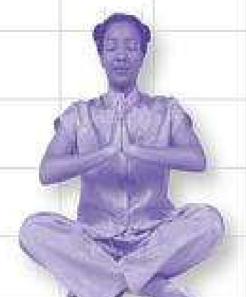
Head First Design Patterns

A Brain-Friendly Guide

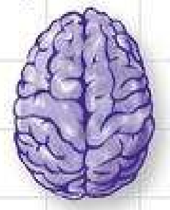

Avoid those embarrassing coupling mistakes



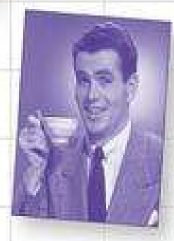
Learn why everything your friends know about Factory pattern is probably wrong


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 Robson
with Kathy Sierra & Bert Bates

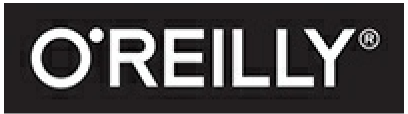
Head First: Design Patterns

Eric Freeman

Elisabeth Robson

Bert Bates

Kathy Sierra



Beijing • Boston • Farnham • Sebastopol • Tokyo

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*~~ *twenty years*.

Praise for *Head First Design Patterns*

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

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

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

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

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

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

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

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

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

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

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

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

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

introduction to design patterns that I have seen.”

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

“Jerry Rice runs patterns better than any receiver in the NFL, but Eric and Elisabeth have out run him. Seriously...this is one of the funniest and smartest books on software design I’ve ever read.”

— Aaron LaBerge, SVP Technology & Product Development, ESPN

More Praise for *Head First Design Patterns*

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

— Cory Doctorow, 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 Freeman and Elisabeth Robson have intrepidly volunteered to look behind the code curtain for us in *Head First Design Patterns*. I'm not sure either of them cares all that much about the PlayStation or 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, CEO and cofounder of Uber and Member of the MIT TR100

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

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

Praise for other books by Eric Freeman and Elisabeth Robson

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

— Satish Kumar

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

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

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

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

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

— David Kaminsky, Master Inventor, IBM

"This book takes you behind the scenes of JavaScript and leaves you with a deep understanding of how this remarkable programming language works."

— Chris Fuselier, Engineering Consultant

"I wish I'd had *Head First JavaScript Programming* when I was starting out!"

— Chris Fuselier, Engineering Consultant

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

— Frank Moore, Web designer and developer

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

— Tim Williams, software entrepreneur

Other O’Reilly books by Eric Freeman and Elisabeth Robson

Head First JavaScript Programming
Head First HTML and CSS
Head First HTML5 Programming

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

Authors of Head First Design Patterns

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

Professionally, Eric recently ended nearly a decade as a media company executive — having held the position of CTO of Disney Online & Disney.com at The Walt Disney Company. Eric is now devoting his time to WickedlySmart, a startup he co-created with Elisabeth.

By training, Eric is a computer scientist, having studied with industry luminary David Gelernter during his Ph.D. work at Yale University. His dissertation is credited as the seminal work in alternatives to the desktop metaphor, and also as the first implementation of activity streams, a concept he and Dr. Gelernter developed.

In his spare time, Eric is deeply involved with music; you’ll find Eric’s latest project, a collaboration with ambient music pioneer Steve Roach, available on the iPhone app store under the name Immersion Station.

Eric lives with his wife and young daughter in Austin, Texas. His daughter is

a frequent vistor to Eric's studio, where she loves to turn the knobs of his synths and audio effects.

Write to Eric at `eric@wickedlysmart.com` or visit his site at `ericfreeman.com`.

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

Elisabeth's been involved with the Internet since the early days; she co-created the award-winning web site, The Ada Project, one of the first web sites designed to help women in computer science find career and mentorship information online.

She's currently co-founder of WickedlySmart, an online education experience centered on web technologies, where she creates books, articles, videos, and more. Previously, as Director of Special Projects at O'Reilly Media, Elisabeth produced in-person workshops and online courses on a variety of technical topics and developed her passion for creating learning experiences to help people understand technology. Prior to her work with O'Reilly, Elisabeth spent time spreading fairy dust at The Walt Disney Company, where she led research and development efforts in digital media.

When not in front of her computer, you'll find Elisabeth hiking, cycling, or

kayaking in the great outdoors, with her camera nearby, or cooking vegetarian meals.

You can send her email at `beth@wickedlysmart.com` or visit her blog at `elisabethrobson.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).

Likes: running, skiing, skateboarding, playing with her Icelandic horses, and weird science. Dislikes: entropy.

You can find her on javaranch, or occasionally blogging at `seriouspony.com`. 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-

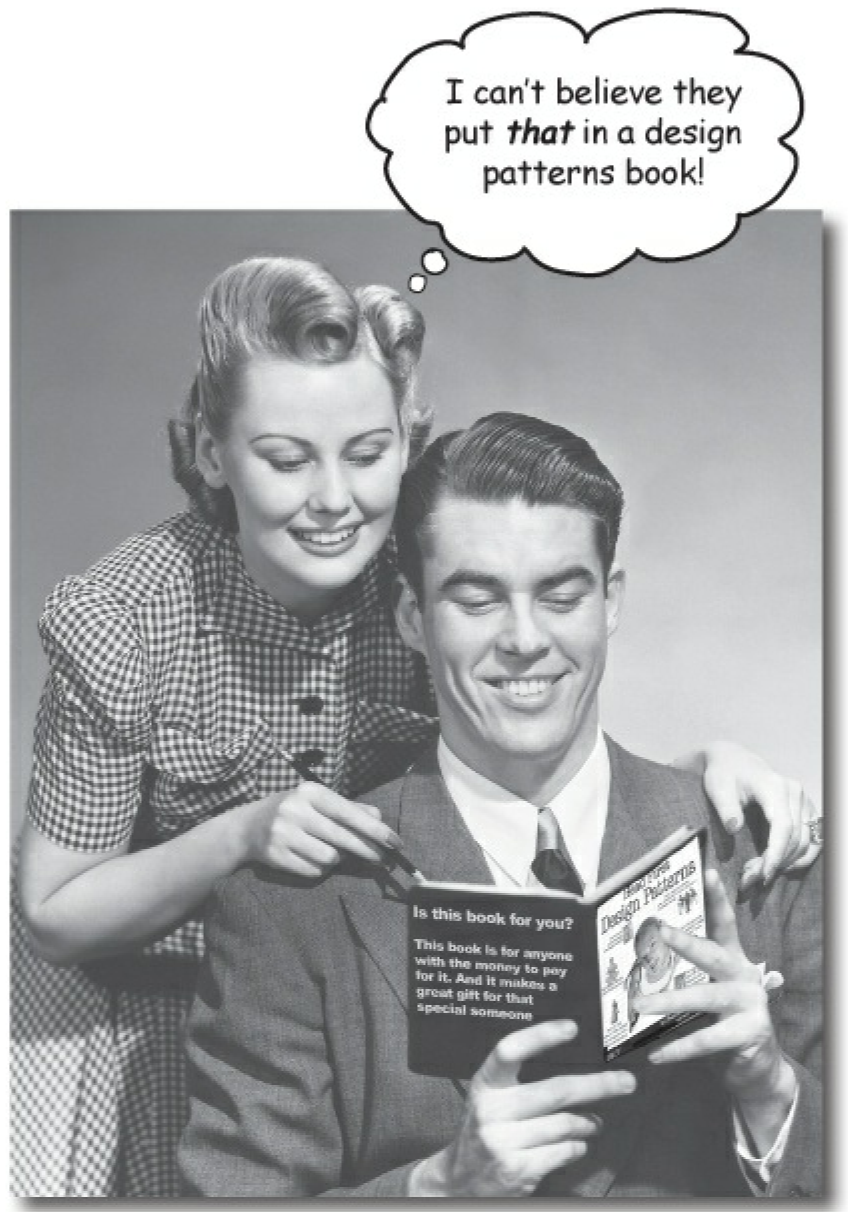
based training. He's been helping clients become better programmers ever since. Recently, he's been heading up the development team for several of Sun's Java Certification exams.

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`.

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.)

> **NOTE**
>
> 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.

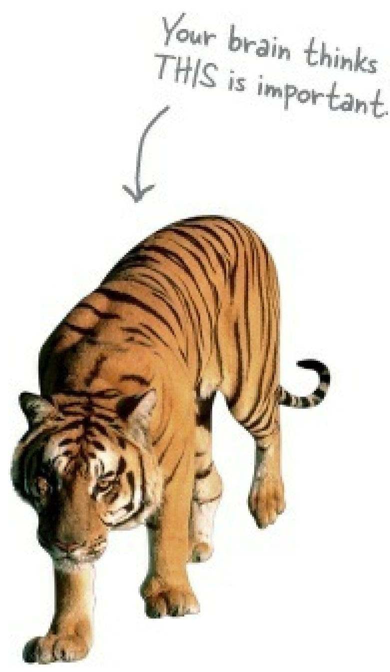
Today, you're less likely to be a tiger snack. But your brain's still looking. You just never know.

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.

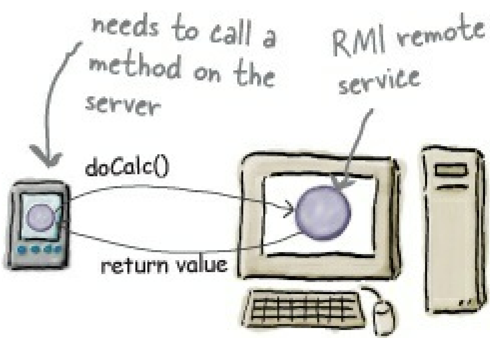
And there’s no simple way to tell your brain, “Hey brain, thank you very much, but no matter how dull this book is, and how little I’m registering on the emotional Richter scale right now, I really do want you to keep this stuff around.”

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.

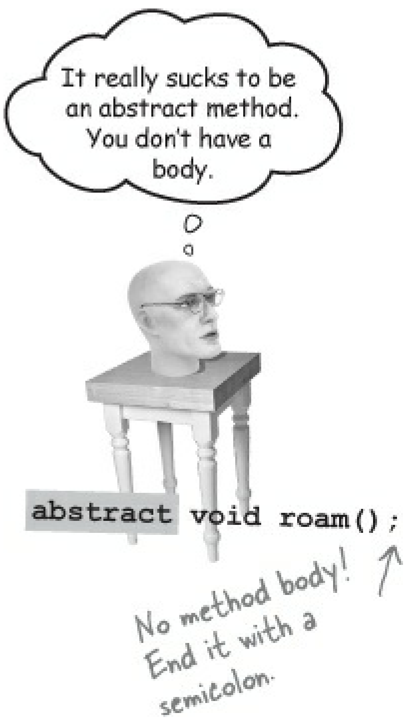
Some of the Head First learning principles:

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.

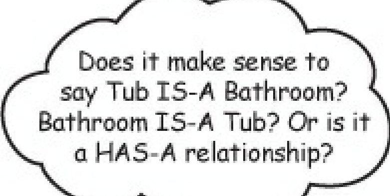


Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. 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

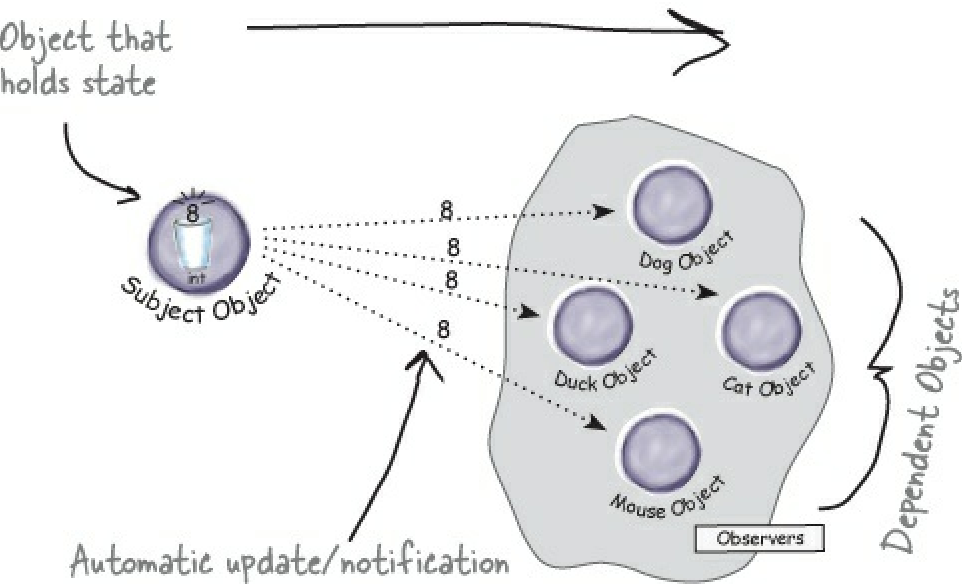
of the solution, and they’re all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to makes sense 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 1,024 words. And when text and pictures work together, we embedded the text in the pictures because your brain works more effectively when the text is within the thing 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***.

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*.

The Patterns Guru

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.

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.

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.

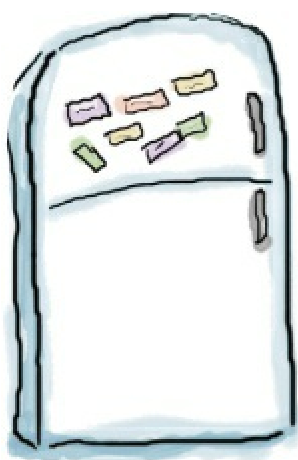
We used ***people***. In stories, examples, pictures, etc., because, well, because *you're* a person. And your brain pays more attention to *people* than it does to *things*.

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 refrigerator.

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.

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

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

⑤ **Drink water. Lots of it.**

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

⑥ **Talk about it. Out loud.**

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

⑦ **Listen to your brain.**

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

⑧ ***Feel* something!**

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

⑨ ***Design* something!**

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

Read Me

This is a learning experience, not a reference book. We deliberately stripped

out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

We use simple UML-like diagrams.

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

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

There are a *lot* of Design Patterns. The original foundational patterns (known as the GoF patterns), enterprise Java patterns, 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 JSE API, we mention it. We've also placed all the source code on the Web so you can download it. You'll find it at `http://wickedlysmart.com/head-first-design-patterns/`

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

packages.

The Brain Power exercises don’t have answers.

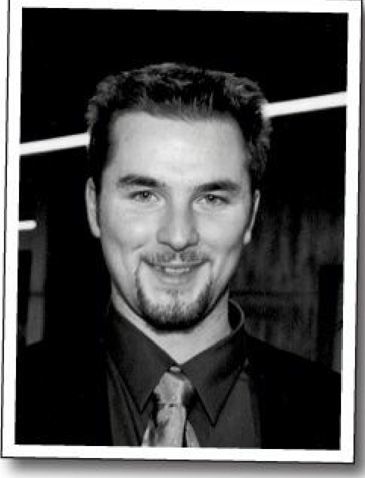
For some of them, there is no right answer, and for others, part of the learning experience of the Brain Power activities is for you to decide if and when your answers are right. In some of the Brain Power exercises you will find hints to point you in the right direction.

Tech Reviewers

Valentin Crettaz

Jef Cumps

Barney Marispini



Ike Van Atta

Fearless leader of the HFDP Extreme Review Team.

Johannes deJong

Jason Menard

Mark Spritzler

Dirk Schreckmann

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**, "In Design King" **Ron Bilodeau**, rock-and-roll star **Ellie Volkhausen** for her inspired cover design, **Melanie Yarbrough** for shepherding production, **Colleen Gorman** and **Rachel Monaghan** for their hardcore copyedits, and **Bob Pfahler** for a much improved index. Finally, thanks to **Mike Hendrickson** and **Meghan Blanchette** for championing this book and building the team.

Our intrepid reviewers:

We are extremely grateful for our technical review director **Johannes deJong**. You are our hero, Johannes. And we deeply appreciate the contributions of the co-manager of the **Javaranch** review team, the late **Philippe Maquet**. You have single-handedly brightened the lives of thousands of developers, and the impact you've had on their (and our) lives is

forever. **Jef Cumps** is scarily good at finding problems in our draft chapters, and once again made a huge difference for the book. Thanks Jef! **Valentin Cretazz** (AOP guy), who has been with us from the very first Head First book, proved (as always) just how much we really need his technical expertise and insight. You rock Valentin (but lose the tie).

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

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

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

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

Even more people[1]

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, Mike Hendrickson**, and **Meghan**

Blanchette. Mike L. was with us every step of the way. Mike, your insightful feedback helped shape the book and your encouragement kept us moving ahead. Mike H., thanks for your persistence over five years in trying to get us to write a patterns book; we finally did it and we're glad we waited for Head First. And Meg, thanks for diving into the update with us; we couldn't have done it without you.

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. Thanks to Apple's **James Dempsey** for allowing us to use his MVC song. And thank you to **Richard Warburton** who made sure our Java 8 code updates were up to snuff for this updated edition of the book.

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

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?

[1] 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

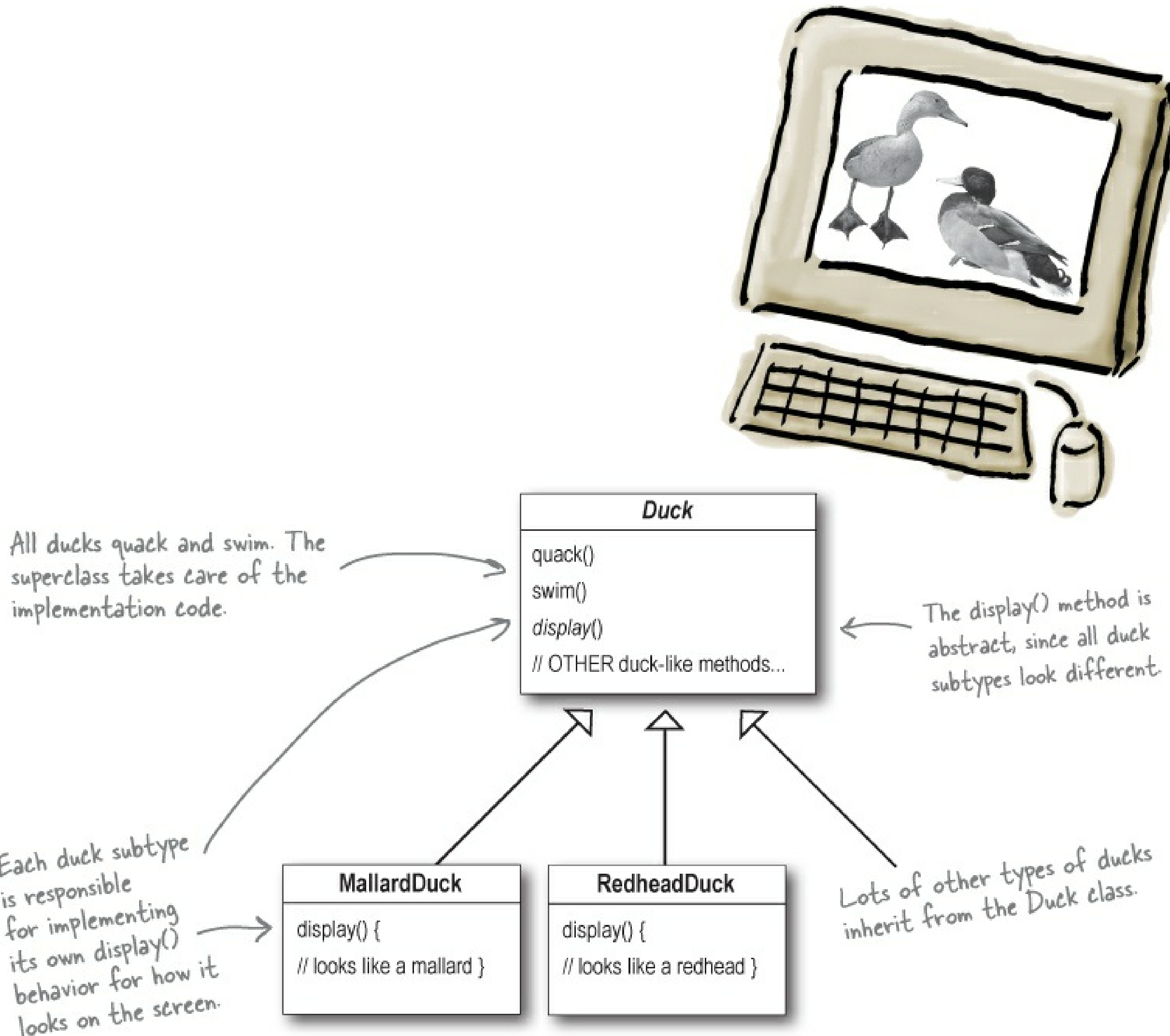
you have a large family, write to us.

Chapter 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

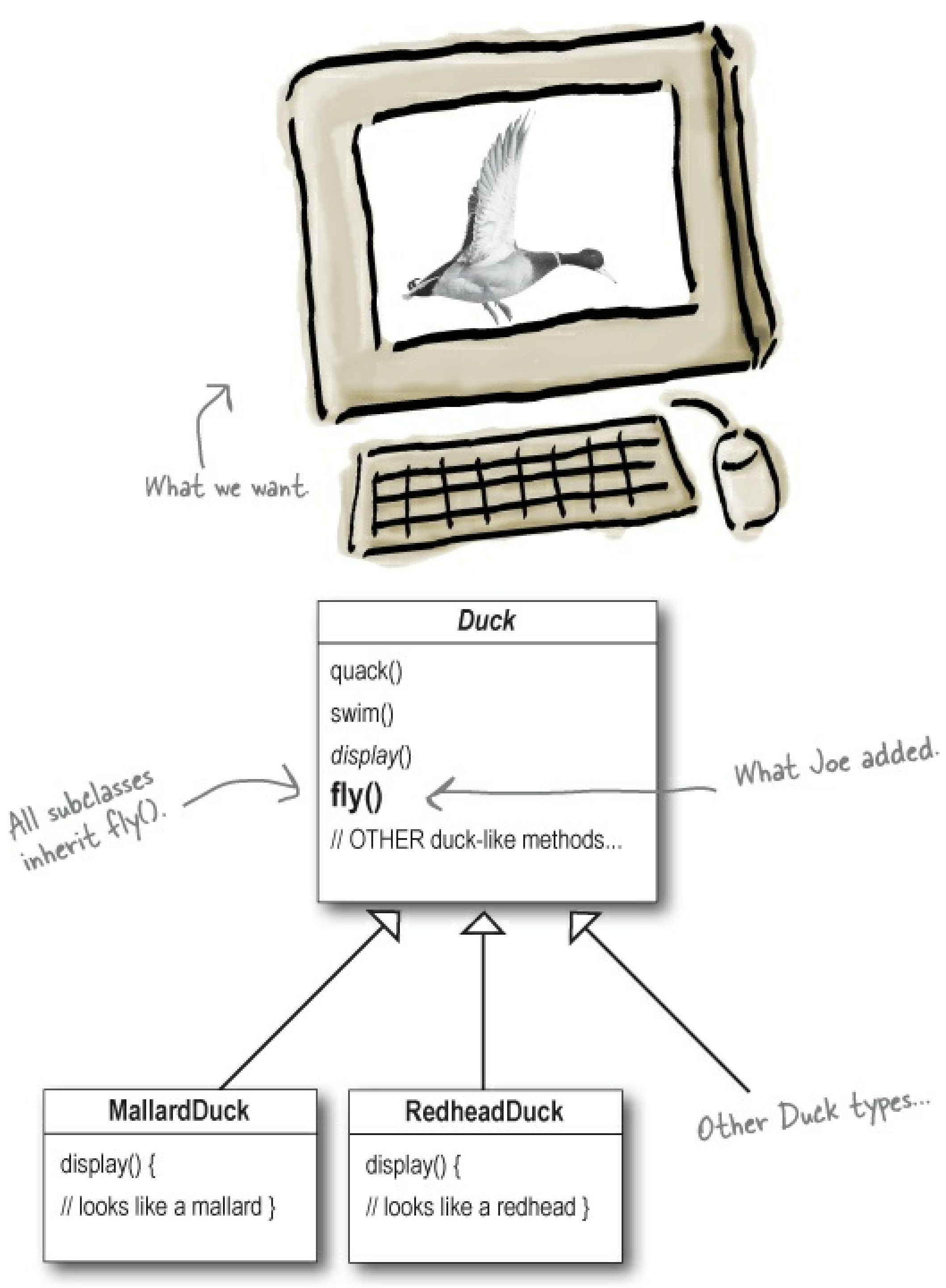
Joe works for a company that makes a highly successful duck pond simulation game, *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck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



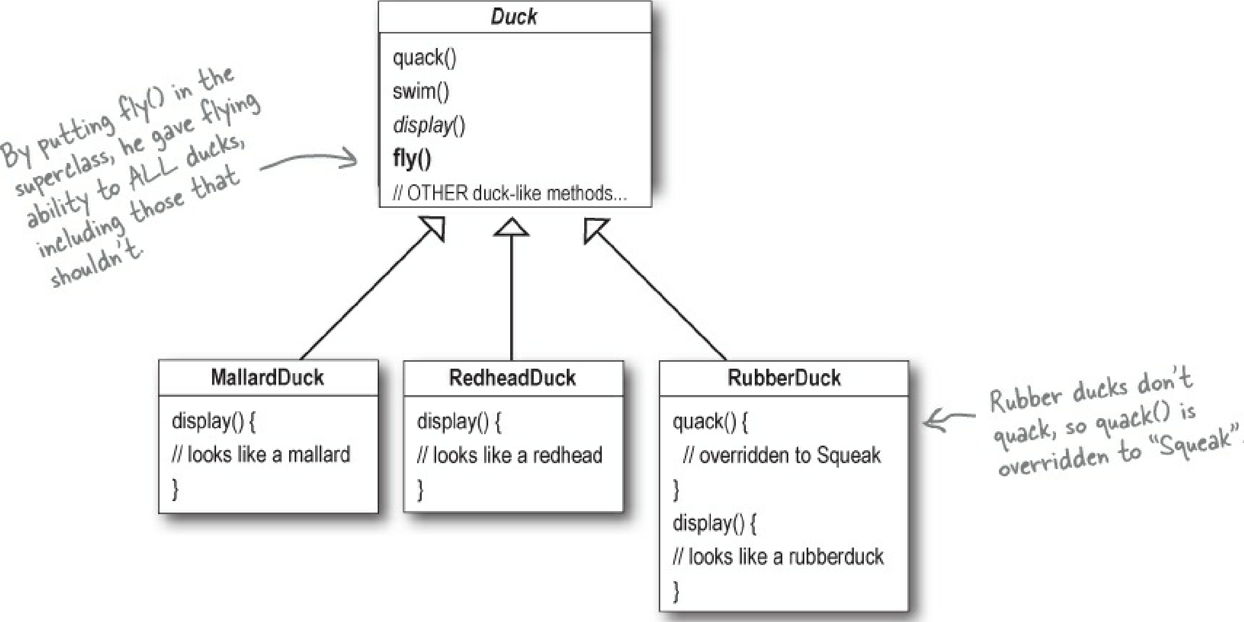
But something went horribly wrong...

Joe, I'm at the shareholder's meeting. They just gave a demo and there were **rubber duckies** flying around the screen. Was this your idea of a joke? You might want to spend some time on Monster.com...

What happened?

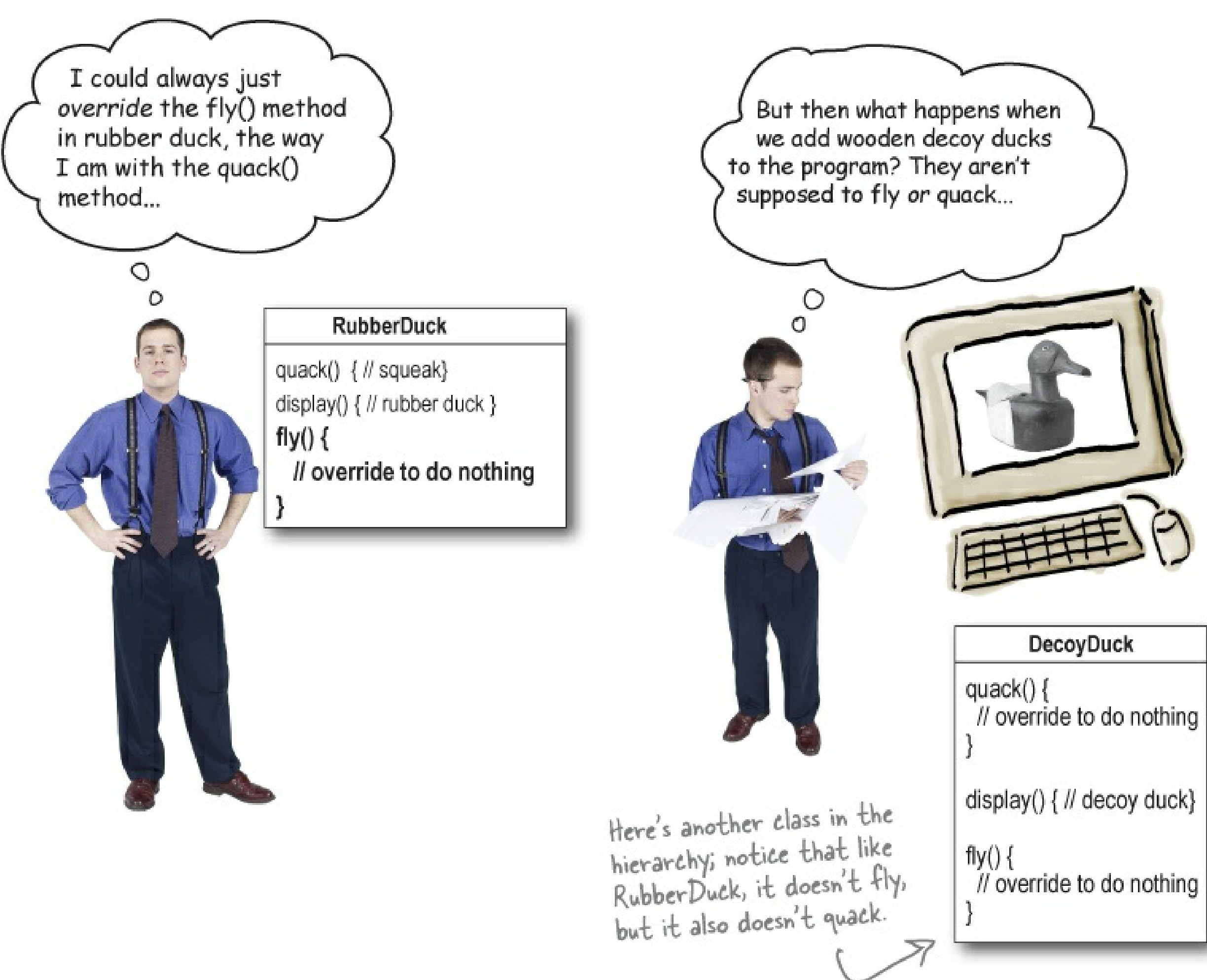
Joe failed to notice that not *all* subclasses of Duck should *fly*. When Joe added new behavior to the Duck superclass, he was also adding behavior that was *not* appropriate for some Duck subclasses. He now has flying inanimate objects in the SimUDuck program.

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



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

Joe thinks about inheritance...



SHARPEN YOUR PENCIL

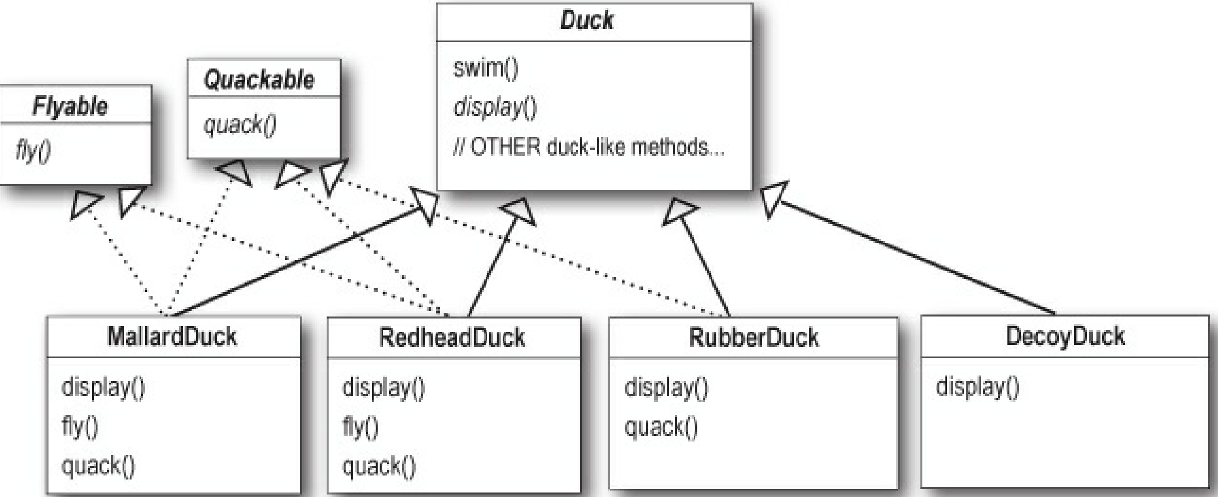
Which of the following are disadvantages of using *inheritance* to provide Duck behavior? (Choose all that apply.)

❏	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.



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

What do YOU think about this design?

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.*

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.

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

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!

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!

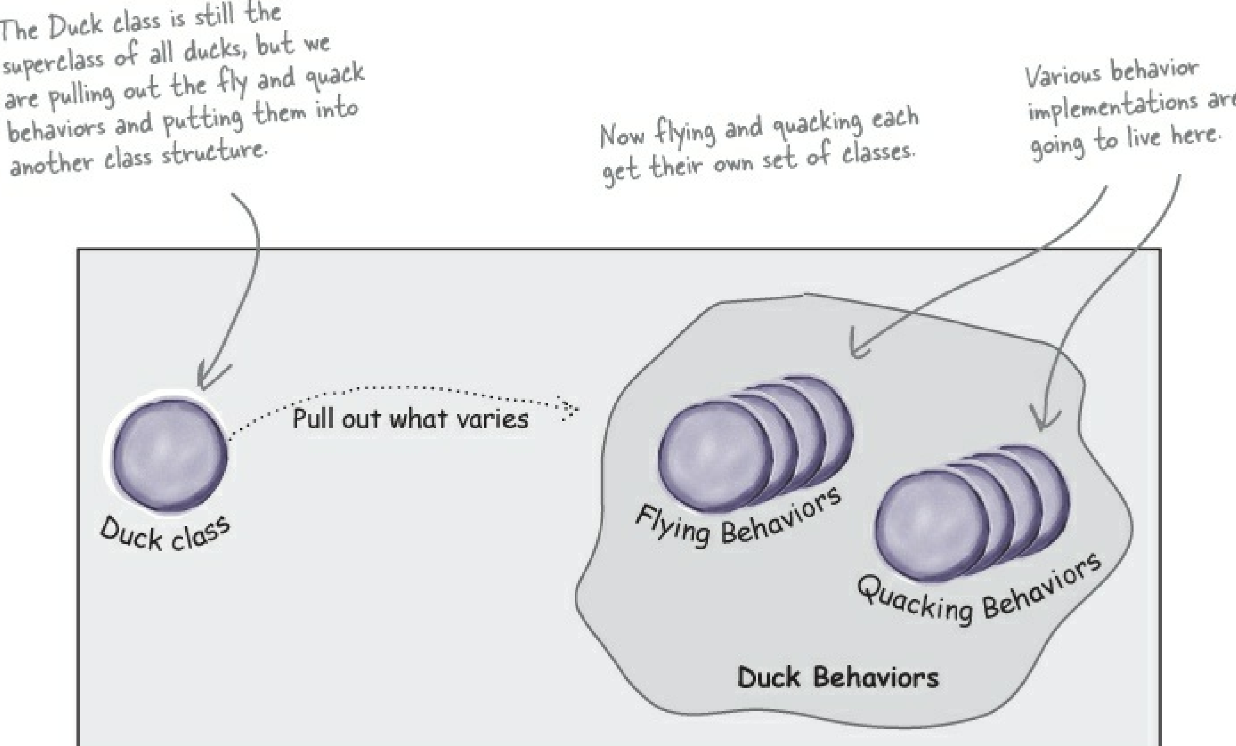
Separating what changes from what stays the same

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

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

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

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



Designing the Duck Behaviors

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

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

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

DESIGN PRINCIPLE

Program to an interface, not an implementation.

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

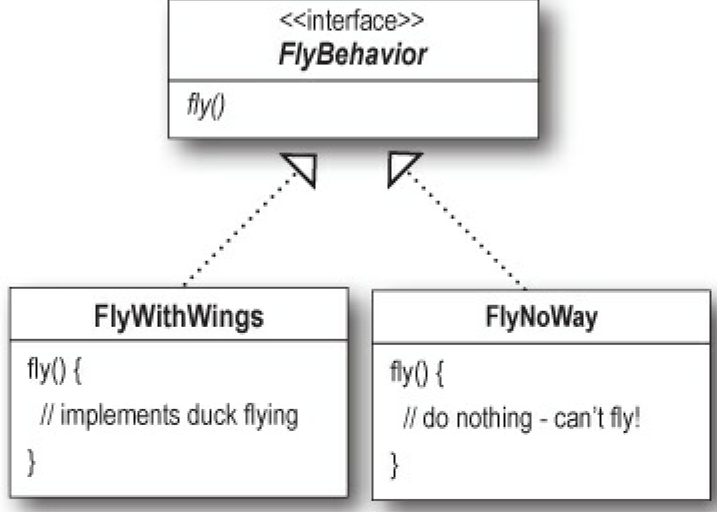
That way, the Duck classes won't need to know any of the implementation details for their own behaviors.

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

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

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

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



"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();
```

> **NOTE**
>
> 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:

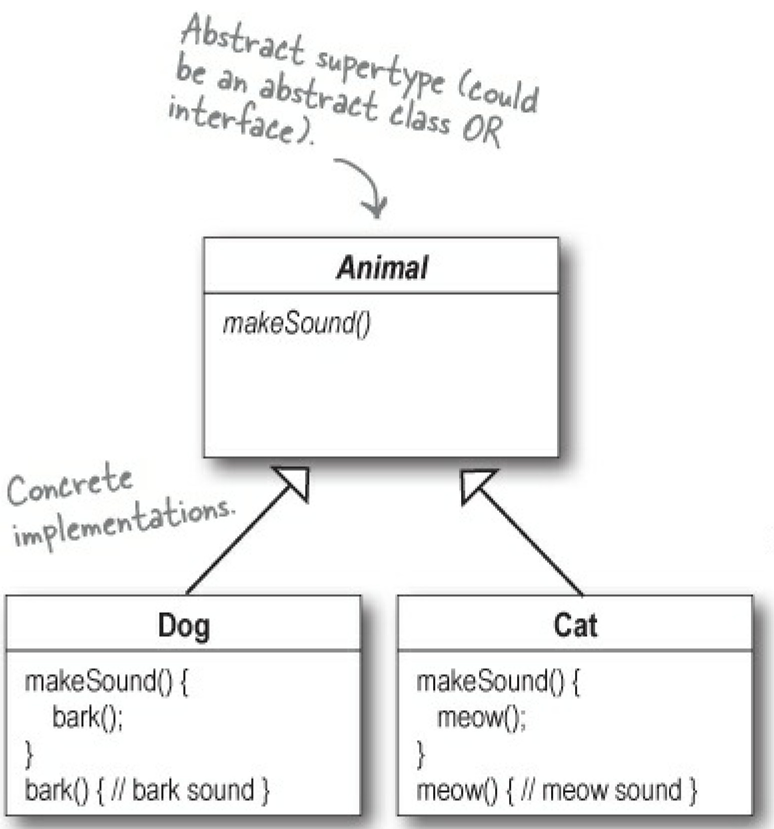
```
Animal animal = new Dog();
animal.makeSound();
```

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

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

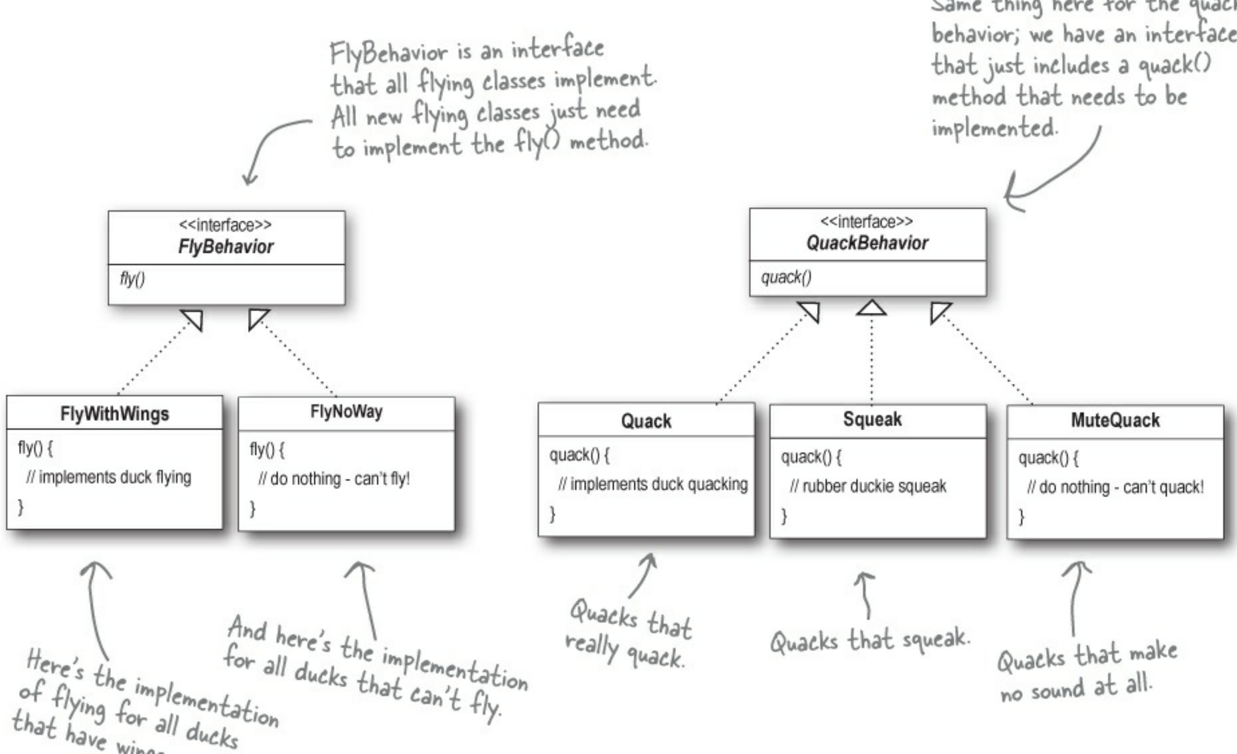
```
a = getAnimal();
a.makeSound();
```

> **NOTE**
>
> 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:



NOTE

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: 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: 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: Q: Should we make Duck an interface too?

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

Q: 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: **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.

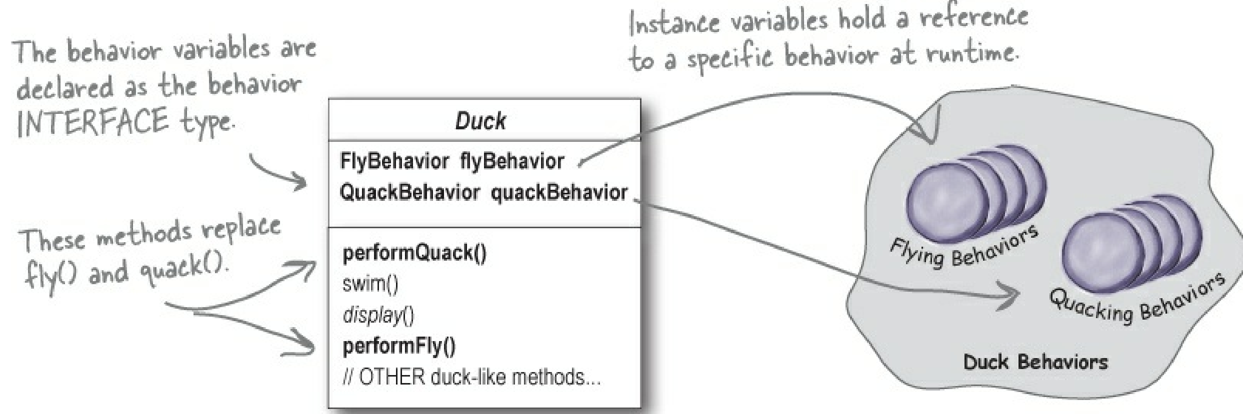
2) One example, a duck call (a device that makes duck sounds).

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!");
    }
}
```


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).

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

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

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

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

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

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

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

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

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

⑤ **Run the code!**

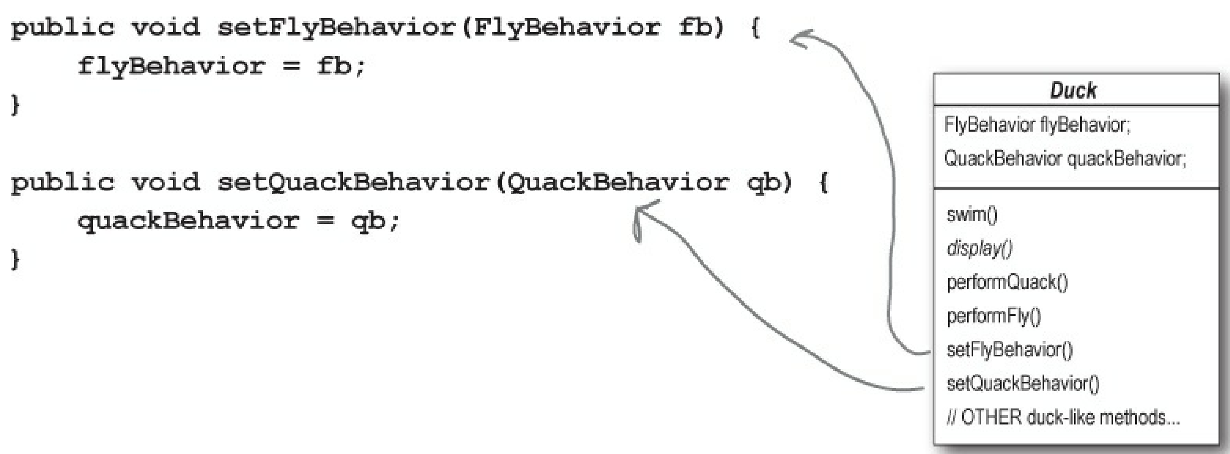
```
File  Edit  Window  Help  Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```

Setting behavior dynamically

What a shame to have all this dynamic talent built into our ducks and not be using it! Imagine you want to set the duck's behavior type through a setter method on the duck 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.

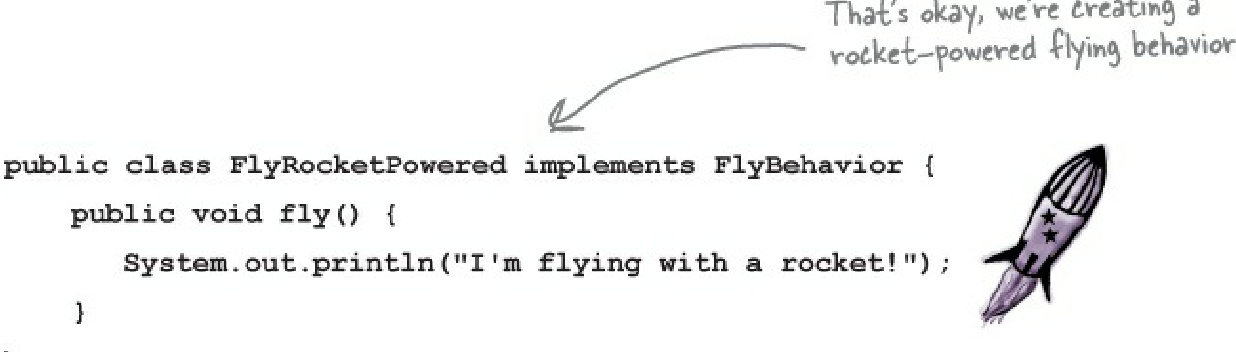
> **NOTE**
>
> Editor note: gratuitous pun - fix

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");
    }
}
```

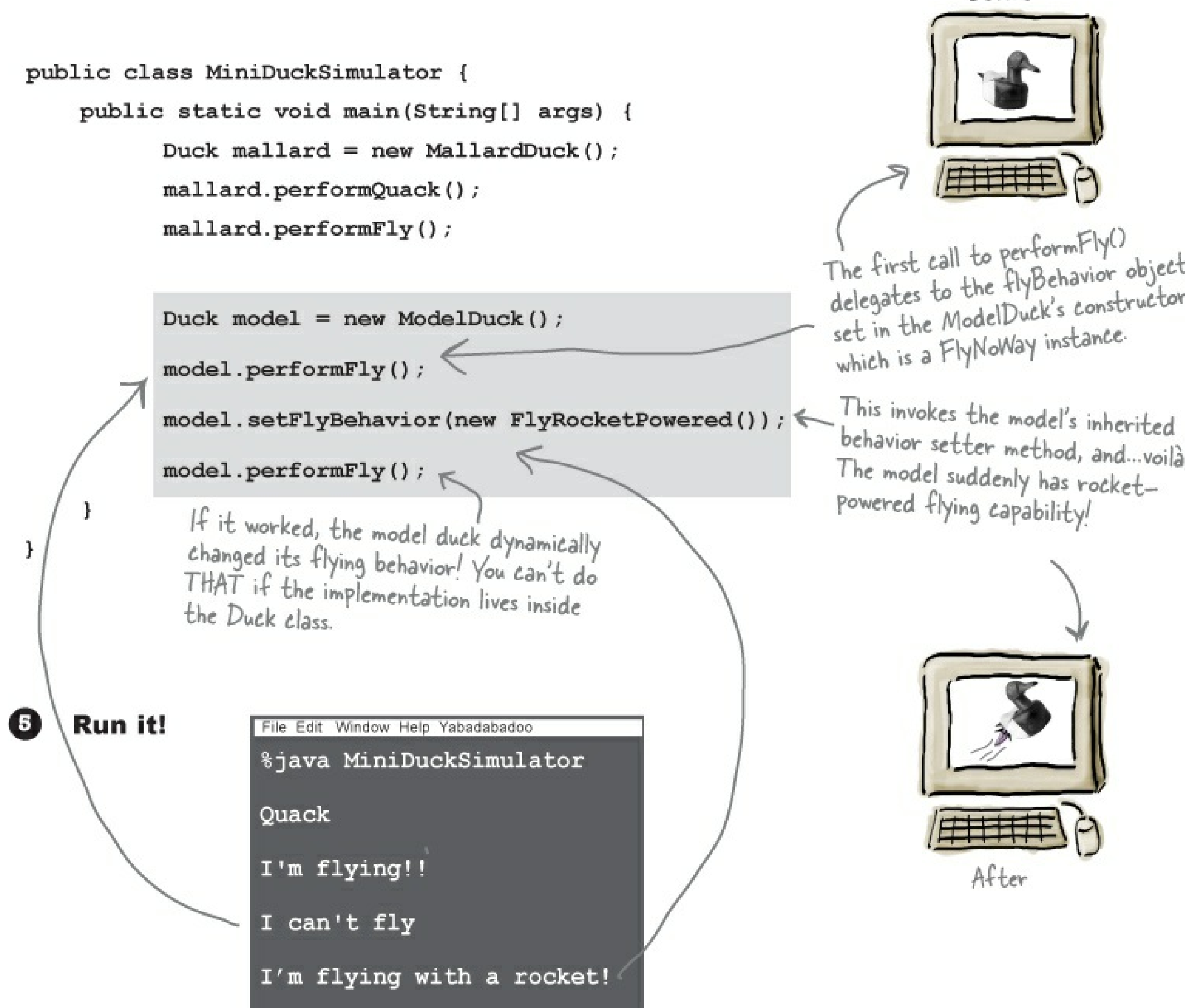
Our model duck begins life grounded... without a way to fly.

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




```
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.



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

The Big Picture on encapsulated behaviors

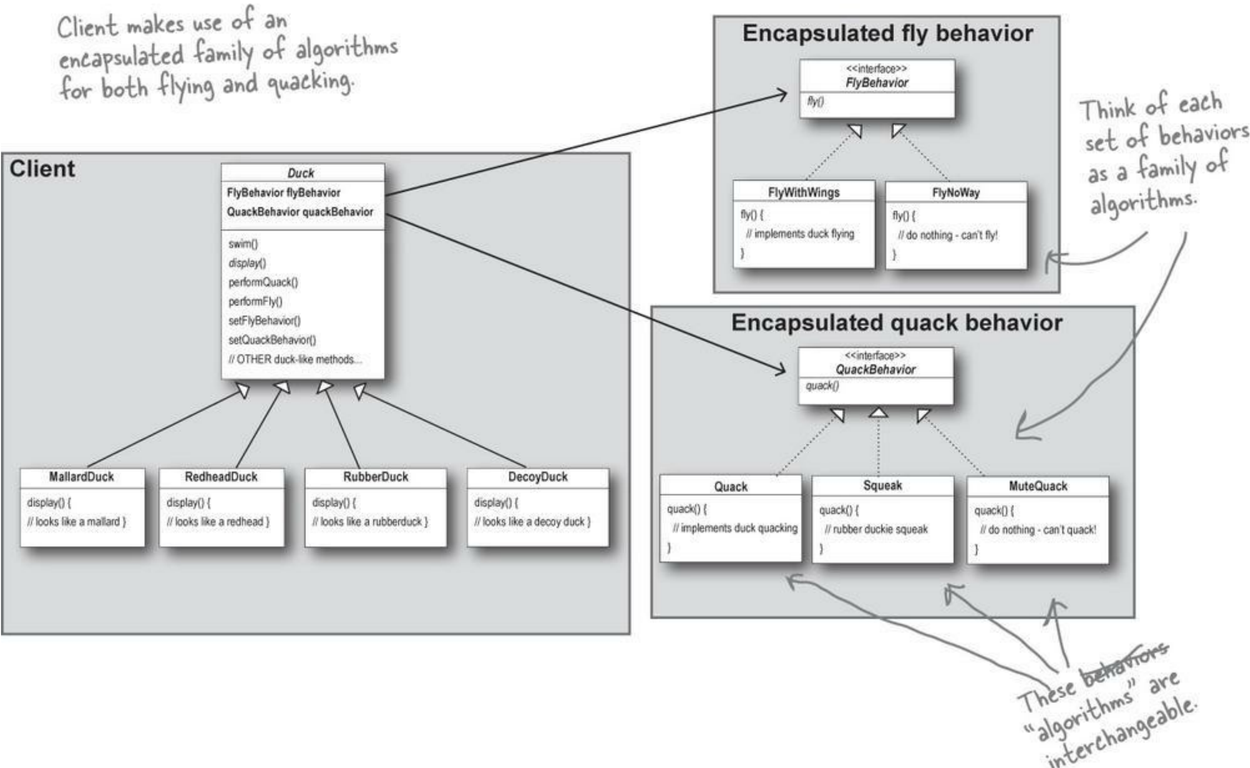
Okay, now that we've done the deep dive on the duck simulator design, it's time to come back up for air and take a look at the big picture.

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.

BRAIN POWER

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

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 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 on initial development.*

Master: *So Grasshopper, should effort go into reuse* ***above*** *maintainability 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 Maui.

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

> **NOTE**
>
> **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.)

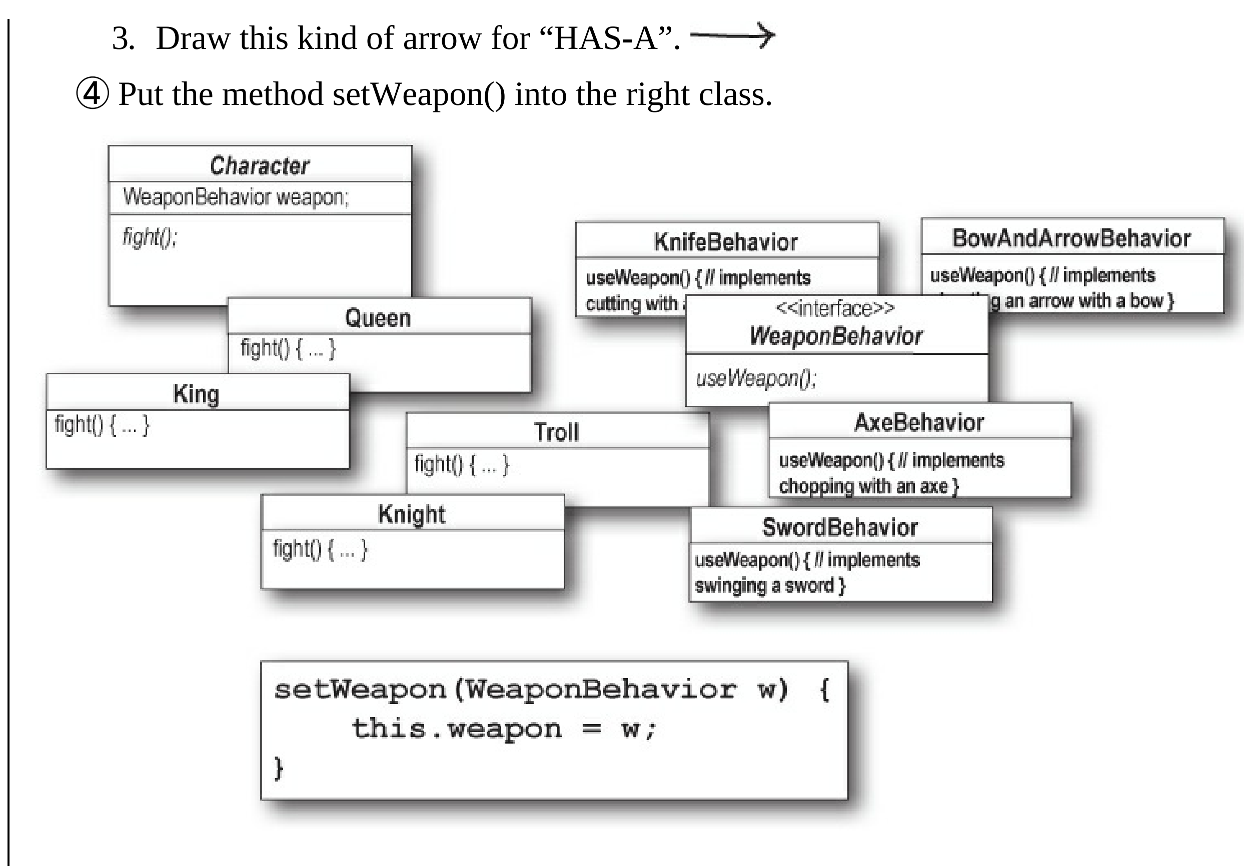
Your task:

① Arrange the classes.
② Identify one abstract class, one interface, and eight classes.
③ Draw arrows between classes.

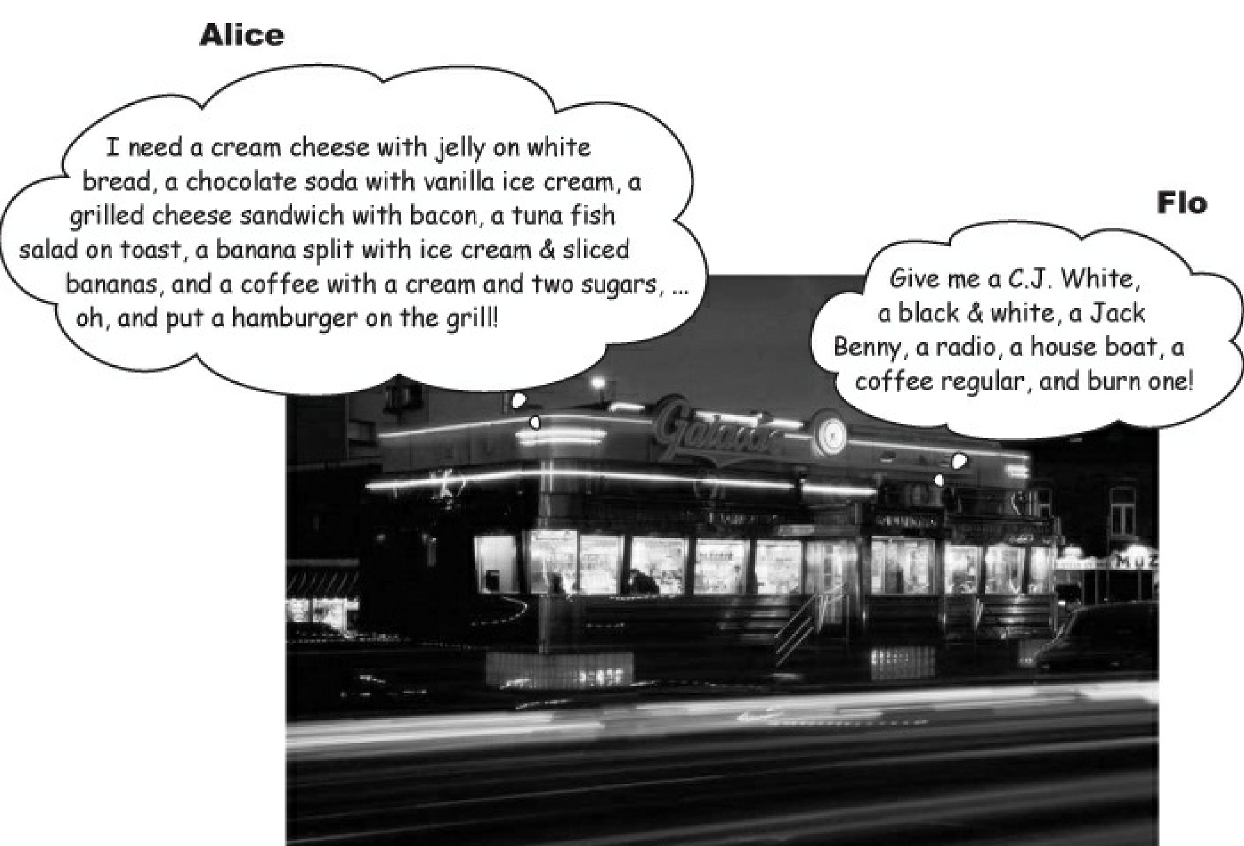
1. Draw this kind of arrow for inheritance ("extends").
2. Draw this kind of arrow for interface ("implements").

3. Draw this kind of arrow for “HAS-A”. ⟶

④ Put the method setWeapon() into the right class.



Overheard at the local diner...

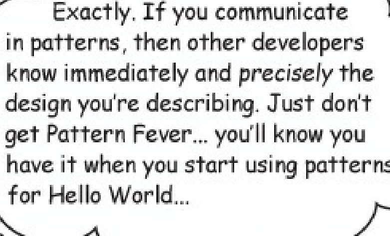


What's the difference between these two orders? Not a thing! They're both the same order, except Alice is using twice the number of words and trying the patience of a grumpy short-order cook.

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

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

Overheard in the next cubicle...



BRAIN POWER

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

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

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.

NOTE

"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.

NOTE

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.

NOTE

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.

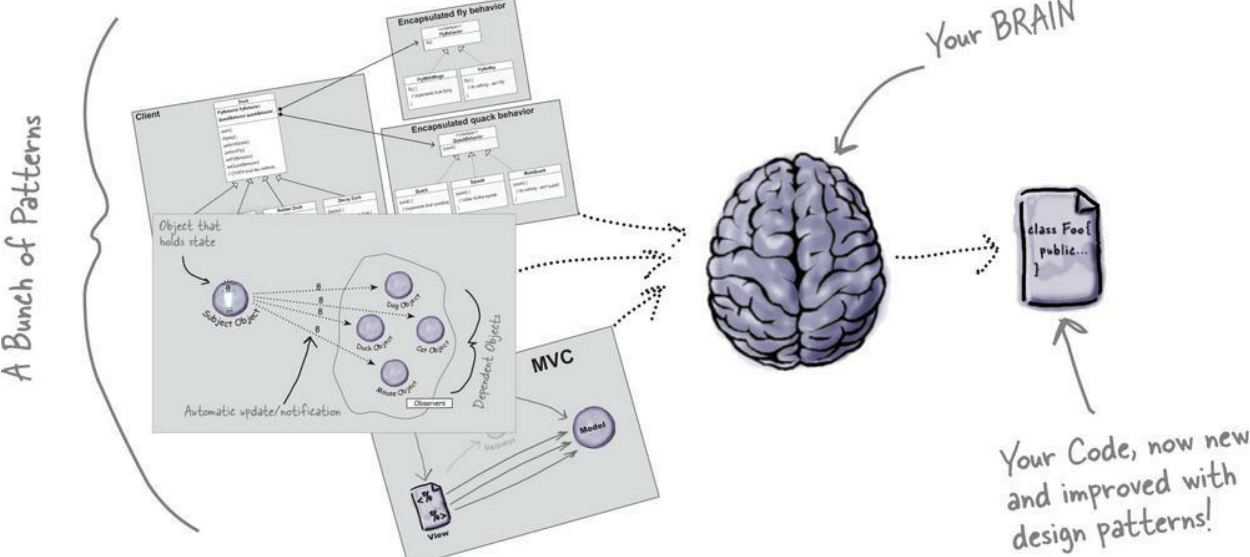
NOTE

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

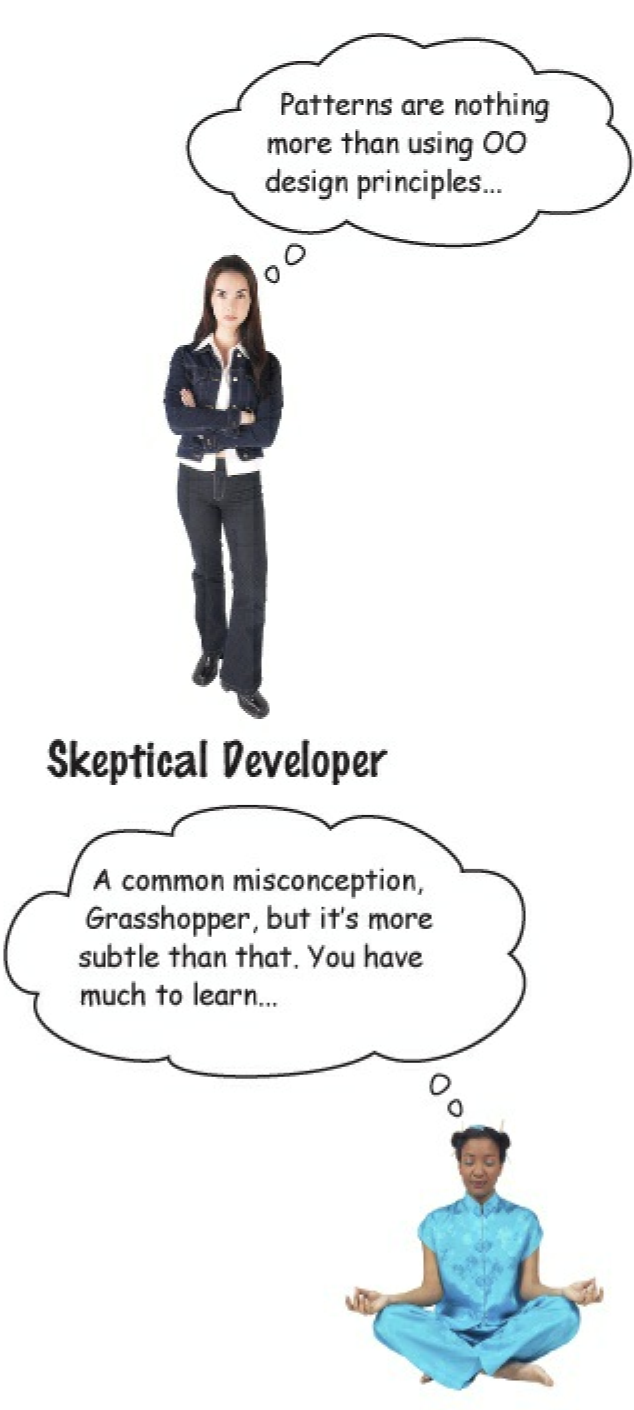
A: **A:** Design patterns are higher level than libraries. Design patterns tell us how to structure classes and objects to solve certain problems and it is our job to adapt those designs to fit our particular application.

Q: **Q: Aren’t libraries and frameworks also design patterns?**

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

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



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?

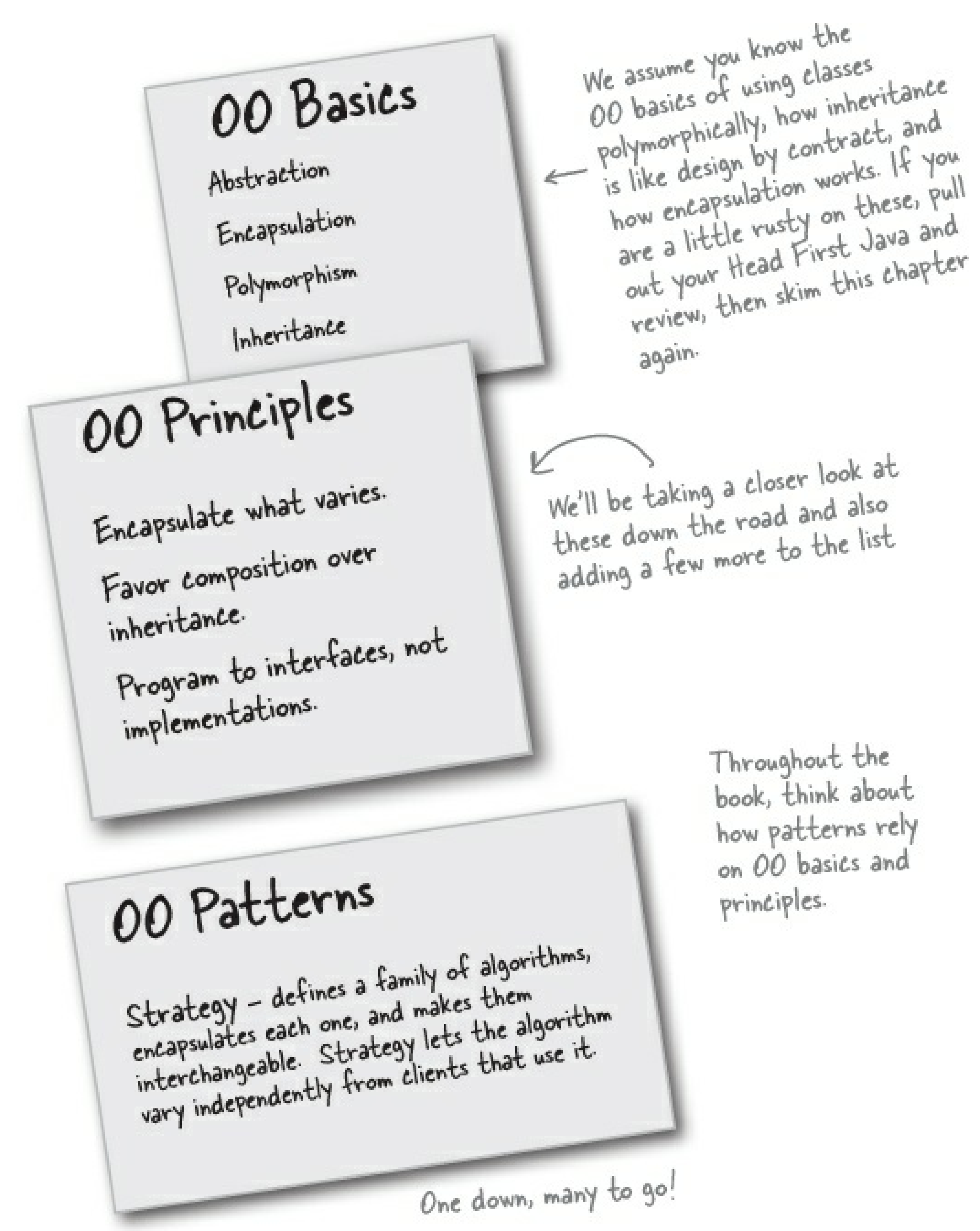
Guru: There are some object-oriented principles that underlie the patterns, and knowing these will help you to cope when you can’t find a pattern that matches your problem.

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.



BULLET POINTS

- Knowing the OO basics does not make you a good OO designer.
- Good OO designs are reusable, extensible, and maintainable.
- Patterns show you how to build systems with good OO design qualities.
- Patterns are proven object-oriented experience.
- Patterns don't give you code, they give you general solutions to design problems. You apply them to your specific application.

- Patterns aren't *invented*, they are *discovered*.
- Most patterns and principles address issues of *change* in software.
- Most patterns allow some part of a system to vary independently of all other parts.
- We often try to take what varies in a system and encapsulate it.
- Patterns provide a shared language that can maximize the value of your communication with other developers.

DESIGN PATTERNS CROSSWORD

Let's give your right brain something to do.

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

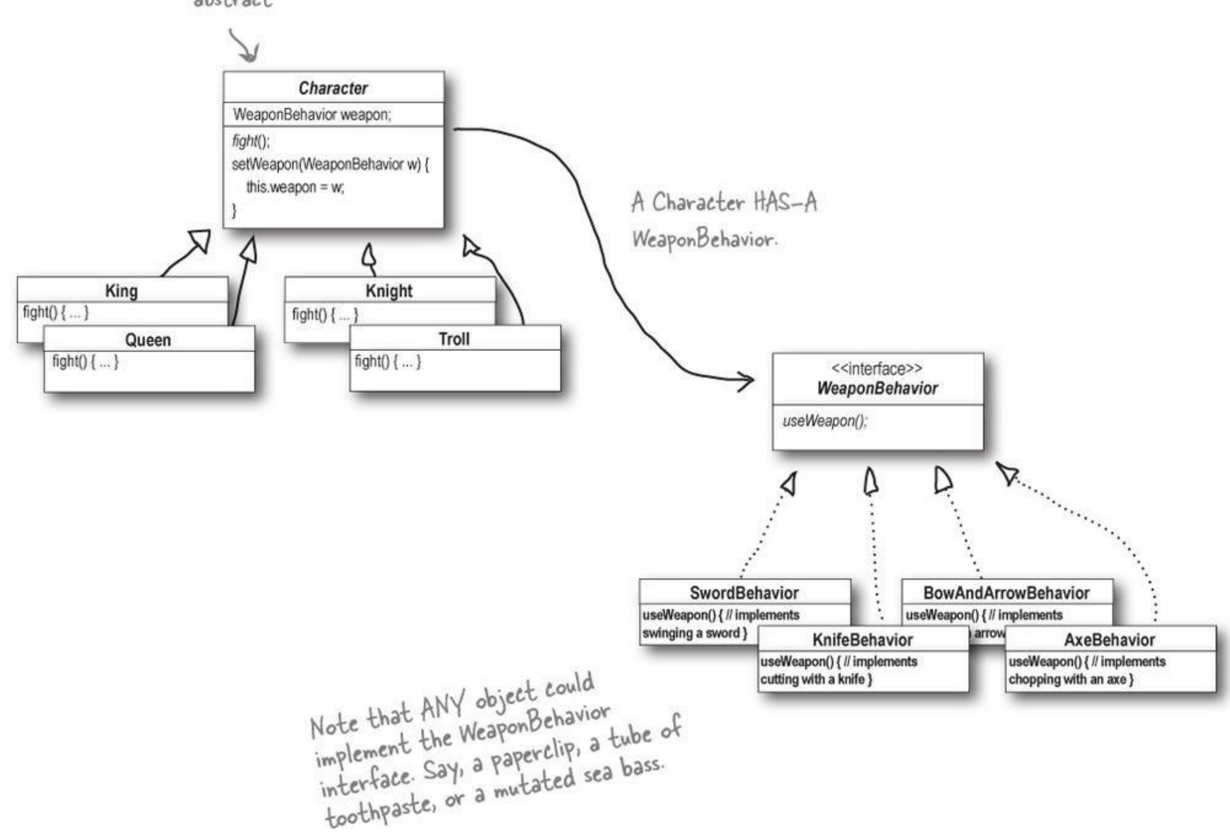
Across	Down
2. ________ what varies. 4. Design patterns __________.	1. Patterns _______ in many applications. 3. Favor this over inheritance.

6. Java IO, Networking, Sound.	5. Dan was thrilled with this pattern.
9. Rubber ducks make a __________.	7. Most patterns follow from OO _________.
13. Bartender thought they were called.	8. Not your own __________.
15. Program to this, not an implementation.	10. High level libraries.
17. Patterns go into your _________.	11. Joe’s favorite drink.
18. Learn from the other guy’s ___________.	12. Pattern that fixed the simulator.
19. Development constant.	13. Duck that can’t quack.
20. Patterns give us a shared ____________.	14. Grilled cheese with bacon.
	15. Duck demo was located here.

DESIGN PUZZLE SOLUTION

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

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



SHARPEN YOUR PENCIL SOLUTION

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

☑	A.	Code is duplicated across subclasses.
☑	B.	Runtime behavior changes are difficult.
❑	C.	We can't make duck's 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.

SHARPEN YOUR PENCIL SOLUTION

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

NOTE

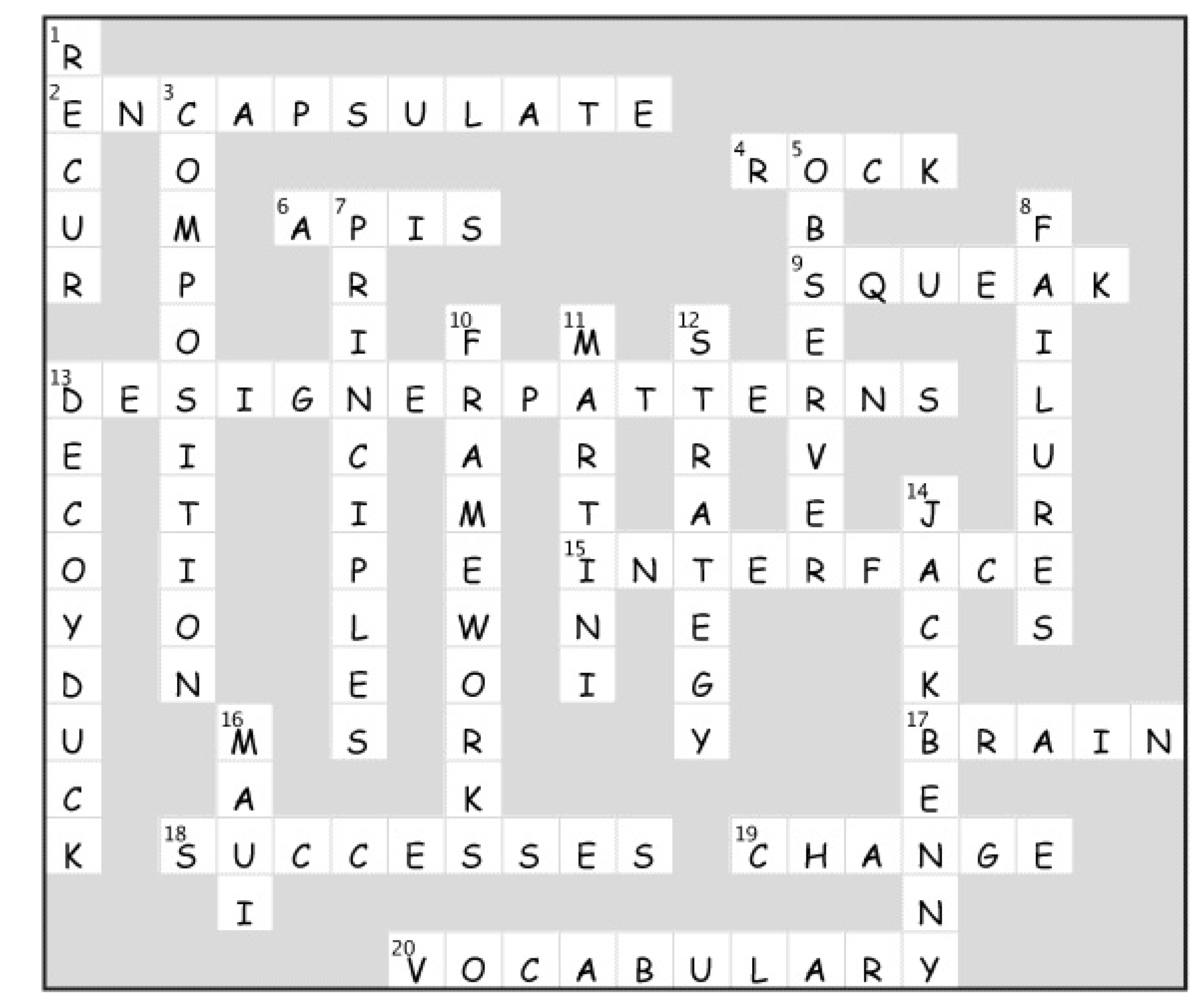
My customers or users decide they want something else, or they want new functionality.

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

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

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

DESIGN PATTERNS CROSSWORD SOLUTION



1 RECUR
2 ENCAPSULATE
3 COMPOSITION
4 ROCK
5 OBSERVER
6 APIS
7 PRINCIPLES
8 FAILURES
9 SQUEAK
10 FRAMEWORK
11 MARTINI
12 STRATEGY
13 DESIGNERPATTERNS
13 DECOYDUCK
14 JACKBENNY
15 INTERFACE
16 MAUI
17 BRAIN
18 SUCCESSES
19 CHANGE
20 VOCABULARY

Chapter 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.

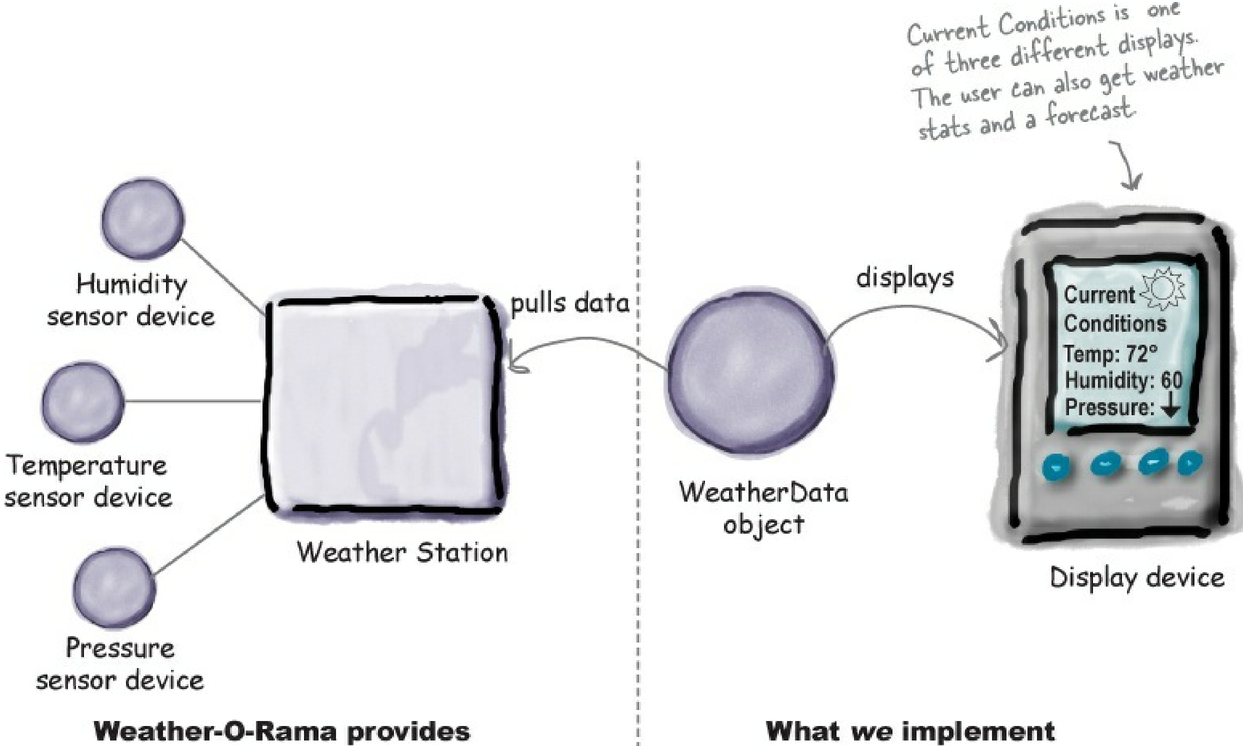
Congratulations!

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 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-ORama 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.

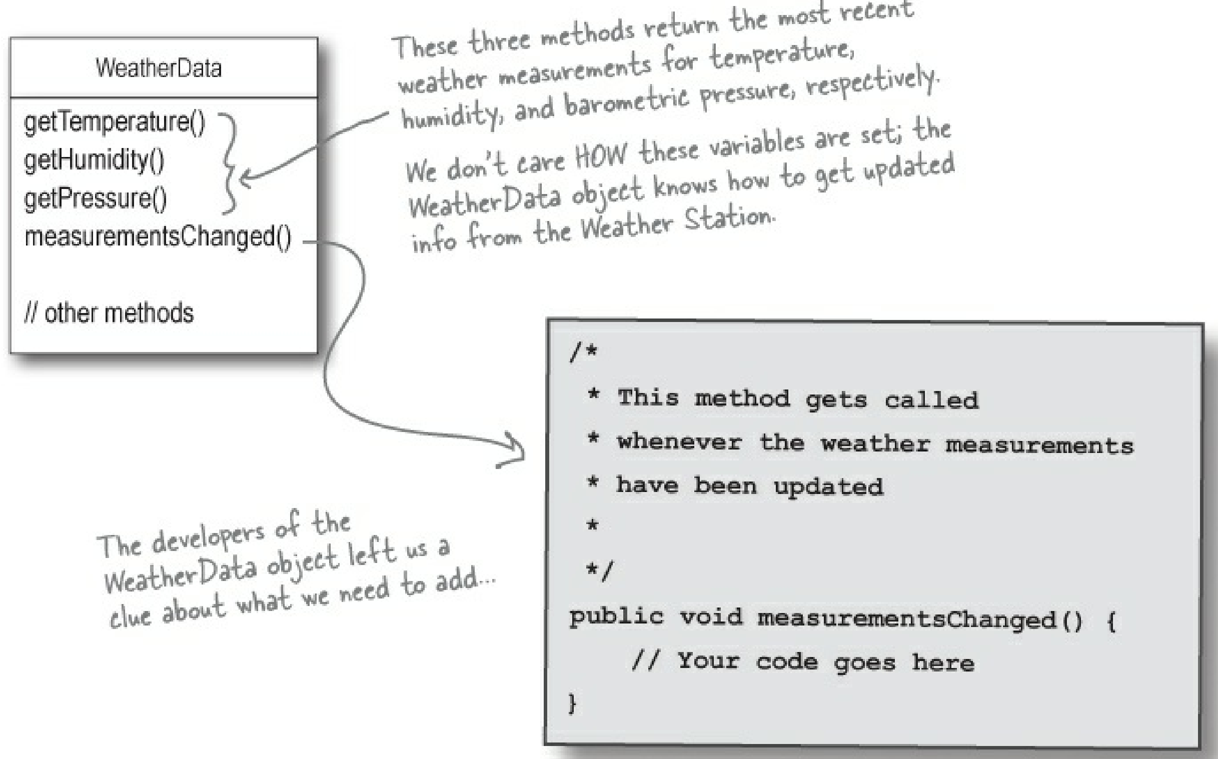


The WeatherData object knows how to talk to the physical Weather Station, to get updated data. The WeatherData object then updates its displays for the three different display elements: Current Conditions (shows temperature, humidity, and pressure), Weather Statistics, and a simple forecast.

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. When we peek inside the code, things look pretty straightforward:

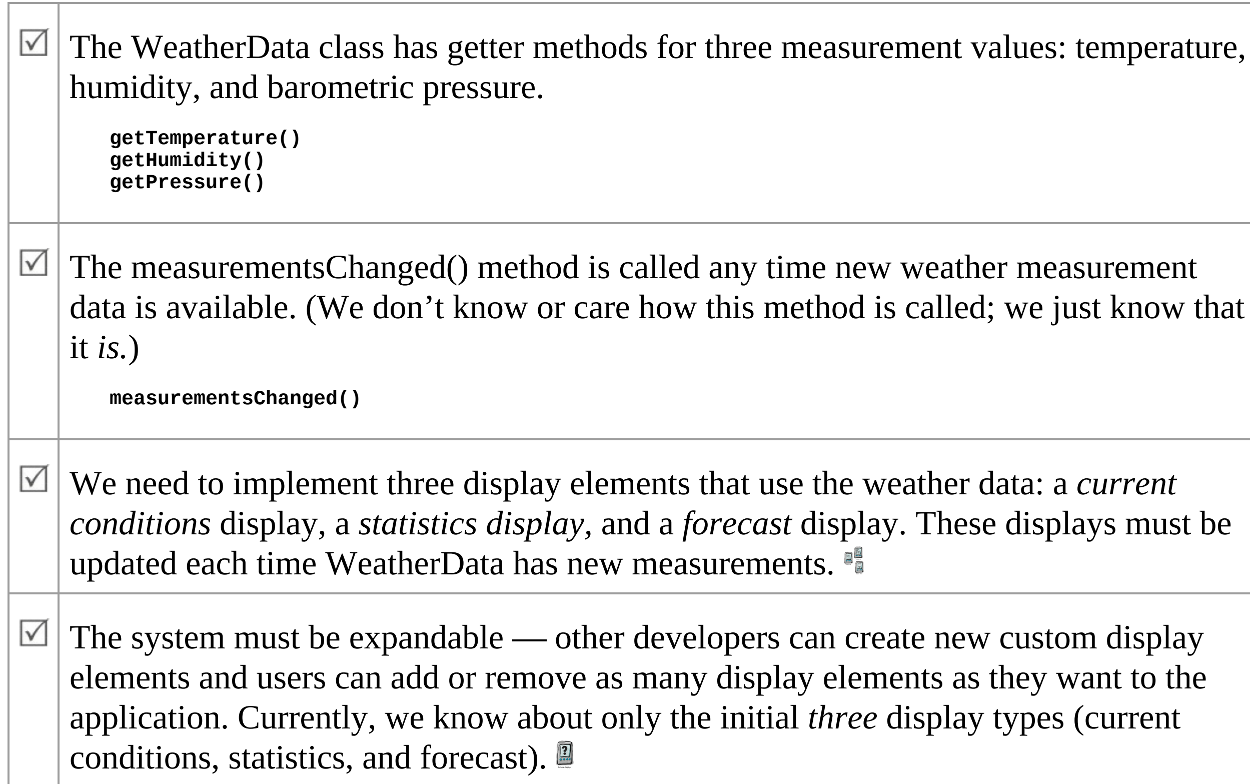


Display device

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.
☑	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).

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 measurements 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

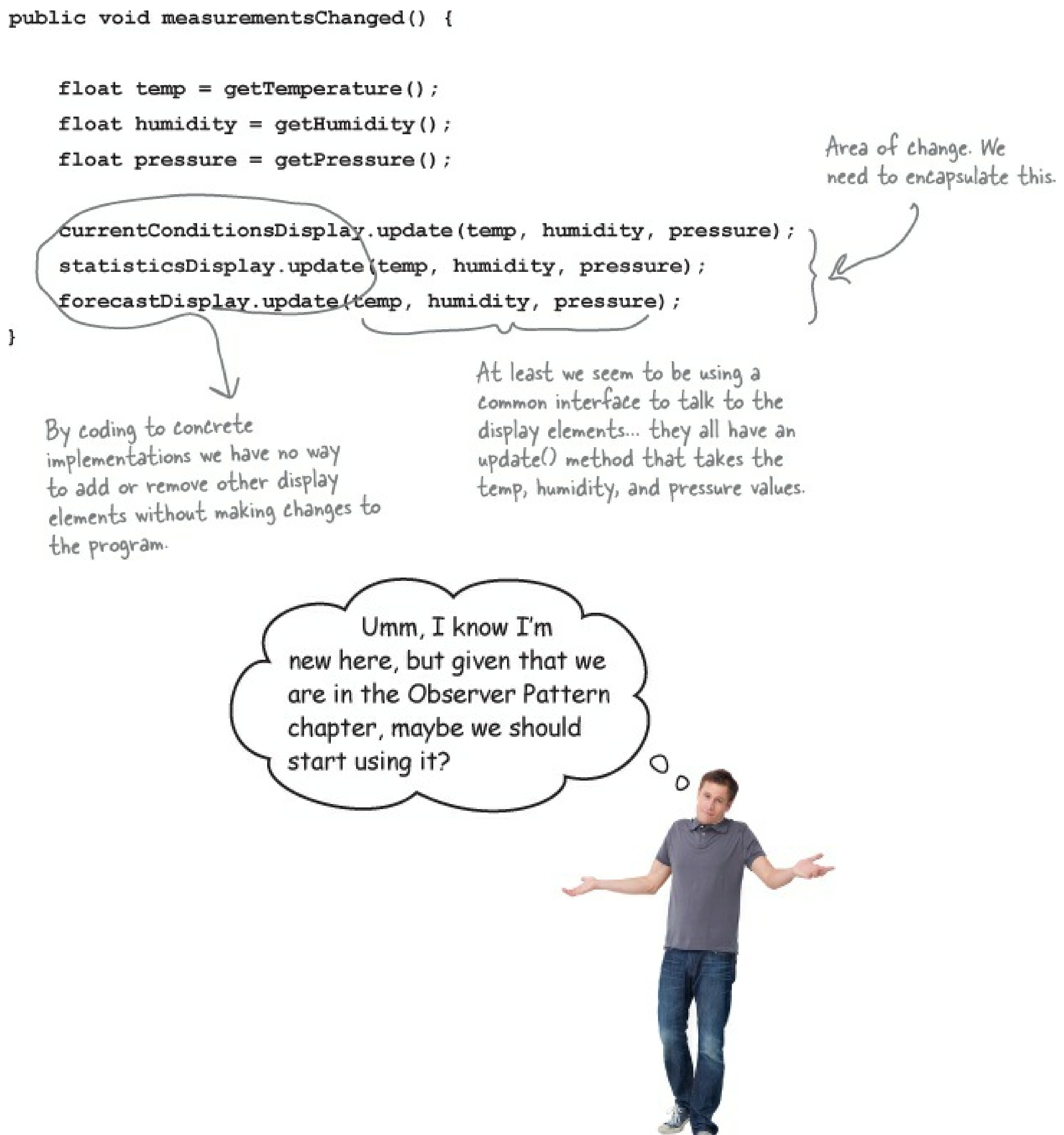
Based on our first implementation, which of the following apply? (Choose all that apply.)

❑	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...



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:

The observers have subscribed to (registered with) the Subject to receive updates when the Subject's data changes.

When data in the Subject changes, the observers are notified.

Subject object manages some bit of data.

2

int

Subject Object

2

2

2

Dog Object

Cat Object

Mouse Object

Observer Objects

New data values are communicated to the observers in some form when they change.

Duck Object

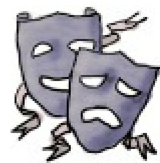
This object isn't an observer, so it doesn't get notified when the Subject's data changes.

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...

This is Lori. I'm looking for a Java development position. I've got five years of experience and...
Uh, yeah, you and everybody else, baby. I'm putting you on my list of Java developers. Don't call me, I'll call you!
Hi, I'm Jill. I've written a lot of EJB systems. I'm interested in any job you've got with Java development.
I'll add you to the list– you'll know along with everyone else.
1
Software Developer #1
2
Headhunter/Subject
3
Software Developer #2
4
Subject

Two weeks later...

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

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

The Observer Pattern defined

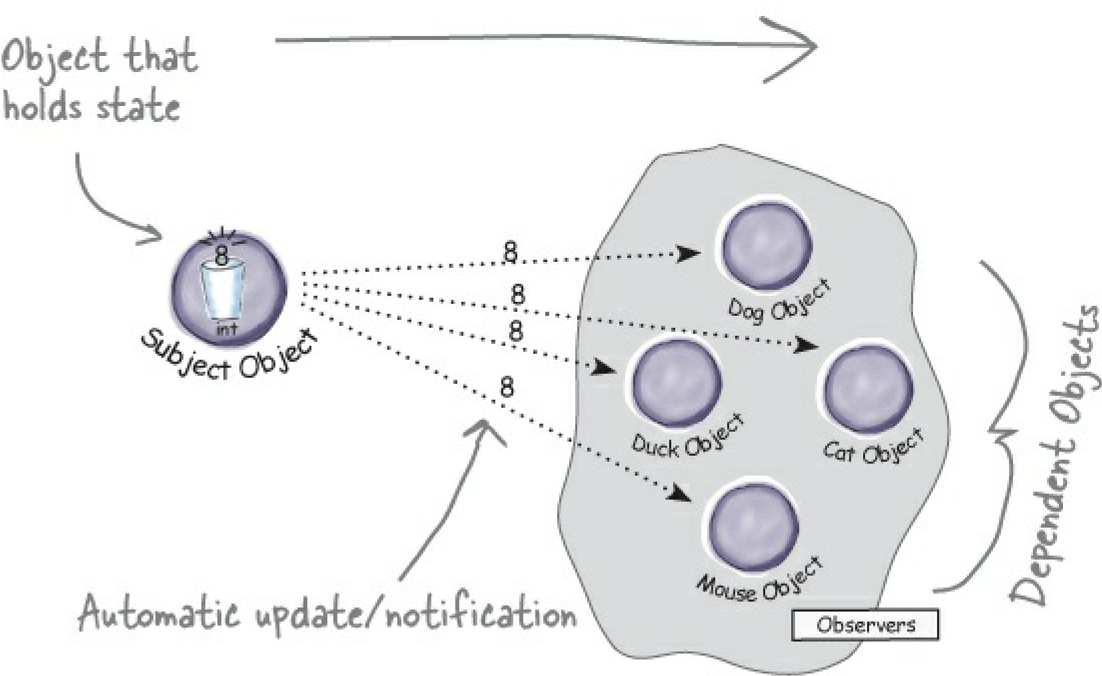
When you’re trying to picture the Observer Pattern, a newspaper subscription service with its publisher and subscribers is a good way to visualize the pattern.

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

NOTE

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:



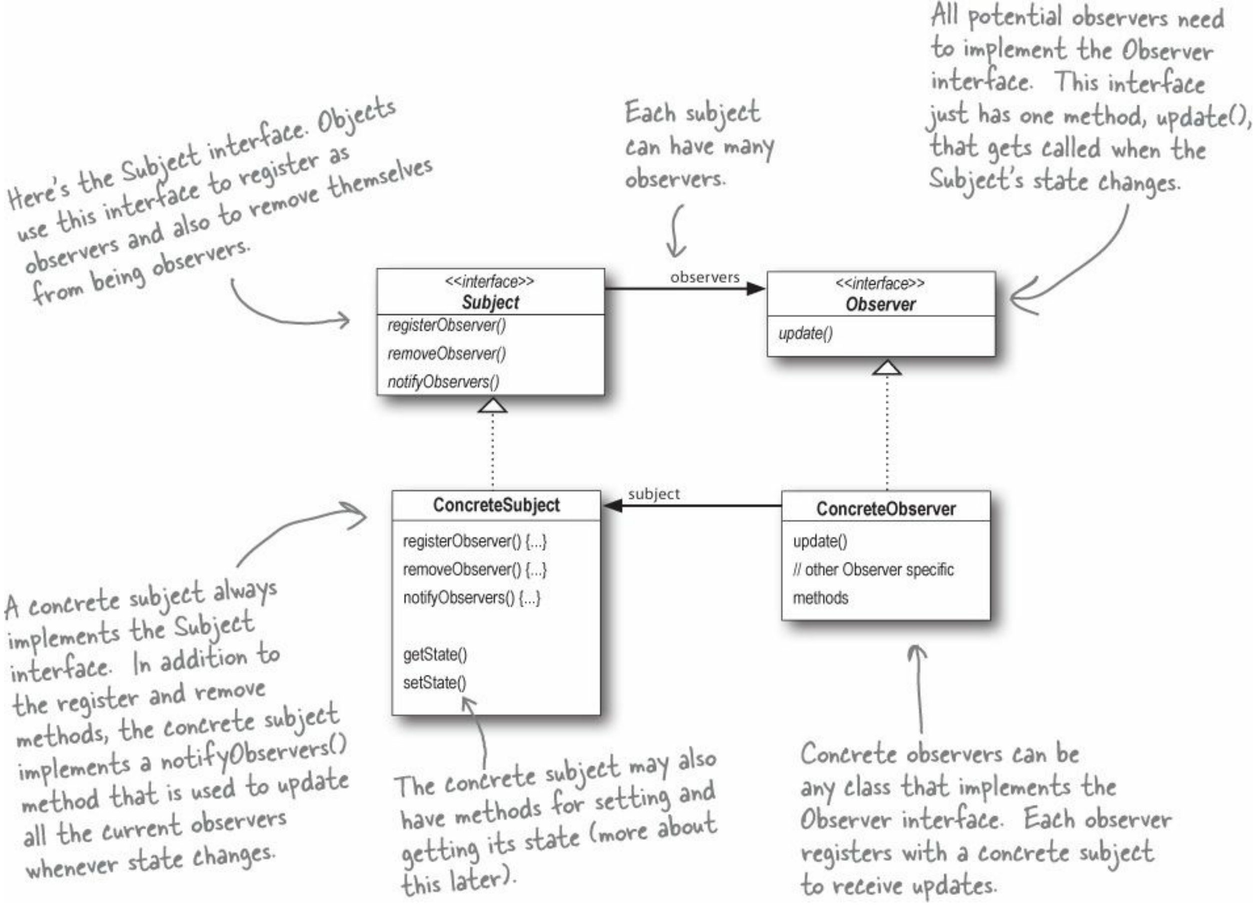
The Observer Pattern defines a one-to-many relationship between a set of objects. When the state of one object changes, all of its dependents are notified.

The 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 defined: the class diagram



THERE ARE NO DUMB QUESTIONS

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

A: **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: **Q: How does dependence come into this?**

A: **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.

NOTE

How many different kinds of change can you identify here?

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.

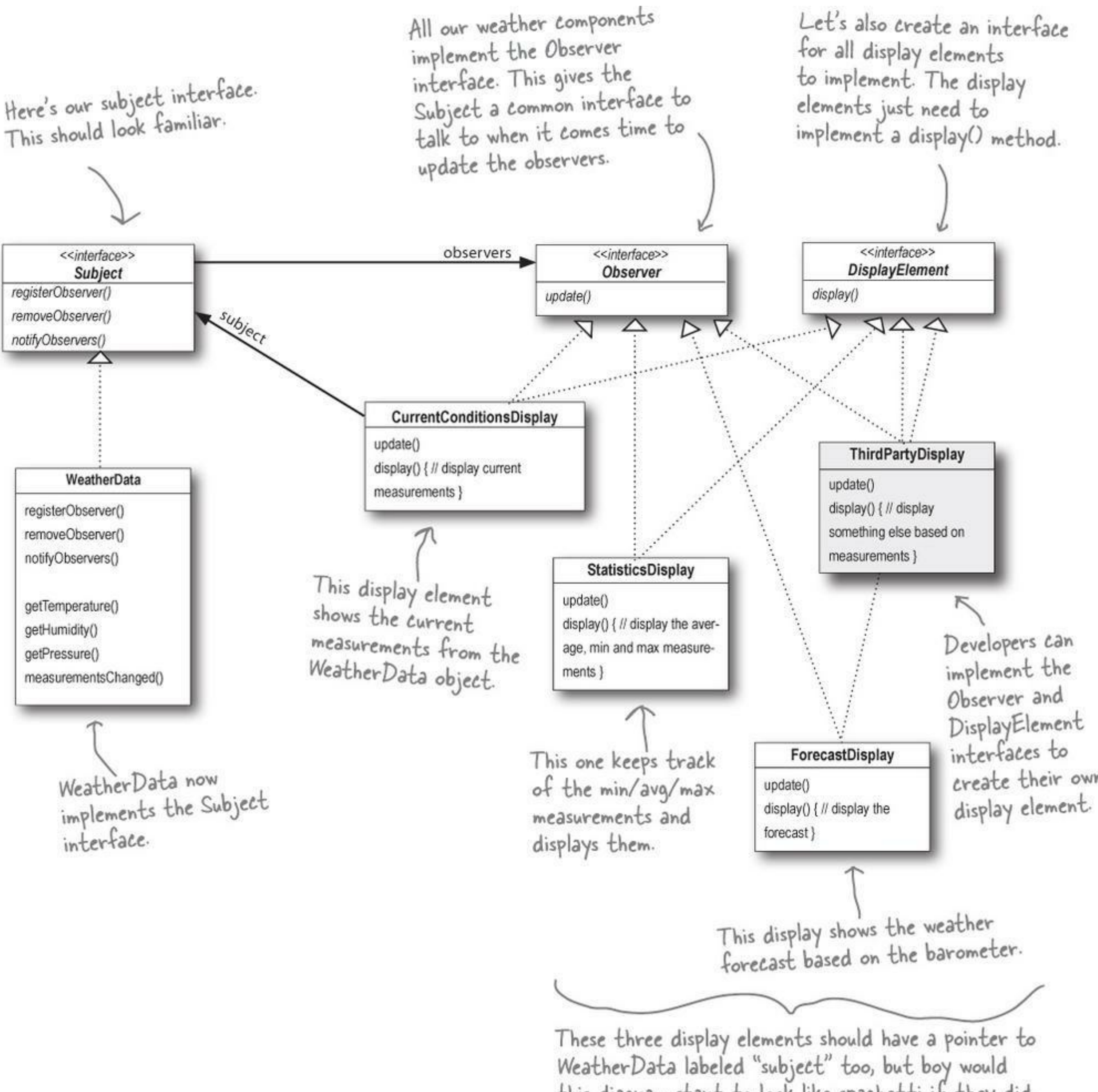
Sue: We gotta remember that every display element can be different... so I think that’s where having a common interface comes in. Even though every component has a different type, they should all implement the same interface so that the WeatherData object will know how to send them the measurements.

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.

BRAIN POWER

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

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

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...

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

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

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

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

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

    public void measurementsChanged() {
        notifyObservers();
    }

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

    // other WeatherData methods here
}
```

WeatherData now implements the Subject interface.

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

Here we implement the Subject interface.

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.

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

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

Now, let's build those display elements

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

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

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

A: **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 and they can’t possibly ship without a Heat Index display element. Here are the details.

The heat index is an index that combines temperature and humidity to determine the apparent temperature (how hot it actually feels). To compute the heat index, you take the

temperature, T, and the relative humidity, RH, and use this formula:

heatindex =

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

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.

NOTE

You can get heatindex.txt from wickedlysmart.com.

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

When you finish, your output should look like this:

```
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
%
```

Here's what changed in this output.

FIRESIDE CHATS

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

Subject:	Observer:

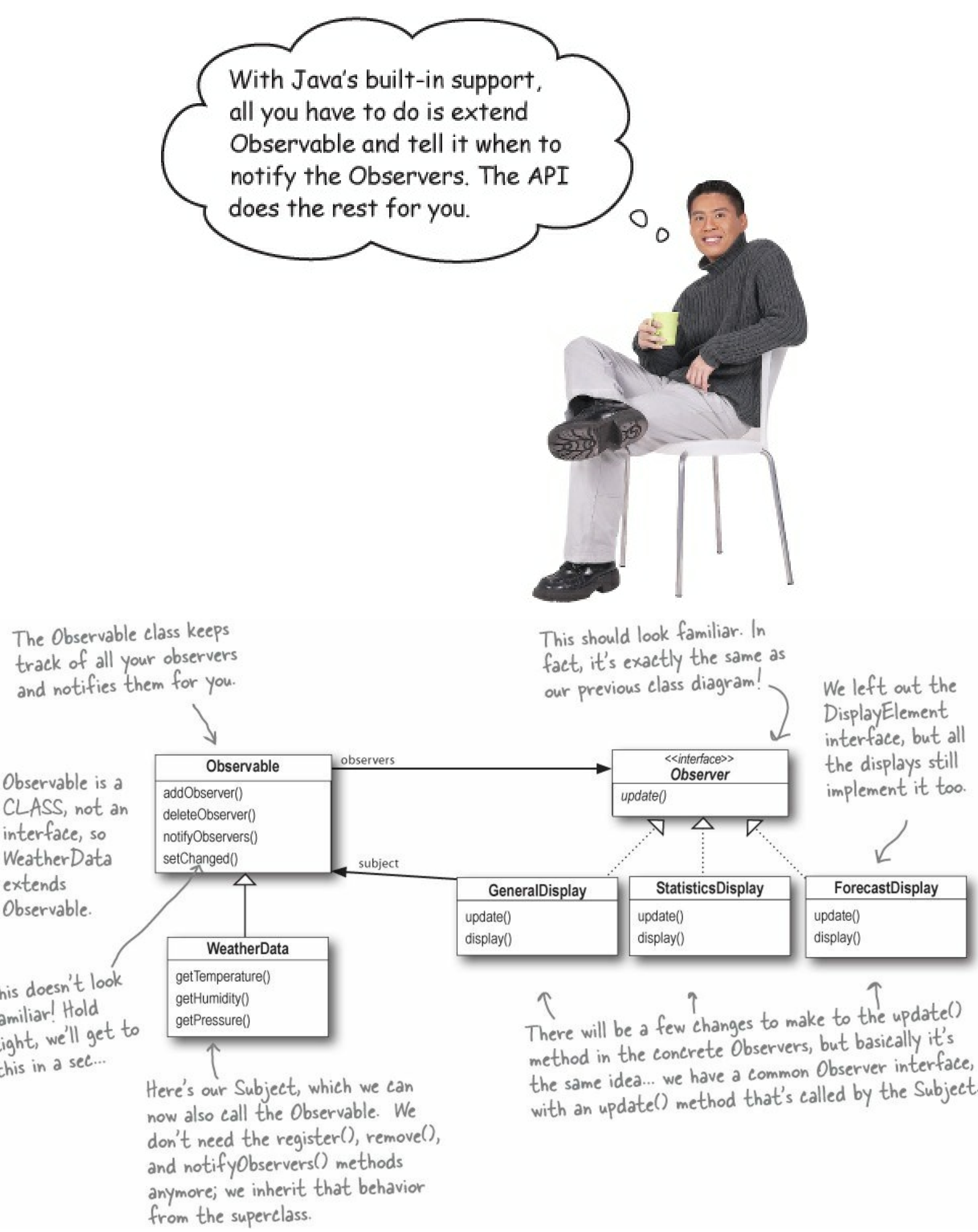
I’m glad we’re finally getting a chance to chat in person.	
	Really? I thought you didn’t care much about us Observers.
Well, I do my job, don’t I? I always tell you what’s going on... Just because I don’t really know who you are doesn’t mean I don’t care. And besides, I do know the most important thing about you — you implement the Observer interface.	
	Yeah, but that’s just a small part of who I am. Anyway, I know a lot more about you...
Oh yeah, like what?	
	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...
Well, excuuuse me. I have to send my state with my notifications so all you lazy Observers will know what happened!	
	Okay, 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?
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.	
	Why don’t you just write some public getter methods that will let us pull out the state we need?
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.	
	Don't be so pushy! There are so many different kinds of us Observers, there's no way you can anticipate everything we need. Just let us come to you to get the state we need. That way, if some of us only need a little bit of state, we aren't forced to get it all. It also makes things easier to modify later. Say, for example, you expand yourself and add some more state. If you use pull, you don't have to go around and change the update calls on every observer; you just need to change yourself to allow more getter methods to access our additional state.
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.	
	Oh really? I think we're going to look at that next....
Great... maybe I'll get to see a good example of pull and change my mind.	
	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 interfaces, 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:



How Java's built-in Observer Pattern works

The built-in Observer Pattern works a bit differently than the implementation

that we used on the Weather Station. The most obvious difference is that WeatherData (our subject) now extends the Observable class and inherits the add, delete, and notify Observer methods (among a few others). Here's how we use Java's version:

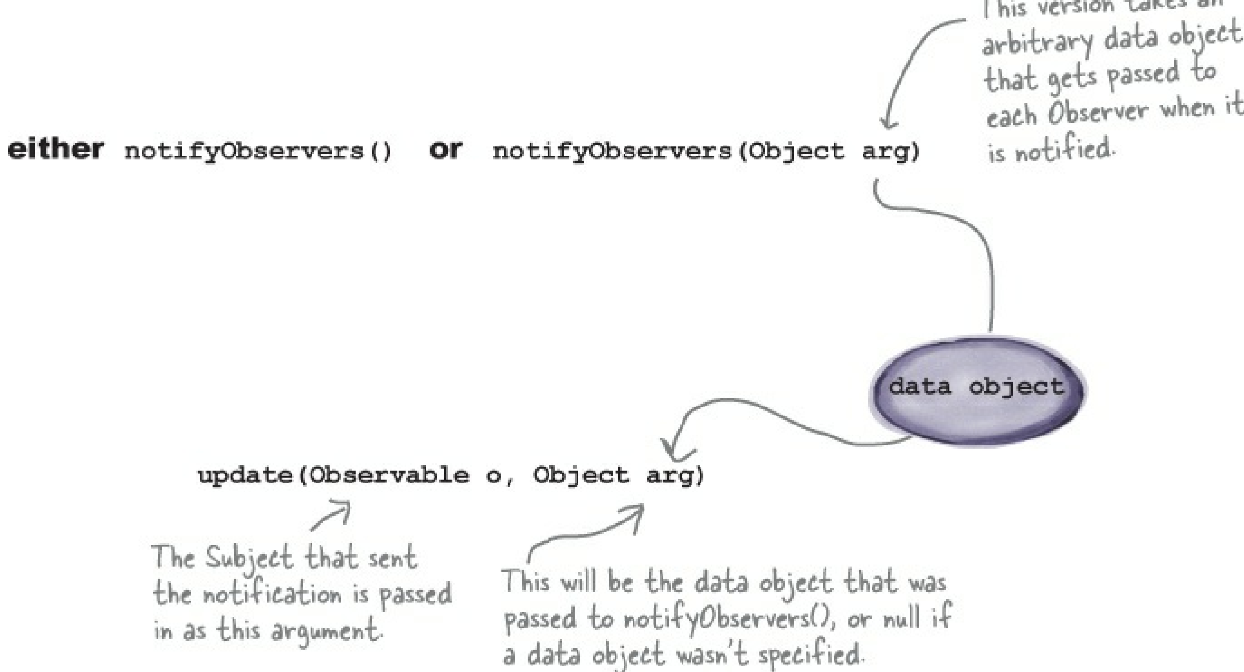
For an Object to become an observer...

As usual, implement the Observer interface (this time the java.util.Observer interface) and call addObserver() on any Observable object. Likewise, to remove yourself as an observer, just call deleteObserver().

For the Observable to send notifications...

First of all you need to be Observable by extending the java.util.Observable superclass. From there it is a two-step process:

① You first must call the setChanged() method to signify that the state has changed in your object.
② Then, call one of two notifyObservers() methods:



For an Observer to receive notifications...

It implements the update method, as before, but the signature of the method is a bit different:

If you want to "push" data to the observers, you can pass the data as a data object to the notifyObservers(arg) method. If not, then the Observer has to

“pull” the data it wants from the Observable object passed to it. How? Let’s rework the Weather Station and you’ll see.

The setChanged() method is used to signify that the state has changed and that notifyObservers(), when it is called, should update its observers. If notifyObservers() is called without first calling setChanged(), the observers will NOT be notified. Let’s take a look behind the scenes of Observable to see how this works:

BEHIND THE SCENES

```
setChanged() {
   changed = true
}

notifyObservers(Object arg) {
   if (changed) {
      for every observer on the list {
         call update (this, arg)
      }
      changed = false
   }
}

notifyObservers() {
   notifyObservers(null)
}
```

The setChanged() method sets a changed flag to true.

notifyObservers() only notifies its observers if the changed flag is TRUE.

And after it notifies the observers, it sets the changed flag back to false.

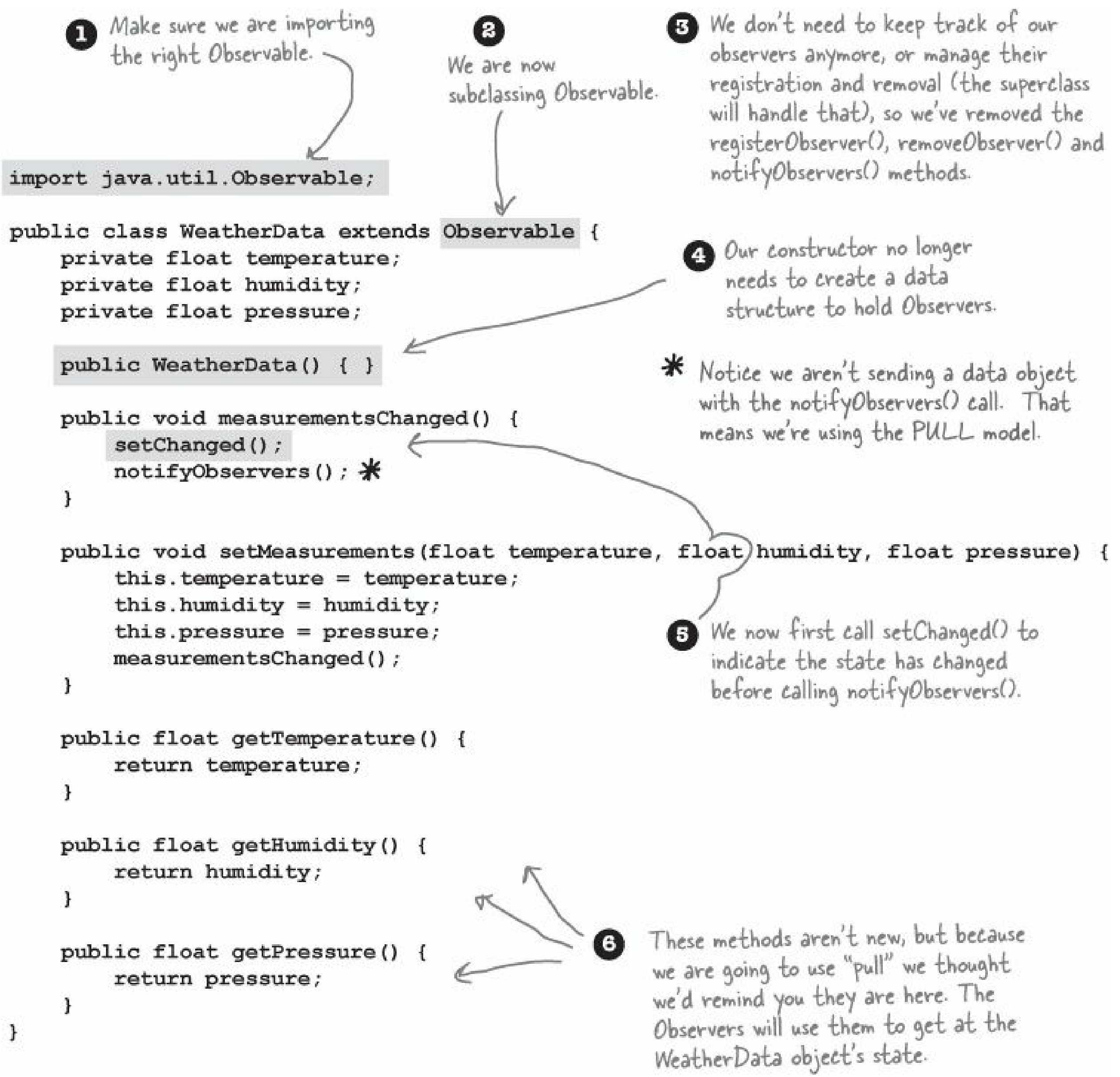
Pseudocode for the Observable class.

Why is this necessary? The setChanged() method is meant to give you more flexibility in how you update observers by allowing you to optimize the notifications. For example, in our Weather Station, imagine if our measurements were so sensitive that the temperature readings were constantly fluctuating by a few tenths of a degree. That might cause the WeatherData object to send out notifications constantly. Instead, we might want to send out notifications only if the temperature changes more than half a degree and we could call setChanged() only after that happened.

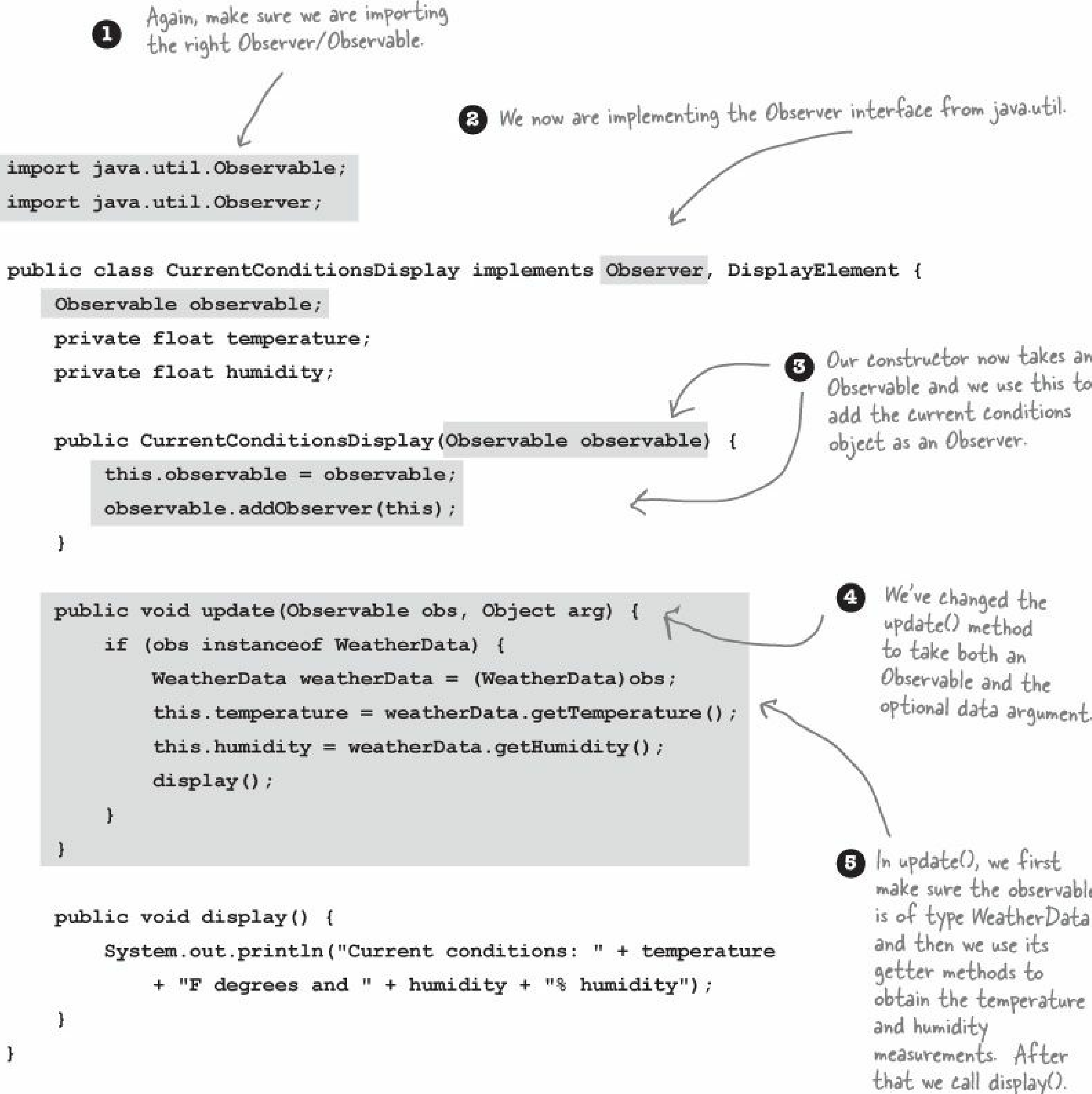
You might not use this functionality very often, but it's there if you need it. In either case, you need to call setChanged() for notifications to work. If this functionality is something that is useful to you, you may also want to use the clearChanged() method, which sets the changed state back to false, and the hasChanged() method, which tells you the current state of the changed flag.

Reworking the Weather Station with the built-in support

First, let's rework WeatherData to use java.util.Observable



Now, let's rework the CurrentConditionsDisplay



CODE MAGNETS

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

```
public ForecastDisplay(Observable
observable) {
```

```
display();
```

```
observable.addObserver(this);
```

```
if (observable instanceof WeatherData) {
```

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

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

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

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

```
WeatherData weatherData = (WeatherData)observable;
```

```
public void update(Observable observable,
    Object arg) {
```

```
import java.util.Observable;
import java.util.Observer;
```

Running the new code

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

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

Hmm, do you notice anything different? Look again...

You'll see all the same calculations, but mysteriously, the order of the text output is different. Why might this happen? Think for a minute before reading on...

Never depend on order of evaluation of the Observer notifications

The java.util.Observable has implemented its notifyObservers() method such that the Observers are notified in a *different* order than our own implementation. Who's right? Neither; we just chose to implement things in different ways.

What would be incorrect, however, is if we wrote our code to *depend* on a specific notification order. Why? Because if you need to change Observable/Observer implementations, the order of notification could change and your application would produce incorrect results. Now that's definitely not what we'd consider loosely coupled.

The dark side of java.util.Observable

Yes, good catch. As you've noticed, Observable is a class, not an *interface*, and worse, it doesn't even *implement* an interface. Unfortunately, the java.util.Observable implementation has a number of problems that limit its usefulness and reuse. That's not to say it doesn't provide some utility, but there are some large potholes to watch out for.

Observable is a class

You already know from our principles this is a bad idea, but what harm does it really cause?

First, because Observable is a *class*, you have to *subclass* it. That means you can't add on the Observable behavior to an existing class that already extends another superclass. This limits its reuse potential (and isn't that why we are

using patterns in the first place?).

Second, because there isn't an Observable interface, you can't even create your own implementation that plays well with Java's built-in Observer API. Nor do you have the option of swapping out the java.util implementation for another (say, a new, multithreaded implementation).

Observable protects crucial methods

If you look at the Observable API, the setChanged() method is protected. So what? Well, this means you can't call setChanged() unless you've subclassed Observable. This means you can't even create an instance of the Observable class and compose it with your own objects, you *have* to subclass. The design violates a second design principle here...*favor composition over inheritance*.

What to do?

Observable *may* serve your needs if you can extend java.util.Observable. On the other hand, you may need to roll your own implementation as we did at the beginning of the chapter. In either case, you know the Observer Pattern well and you're in a good position to work with any API that makes use of the pattern.

Other places you'll find the Observer Pattern in the JDK

The java.util implementation of Observer/Observable is not the only place you'll find the Observer Pattern in the JDK; both JavaBeans and Swing also provide their own implementations of the pattern. At this point you understand enough about Observer to explore these APIs on your own; however, let's do a quick, simple Swing example just for the fun of it.

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

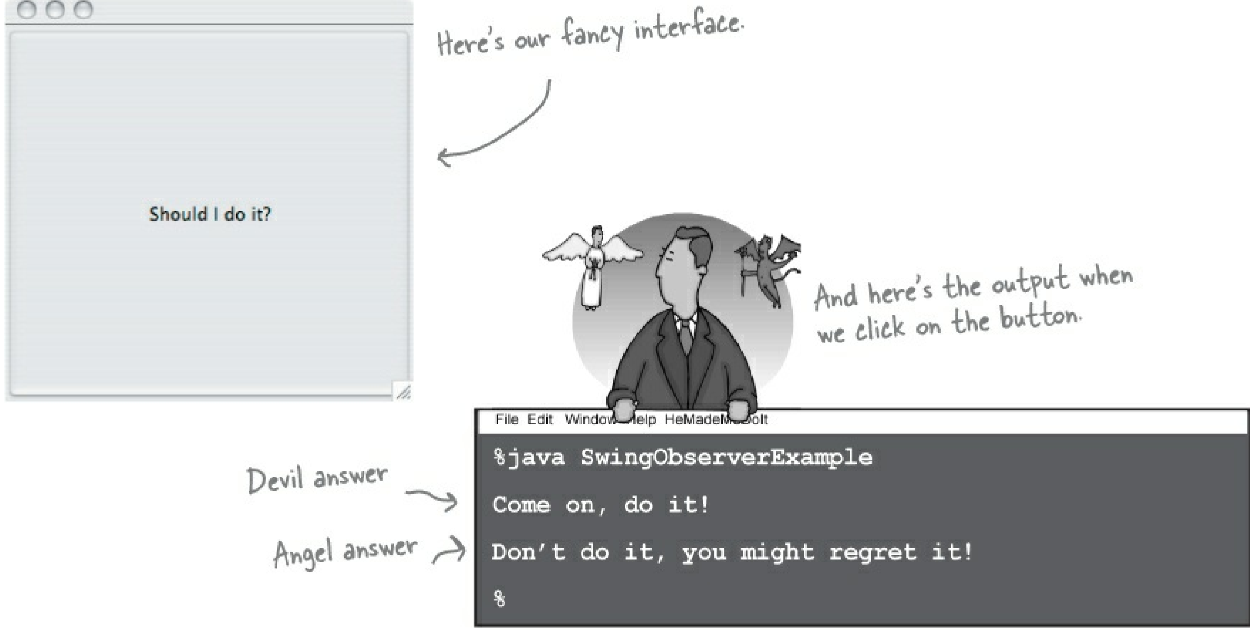
A little background...

Let's take a look at a simple part of the Swing API, the JButton. If you look under the hood at JButton's superclass, AbstractButton, you'll see that it has a lot of add/ remove listener methods. These methods allow you to add and

remove observers, or, as they are called in Swing, listeners, to listen for various types of events that occur on the Swing component. For instance, an ActionListener lets you "listen in" on any types of actions that might occur on a button, like a button press. You'll find various types of listeners all over the Swing API.

A little life-changing application

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



And the code...

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

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

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

        // Set frame properties here
    }

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

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

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

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

Code to set up the frame goes here.

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

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

NOTE

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

Yes, you're still using the Observer Pattern. By using a lambda expression rather than an inner class, you're just skipping the step of creating an ActionListener object. With a lambda expression, you create a function object instead, and this function object is the observer. When you pass that function object to addActionListener(), Java ensures its signature matches actionPerformed(), the one method in the ActionListener interface.

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

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

The updated code, using lambda expressions

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

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

        // Set frame properties here
    }
}
```

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

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

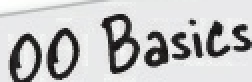
Using lambda expressions makes this code a lot more concise.

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

For more on lambda expressions, check out the Java docs, and Chapter 6.

Tools for your Design Toolbox

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



OO Basics

Abstraction

tion

nism

ce

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

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

OO Patterns

Stra

encap

inter

vary

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

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

BULLET POINTS

- The Observer Pattern defines a one-to-many relationship between objects.
- Subjects, or as we also know them, Observables, update Observers using a common interface.
- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement the Observer interface.
- You can push or pull data from the Observable when using the pattern (pull is considered more “correct”).
- Don’t depend on a specific order of notification for your Observers.
- Java has several implementations of the Observer Pattern, including the general purpose java.util.Observable.
- Watch out for issues with the java.util.Observable implementation.
- Don’t be afraid to create your own Observable implementation if needed.
- Swing makes heavy use of the Observer Pattern, as do many GUI frameworks.
- You’ll also find the pattern in many other places, including JavaBeans and RMI.

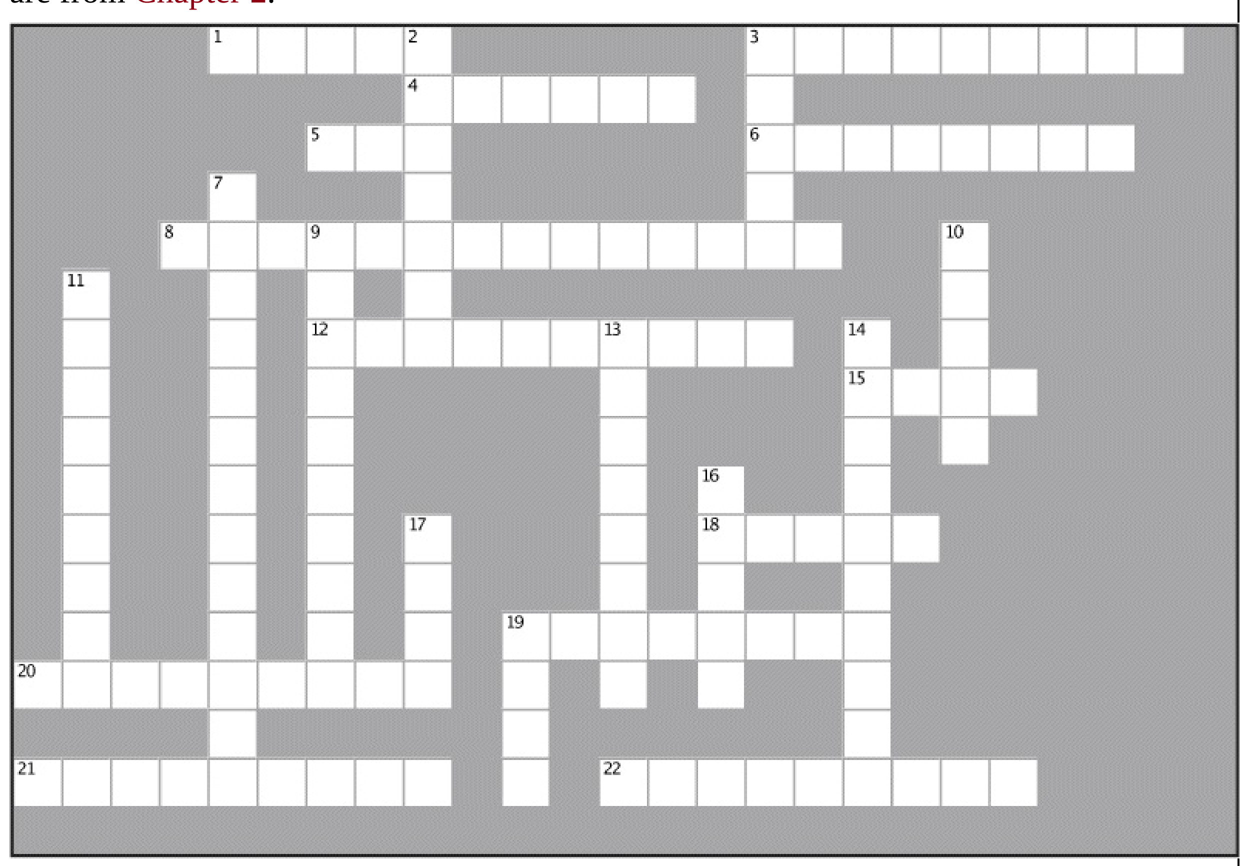
DESIGN PRINCIPLE CHALLENGE

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

DESIGN PRINCIPLE *Identify the aspects of your application that vary and separate them from what stays the same.*	
DESIGN PRINCIPLE *Program to an interface, not an implementation.*	
DESIGN PRINCIPLE *Favor composition over inheritance.*	This is a hard one, hint: think about how observers and subjects work together.

DESIGN PATTERNS CROSSWORD

Time to give your right brain something to do again! This time all of the solution words are from Chapter 2.



Across	Down
1. Observable is a ___________, not an interface. 3. Devil and Angel are _________ to the button. 4. Implement this method to get notified. 5. Jill got one of her own. 6. CurrentConditionsDisplay implements this interface. 8. How to get yourself off the Observer list. 12. You forgot this if you're not getting notified when you think you should be. 15. One Subject likes to talk to _______ observers. 18. Don't count on this for notification. 19. Temperature, humidity and __________.	2. Ron was both an Observer and a _________. 3. You want to keep your coupling _________. 7. He says you should go for it. 9. _________ can manage your observers for you. 10. Java framework with lots of Observers. 11. Weather-O-Rama's CEO named after this kind of storm. 13. Observers like to be ___________ when something new happens. 14. The WeatherData class __________ the

20. Observers are __________ on the Subject.	Subject interface.
21. Program to an _________ not an implementation.	16. He didn't want any more ints, so he removed himself.
22. A Subject is similar to a __________.	17. CEO almost forgot the ________ index display
	19. Subject initially wanted to _________ all the data to Observer.

SHARPEN YOUR PENCIL SOLUTION

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

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

DESIGN PRINCIPLE CHALLENGE SOLUTION

DESIGN PRINCIPLE *Identify the aspects of your application that vary and separate them from what stays the same.*	The thing that varies in the Observer Pattern is the state of the Subject and the number and types of Observers. With this pattern, you can vary the objects that are dependent on the state of the Subject, without having to change that Subject. That's called planning ahead!
DESIGN PRINCIPLE *Program to an interface, not an implementation.*	Both the Subject and Observer use interfaces. The Subject keeps track of objects implementing the Observer interface, while the observers register with, and get notified by, the Subject

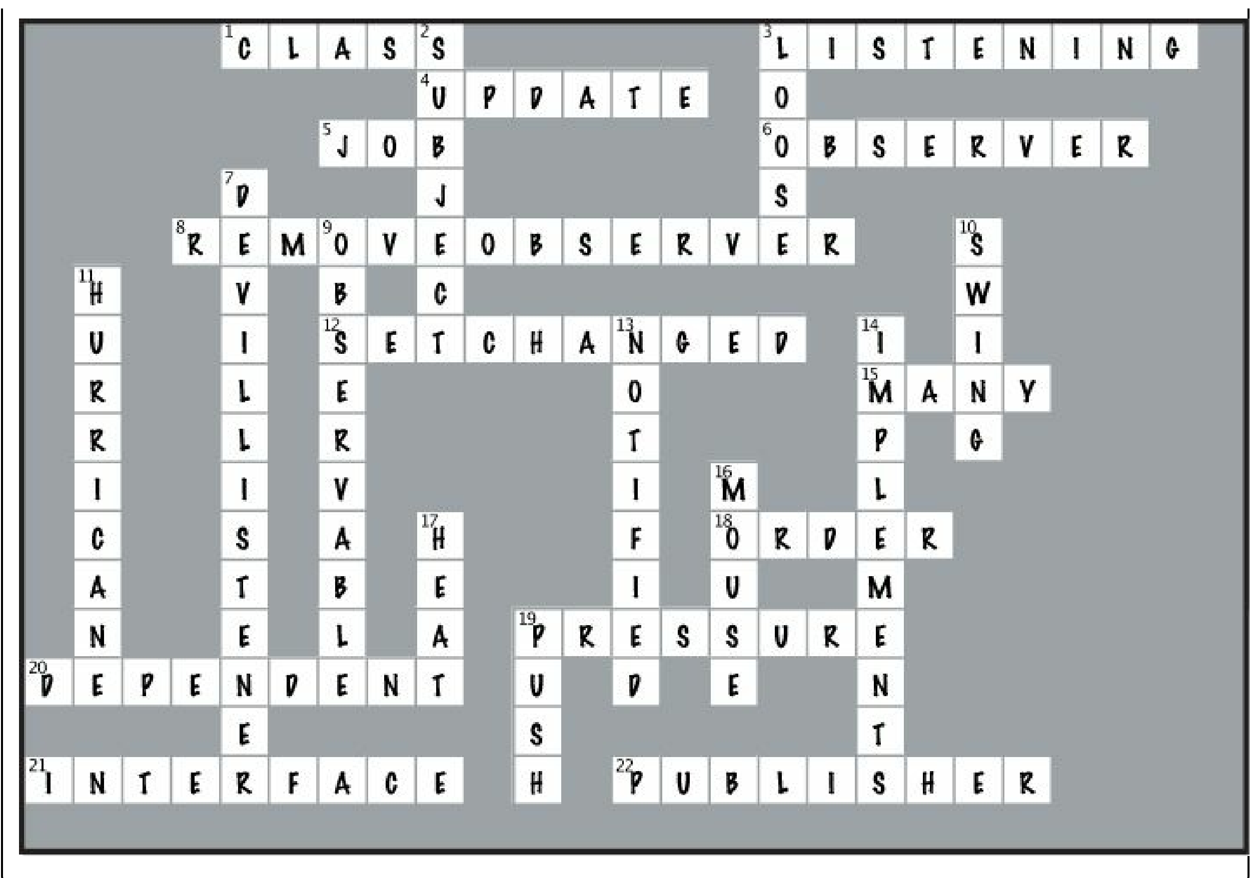
	interface. As we've seen, this keeps things nice and loosely coupled.
DESIGN PRINCIPLE *Favor composition over inheritance.*	The Observer Pattern uses composition to compose any number of Observers with their Subjects. These relationships aren't set up by some kind of inheritance hierarchy. No, they are set up at runtime by composition!

CODE MAGNETS SOLUTION

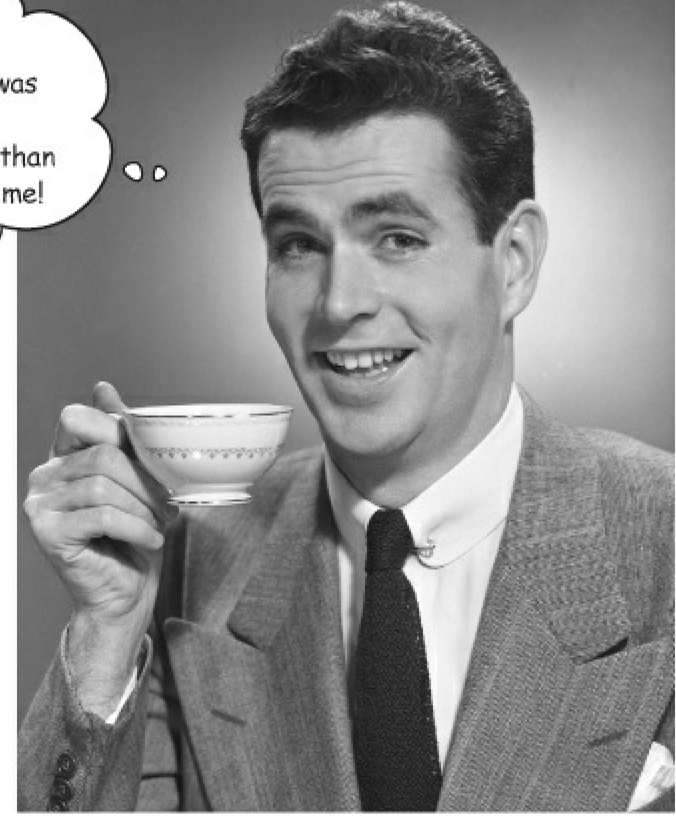
The ForecastDisplay class is all scrambled up on the fridge. Can you reconstruct the code snippets to make it work? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need! Here's our solution.

```java
import java.util.Observable;
import java.util.Observer;

public class ForecastDisplay implements
Observer, DisplayElement {

    private float currentPressure = 29.92f;
    private float lastPressure;

    public ForecastDisplay(Observable
    observable) {

        WeatherData weatherData =
          (WeatherData)observable;

        observable.addObserver(this);

    }

    public void update(Observable observable,
        Object arg) {

        if (observable instanceof WeatherData) {

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

            display();

        }

    }

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

}
```

DESIGN PATTERNS CROSSWORD SOLUTION


1 CLASS
2 SUBJECT
3 LISTENING
3 LOOSE
4 UPDATE
5 JOB
6 OBSERVER
7 DEVILLISTENENE
8 REMOVEOBSERVER
9 OBSERVABLE
10 SWING
11 HURRICANE
12 SETCHANGED
13 NOTIFIED
14 IMPLEMENT
15 MANY
16 MOUSE
17 HEAT
18 ORDER
19 PRESSURE
19 PUSH
20 DEPENDENT
21 INTERFACE
22 PUBLISHER

Chapter 3. The Decorator Pattern: Decorating Objects



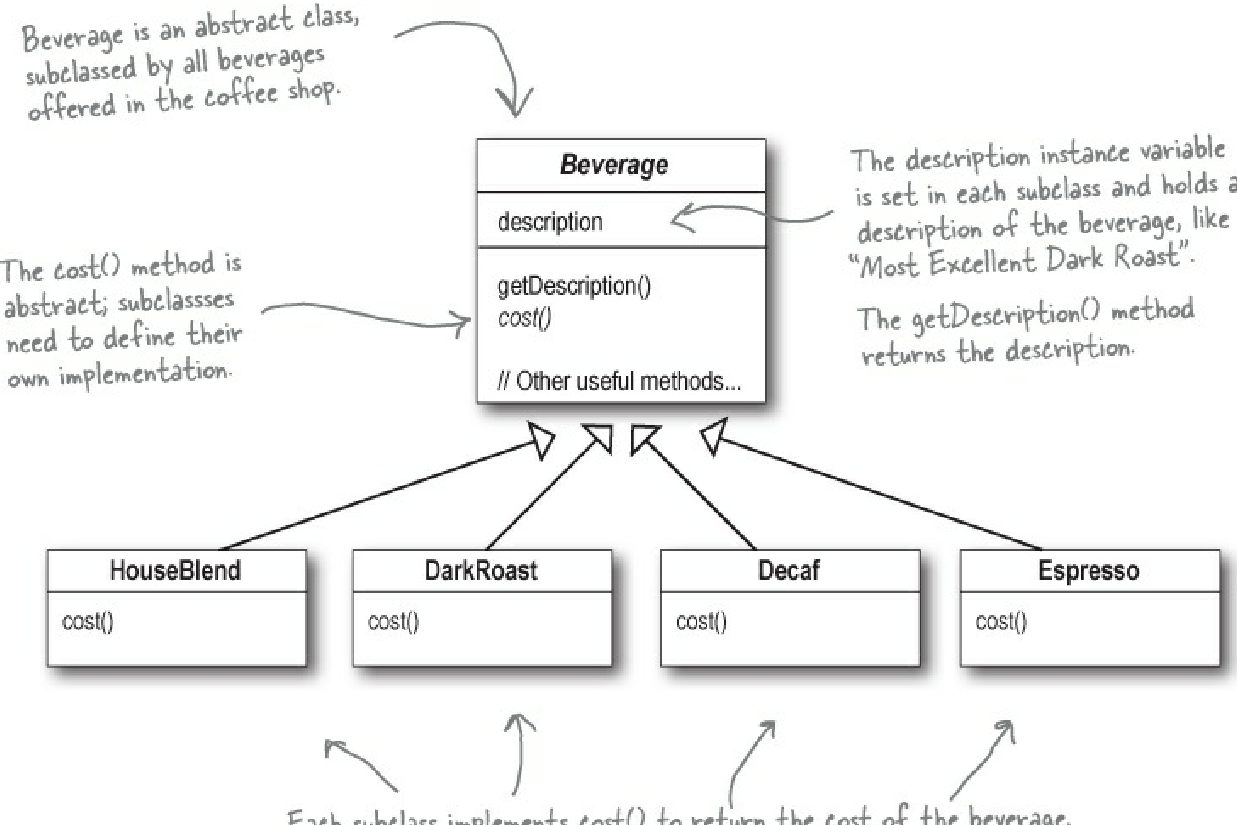
Just call this chapter "Design Eye for the Inheritance Guy." We'll re-examine the typical overuse of inheritance and you'll learn how to decorate your classes at runtime using a form of object composition. Why? Once you know the techniques of decorating, you'll be able to give your (or someone else's)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

Welcome to Starbuzz Coffee

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

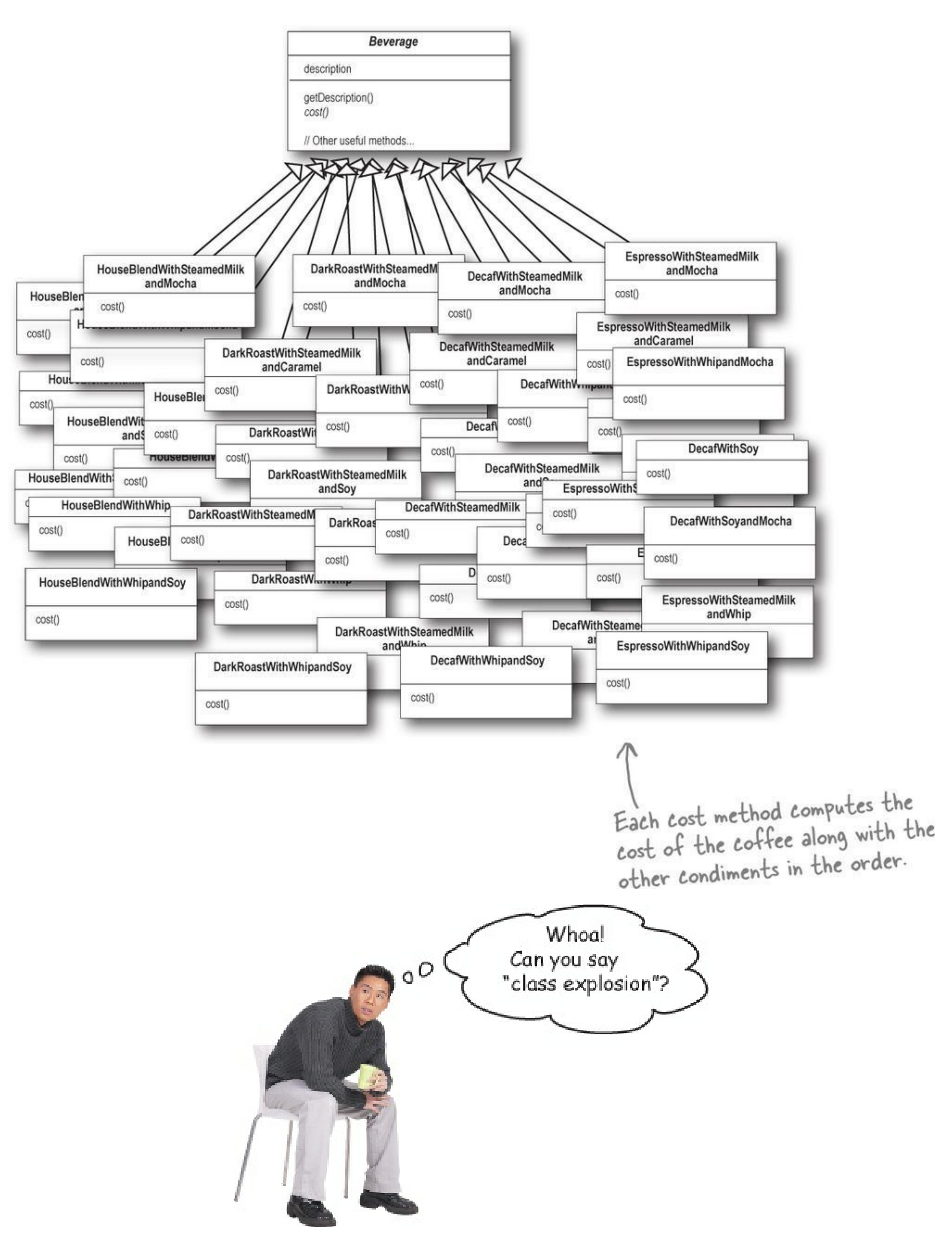
Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.

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



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

Here's their first attempt...



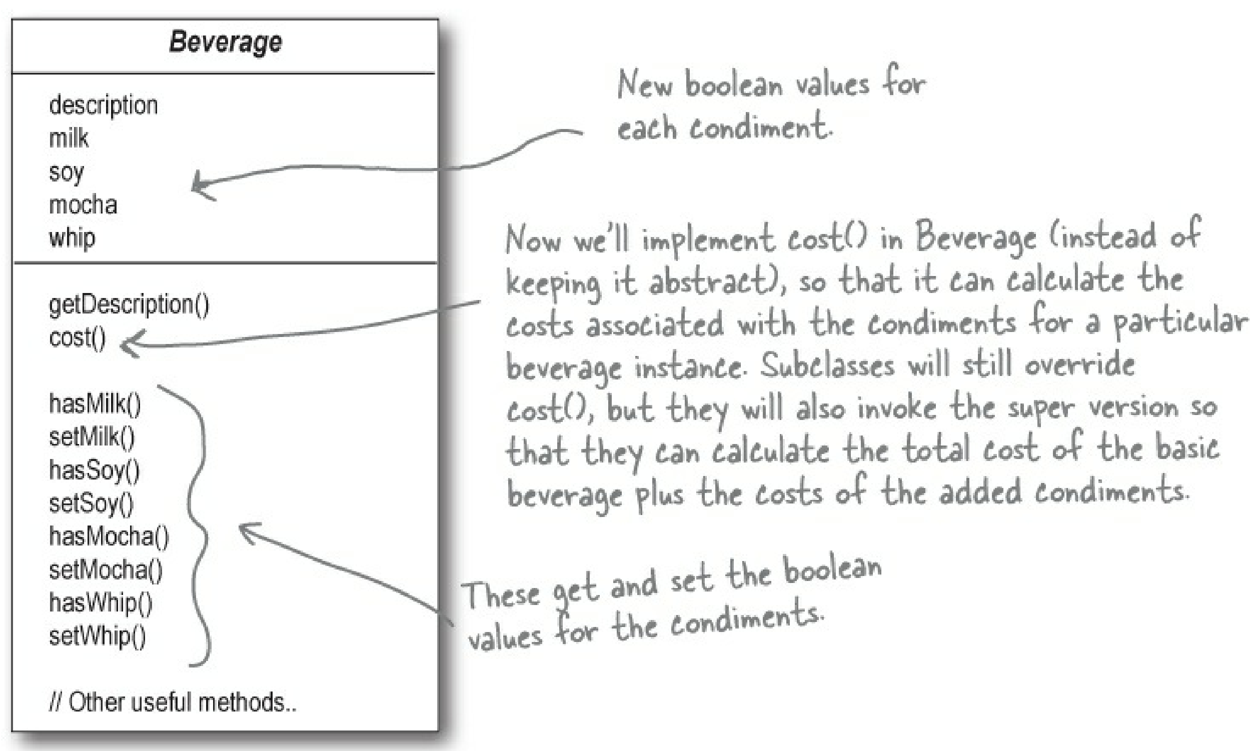
BRAIN POWER

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

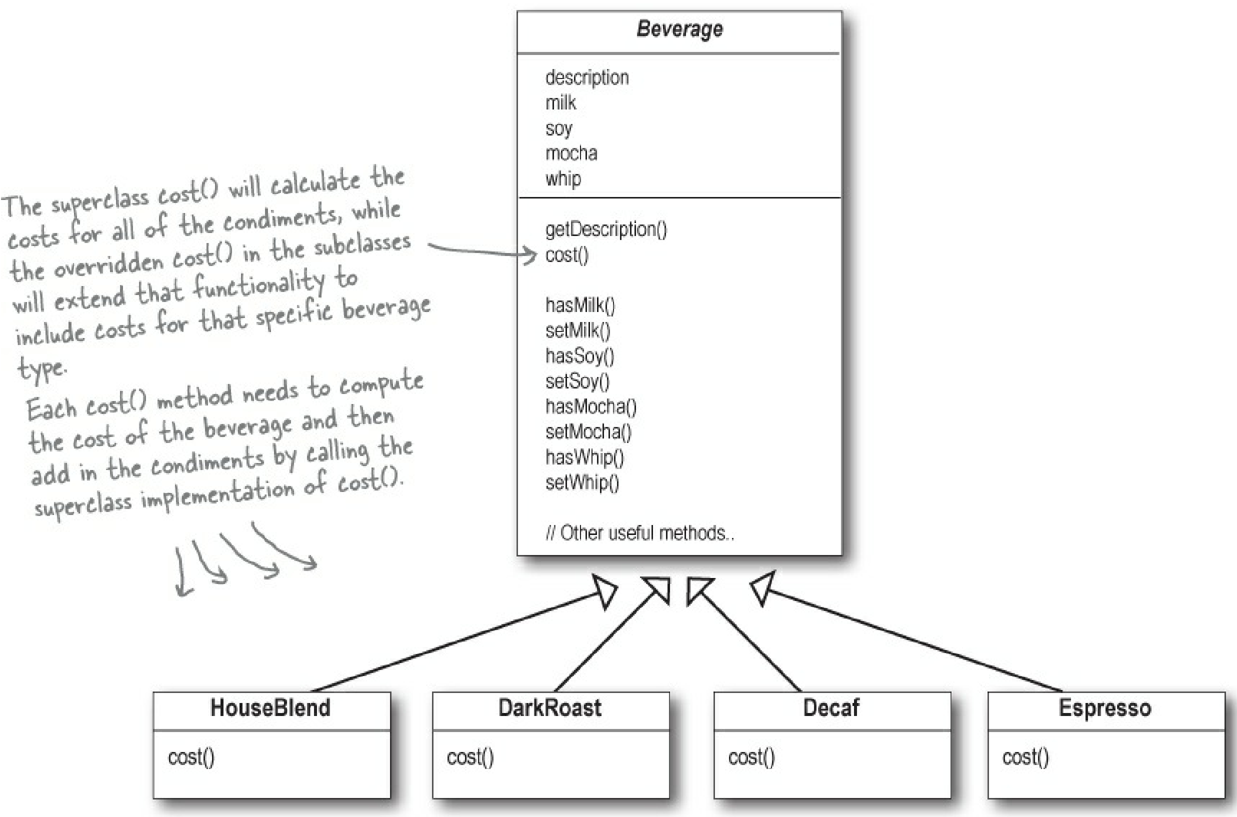
Thinking beyond the maintenance problem, which of the design principles that we’ve covered so far are they violating?

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

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



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



SHARPEN YOUR PENCIL

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

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




  }
}
```

```
public class DarkRoast extends Beverage {

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

  public double cost() {



  }
}
```

See, five classes total. This is definitely the way to go.

I'm not so sure; I can see some potential problems with this approach by thinking about how the design might need to change in the future.

SHARPEN YOUR PENCIL

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

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

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

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

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

What if a customer wants a double mocha?

Your turn:

MASTER AND STUDENT...

Master: *Grasshopper, it has been some time since our last meeting. Have you been deep in meditation on inheritance?*

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

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

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

Master: *Please, go on...*

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

Master: *Very good, Grasshopper, you are beginning to see the power of composition.*

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

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

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

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

The Open-Closed Principle

Grasshopper is on to one of the most important design principles:

DESIGN PRINCIPLE

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

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

Sorry, we're *closed*. That's right, we spent a lot of time getting this code

correct and bug free, so we can't let you alter the existing code. It must remain closed to modification. If you don't like it, you can speak to the manager.

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

THERE ARE NO DUMB QUESTIONS

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

A: **A:** That's a very good question. It certainly sounds contradictory at first. After all, the less modifiable something is, the harder it is to extend, right?
As it turns out, though, there are some clever OO techniques for allowing systems to be extended, even if we can't change the underlying code. Think about the Observer Pattern (in Chapter 2)... by adding new Observers, we can extend the Subject at any time, without adding code to the Subject. You'll see quite a few more ways of extending behavior with other OO design techniques.

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

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

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

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

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

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

While it may seem like a contradiction, there are techniques for allowing code to be extended without direct modification.
Be careful when choosing the areas of code that need to be extended; applying the Open-Closed Principle EVERYWHERE is wasteful and unnecessary, and can lead to complex, hard-to-understand code.

Meet the Decorator Pattern

Okay, we've seen that representing our beverage plus condiment pricing scheme with inheritance has not worked out very well — we get class explosions and rigid designs, or we add functionality to the base class that

isn’t appropriate for some of the subclasses.

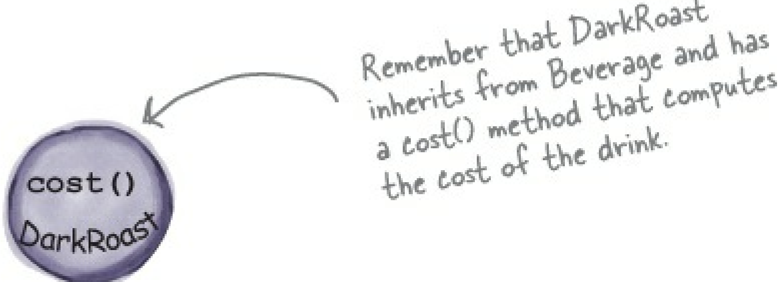
So, here’s what we’ll do instead: we’ll start with a beverage and “decorate” it with the condiments at runtime. For example, if the customer wants a Dark Roast with Mocha and Whip, then we’ll:

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

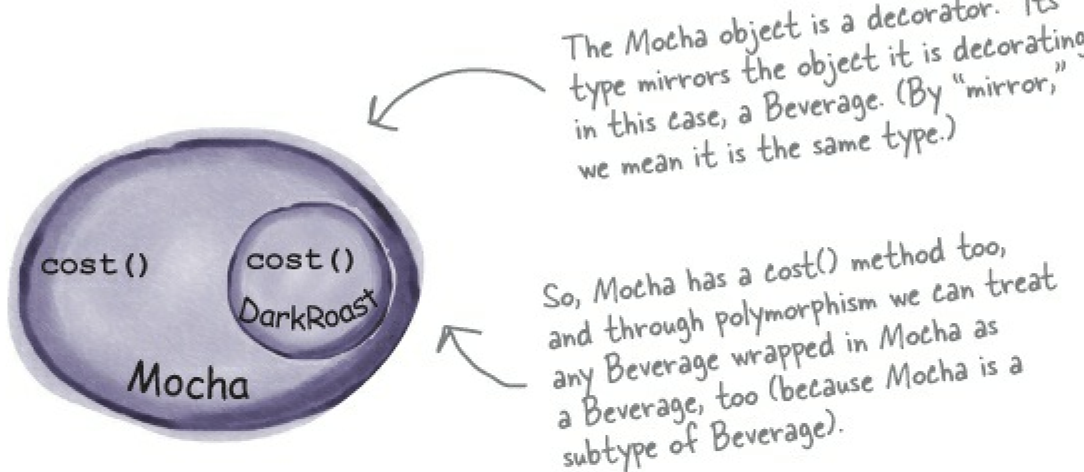
Okay, but how do you “decorate” an object, and how does delegation come into this? A hint: think of decorator objects as “wrappers.” Let’s see how this works...

Constructing a drink order with Decorators

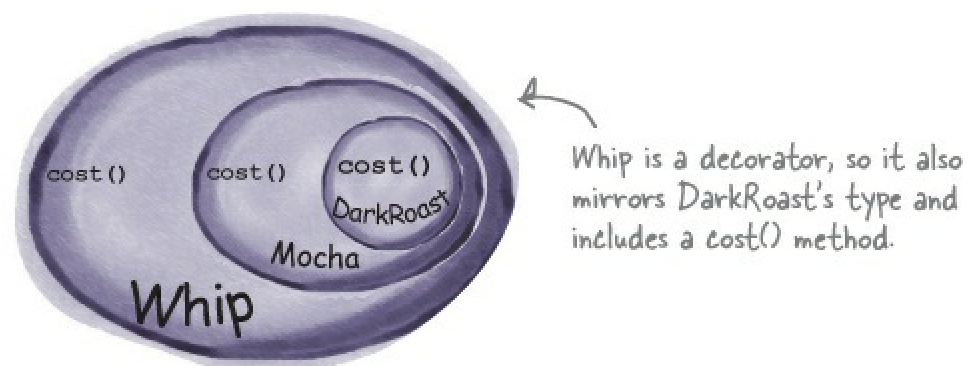
① **We start with our DarkRoast object.**



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



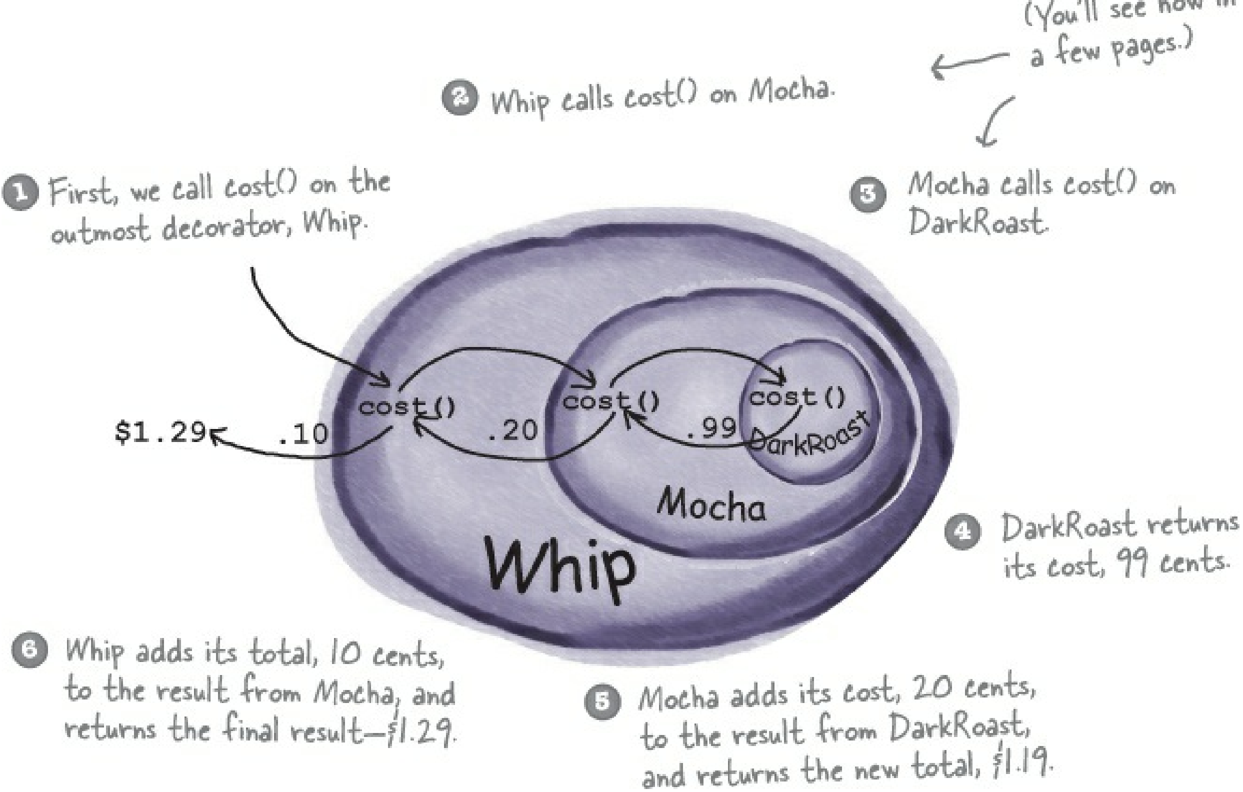
③ **The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.**



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

④ **Now it's time to compute the cost for the customer. We do this by**

calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.



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

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.

> **NOTE**
>
> Key point!

- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

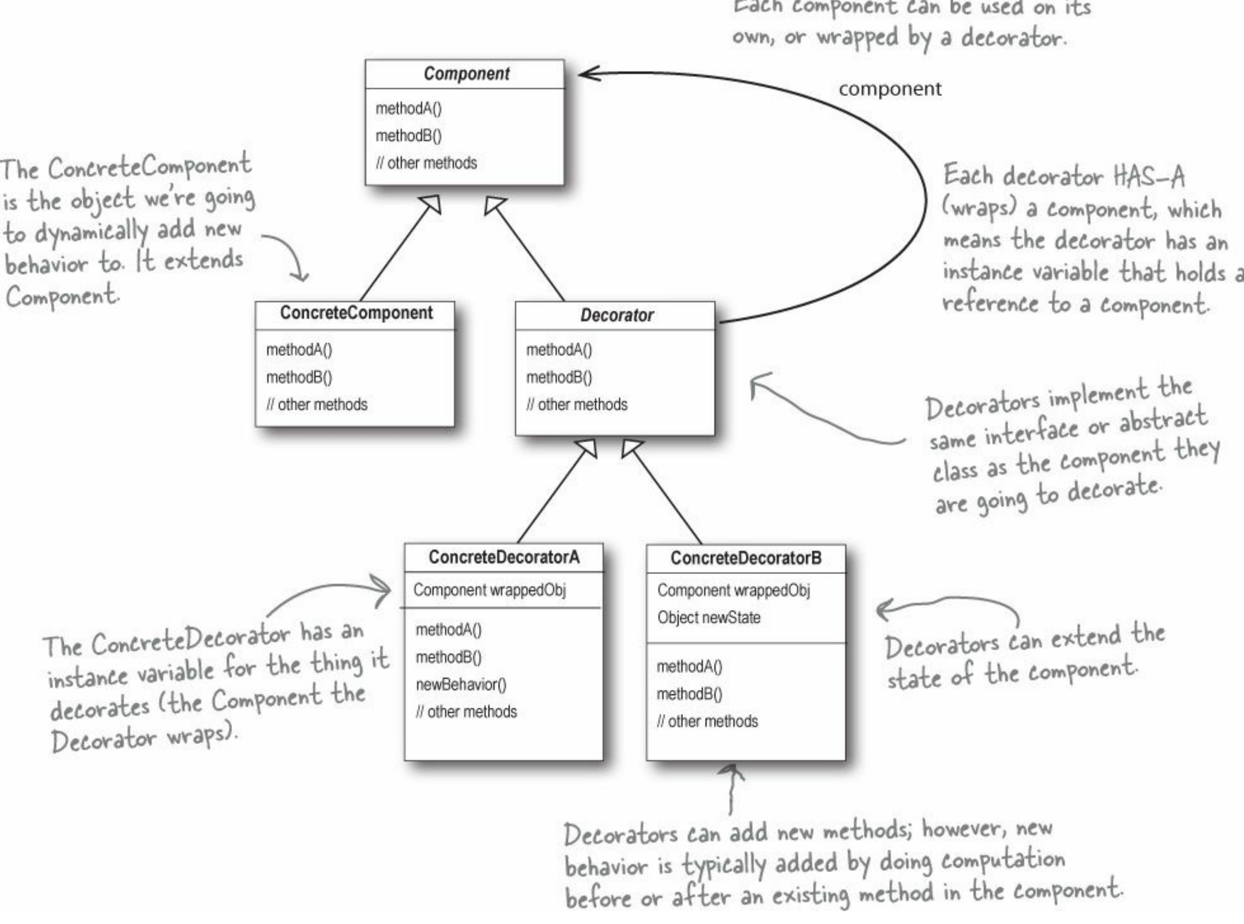
Now let's see how this all really works by looking at the Decorator Pattern definition and writing some code.

The Decorator Pattern defined

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

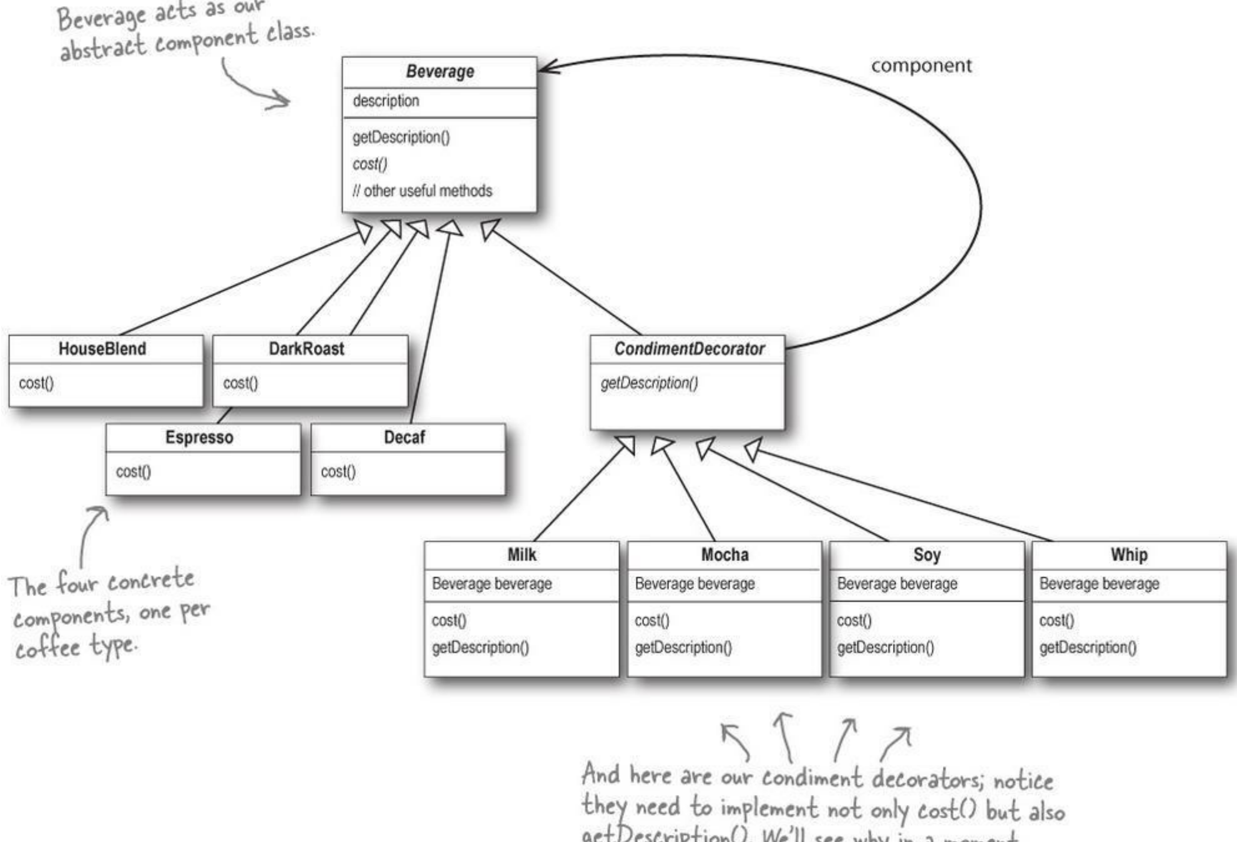
> **NOTE**
>
> **The Decorator Pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

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



Decorating our Beverages

Okay, let's work our Starbuzz beverages into this framework...



BRAIN POWER

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

Cubicle Conversation

Some confusion over Inheritance versus Composition

Sue: What do you mean?

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

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

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

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

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

other decorators.

Sue: That’s right.

Mary: Ooooh, I see. And because we are using object composition, we get a whole lot more flexibility about how to mix and match condiments and beverages. Very smooth.

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

Mary: And as I understand it, we can implement new decorators at any time to add new behavior. If we relied on inheritance, we’d have to go in and change existing code any time we wanted new behavior.

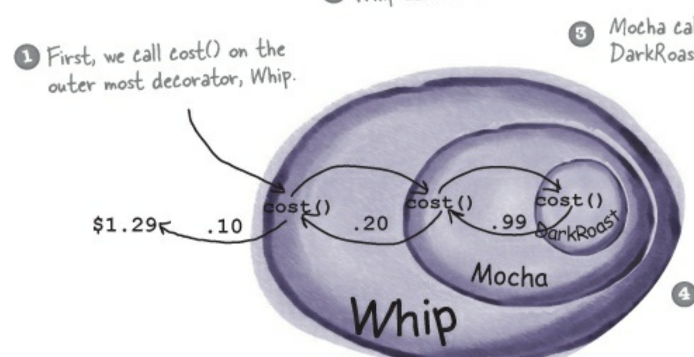
Sue: Exactly.

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

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

New barista training

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



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

6 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

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

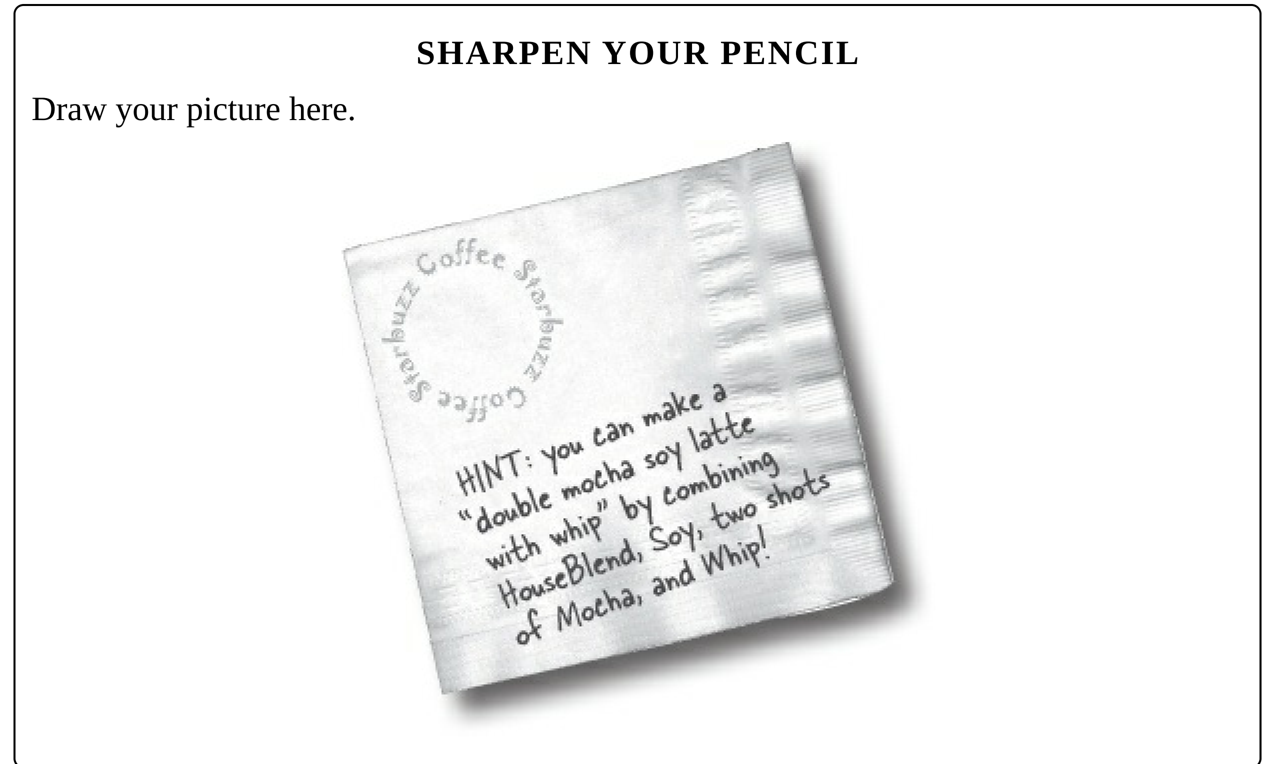
Starbuzz Coffee

Coffees	
House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99

Condiments	
Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10

SHARPEN YOUR PENCIL

Draw your picture here.



Writing the Starbuzz code

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

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

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

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

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

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

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

Coding beverages

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

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

```
public class Espresso extends Beverage {

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

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

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

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

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

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

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

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

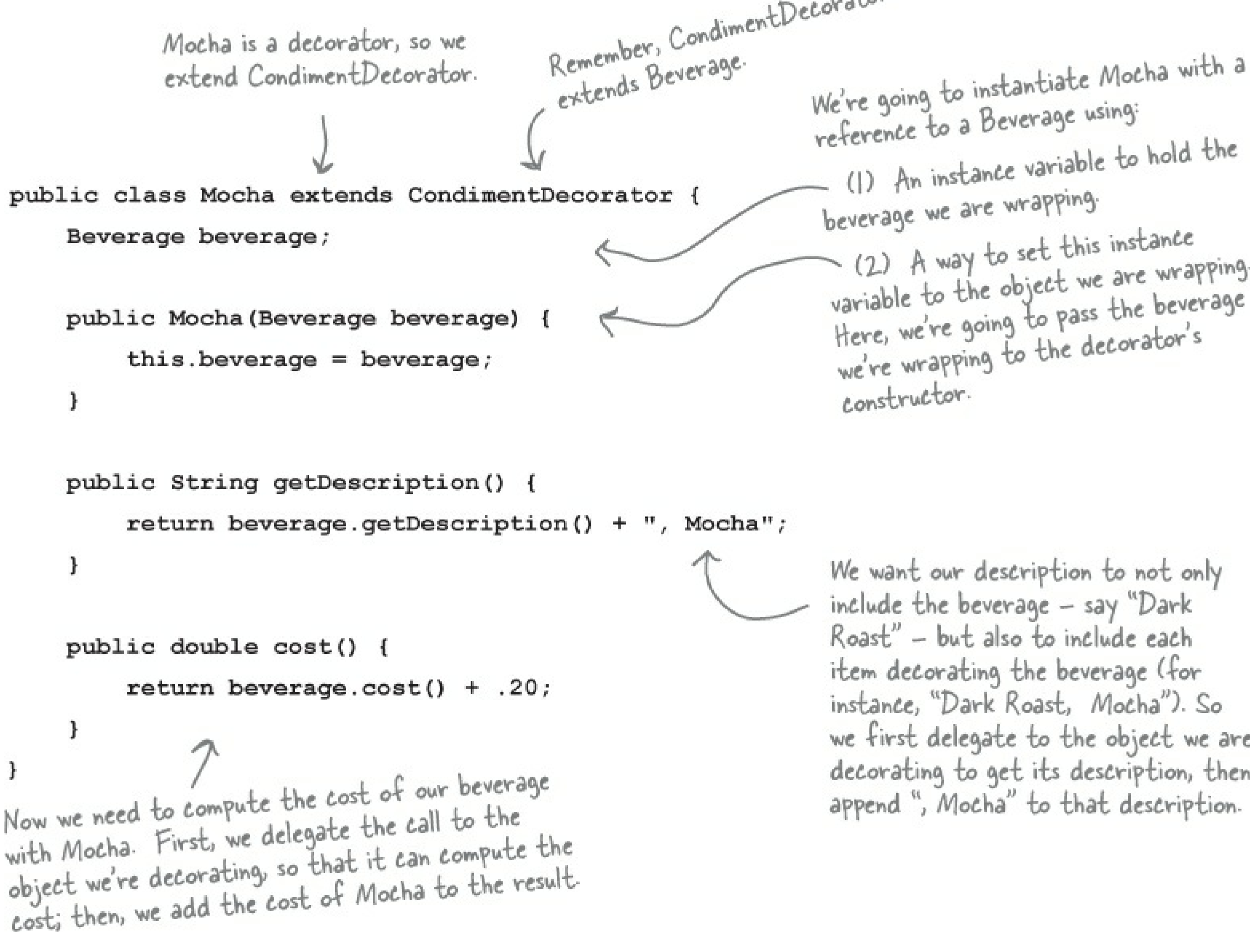
Starbuzz Coffee

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

Coding condiments

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

decorators. Here’s Mocha:



```java
public class Mocha extends CondimentDecorator {
    Beverage beverage;

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

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

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


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

> **EXERCISE**
>
> Write and compile the code for the other Soy and Whip condiments. You’ll need them to finish and test the application.

Serving some coffees

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

Here’s some test code*to make orders:

```
public class StarbuzzCoffee {

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

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

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

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

Make a DarkRoast object.

Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

* We're going to see a much better way of creating decorated objects when we cover the Factory and Builder Design Patterns. Please note that the Builder Pattern is covered in the Appendix.

Now, let's get those orders in:

```
File Edit Window Help CloudsInMyCoffee
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```

THERE ARE NO DUMB QUESTIONS

Q: **Q: I'm a little worried about code that might test for a specific concrete component — say, HouseBlend — and do something, like issue a discount. Once I've wrapped the HouseBlend with decorators, this isn't going to work anymore.**

A: **A:** That is exactly right. If you have code that relies on the concrete component's type, decorators will break that

code. As long as you only write code against the abstract component type, the use of decorators will remain transparent to your code. However, once you start writing code against concrete components, you'll want to rethink your application design and your use of decorators.

Q: **Q: Wouldn't it be easy for some client of a beverage to end up with a decorator that isn't the outermost decorator? Like if I had a DarkRoast with Mocha, Soy, and Whip, it would be easy to write code that somehow ended up with a reference to Soy instead of Whip, which means it would not include Whip in the order.**

A: **A:** You could certainly argue that you have to manage more objects with the Decorator Pattern and so there is an increased chance that coding errors will introduce the kinds of problems you suggest. However, decorators are typically created by using other patterns like Factory and Builder. Once we've covered these patterns, you'll see that the creation of the concrete component with its decorator is "well encapsulated" and doesn't lead to these kinds of problems.

Q: **Q: Can decorators know about the other decorations in the chain? Say I wanted my getDescription() method to print "Whip, Double Mocha" instead of "Mocha, Whip, Mocha." That would require that my outermost decorator know all the decorators it is wrapping.**

A: **A:** Decorators are meant to add behavior to the object they wrap. When you need to peek at multiple layers into the decorator chain, you are starting to push the decorator beyond its true intent. Nevertheless, such things are possible. Imagine a CondimentPrettyPrint decorator that parses the final decription and can print "Mocha, Whip, Mocha" as "Whip, Double Mocha." Note that getDescription() could return an ArrayList of descriptions to make this easier.

SHARPEN YOUR PENCIL

Our friends at Starbuzz have introduced sizes to their menu. You can now order a coffee in tall, grande, and venti sizes (translation: small, medium, and large). Starbuzz saw this as an intrinsic part of the coffee class, so they've added two methods to the Beverage class: setSize() and getSize(). They'd also like for the condiments to be charged according to size, so for instance, Soy costs 10¢, 15¢, and 20¢, respectively, for tall, grande, and venti coffees. The updated Beverage class is shown below.

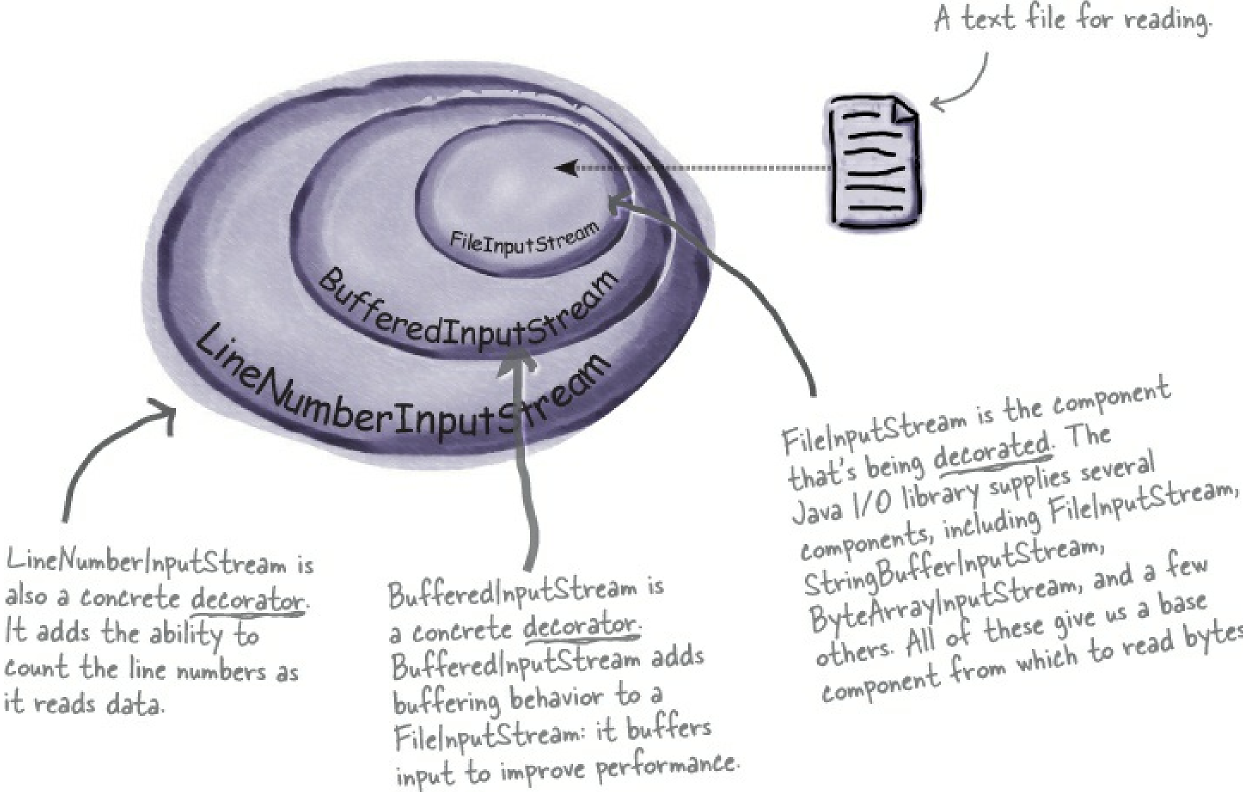
How would you alter the decorator classes to handle this change in requirements?

```
public abstract class Beverage {
      public enum Size { TALL, GRANDE, VENTI };
      Size size = Size.TALL;
      String description = "Unknown Beverage";
      public String getDescription() {
             return description;
      }
      public void setSize(Size size) {
             this.size = size;
      }
      public Size getSize() {
             return this.size;
      }
      public abstract double cost();
}
```

Real World Decorators: Java I/O

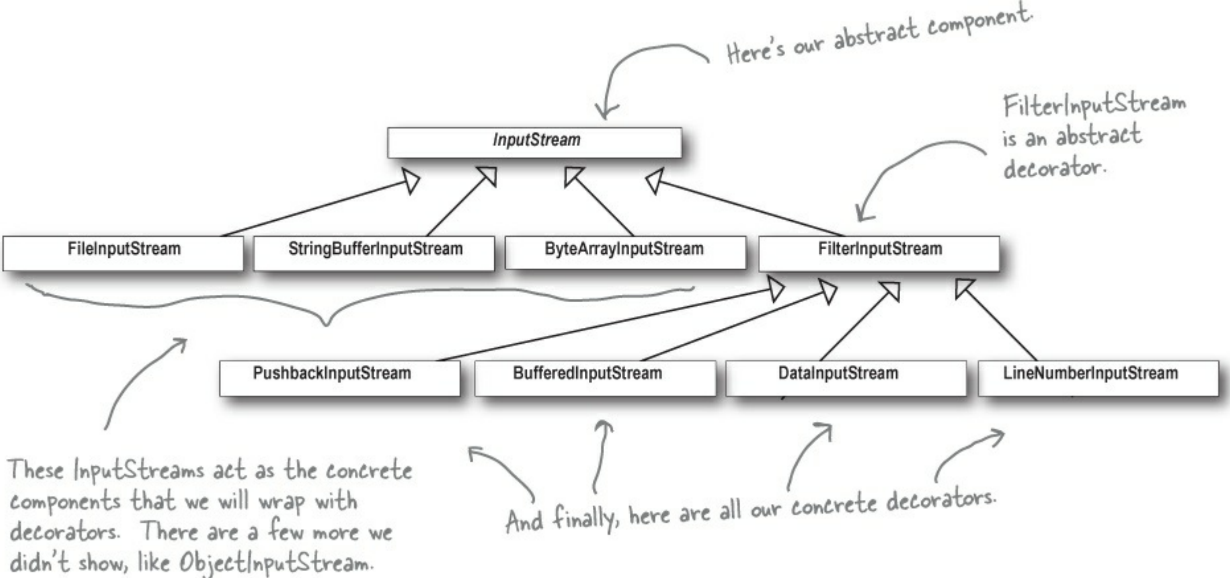
The large number of classes in the java.io package is... *overwhelming*. Don't

feel alone if you said “whoa” the first (and second and third) time you looked at this API. But now that you know the Decorator Pattern, the I/O classes should make more sense since the java.io package is largely based on Decorator. Here’s a typical set of objects that use decorators to add functionality to reading data from a file:



BufferedInputStream and ***LineNumber***InputStream both extend ***Filter***InputStream, which acts as the abstract decorator class.

Decorating the java.io classes



You can see that this isn't so different from the Starbuzz design. You should now be in a good position to look over the java.io API docs and compose decorators on the various *input* streams.

You'll see that the *output* streams have the same design. And you've probably already found that the Reader/Writer streams (for character-based data) closely mirror the design of the streams classes (with a few differences and inconsistencies, but close enough to figure out what's going on).

Java I/O also points out one of the *downsides* of the Decorator Pattern: designs using this pattern often result in a large number of small classes that can be overwhelming to a developer trying to use the Decorator-based API. But now that you know how Decorator works, you can keep things in perspective and when you're using someone else's Decorator-heavy API, you can work through how their classes are organized so that you can easily use wrapping to get the behavior you're after.

Writing your own Java I/O Decorator

Okay, you know the Decorator Pattern, you've seen the I/O class diagram. You should be ready to write your own input decorator.

How about this: write a decorator that converts all uppercase characters to lowercase in the input stream. In other words, if we read in "I know the Decorator Pattern therefore I RULE!" then your decorator converts this to "i know the decorator pattern therefore i rule!"

Don't forget to import java.io... (not shown).

First, extend the FilterInputStream, the abstract decorator for all InputStreams.

```
public class LowerCaseInputStream extends FilterInputStream {

    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = in.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = in.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

Now we need to implement two read methods. They take a byte (or an array of bytes) and convert each byte (that represents a character) to lowercase if it's an uppercase character.

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

Test out your new Java I/O Decorator

Write some quick code to test the I/O decorator:

```
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;

        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }

            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Set up the FileInputStream and decorate it, first with a BufferedInputStream and then our brand new LowerCaseInputStream filter.

```
I know the Decorator Pattern therefore I RULE!
```

test.txt file

You need to make this file.

Just use the stream to read characters until the end of file and print as we go.

Give it a spin

```
File Edit Window Help DecoratorsRule
% java InputTest
i know the decorator pattern therefore i rule!
%
```

PATTERNS EXPOSED

This week's interview: Confessions of a Decorator

Head First: Welcome, Decorator Pattern. We've heard that you've been a bit down on yourself lately?

Decorator: Yes, I know the world sees me as the glamorous design pattern, but you know, I've got my share of problems just like everyone.

HeadFirst: Can you perhaps share some of your troubles with us?

Decorator: Sure. Well, you know I've got the power to add flexibility to designs, that much is for sure, but I also have a *dark side*. You see, I can sometimes add a lot of small

classes to a design and this occasionally results in a design that's less than straightforward for others to understand.

HeadFirst: Can you give us an example?

Decorator: Take the Java I/O libraries. These are notoriously difficult for people to understand at first. But if they just saw the classes as a set of wrappers around an InputStream, life would be much easier.

HeadFirst: That doesn't sound so bad. You're still a great pattern, and improving this is just a matter of public education, right?

Decorator: There's more, I'm afraid. I've got typing problems: you see, people sometimes take a piece of client code that relies on specific types and introduce decorators without thinking through everything. Now, one great thing about me is that ***you can usually insert decorators transparently and the client never has to know it's dealing with a decorator***. But like I said, some code is dependent on specific types and when you start introducing decorators, boom! Bad things happen.

HeadFirst: Well, I think everyone understands that you have to be careful when inserting decorators. I don't think this is a reason to be too down on yourself.

Decorator: I know, I try not to be. I also have the problem that introducing decorators can increase the complexity of the code needed to instantiate the component. Once you've got decorators, you've got to not only instantiate the component, but also wrap it with who knows how many decorators.

HeadFirst: I'll be interviewing the Factory and Builder patterns next week — I hear they can be very helpful with this?

Decorator: That's true; I should talk to those guys more often.

HeadFirst: Well, we all think you're a great pattern for creating flexible designs and staying true to the Open-Closed Principle, so keep your chin up and think positively!

Decorator: I'll do my best, thank you.

Tools for your Design Toolbox

You've got another chapter under your belt and a new principle and pattern in the toolbox.

OO Basics

...ction

...sulation

...rphism

...ance

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

We now have the Open-Closed Principle to guide us. We're going to strive to design our system so that the closed parts are isolated from our new extensions.

OO Patterns

Stra... encap... inter... vary i...

Observer - d... w... d... a...

Decorator - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

And here's our first pattern for creating designs that satisfy the Open-Closed Principle. Or was it really the first? Is there another pattern we've used that follows this principle as well?

BULLET POINTS

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
- In our designs we should allow behavior to be extended without the need to modify existing code.
- Composition and delegation can often be used to add new behaviors at runtime.
- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
- Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)
- Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
- You can wrap a component with any number of decorators.
- Decorators are typically transparent to the client of the component; that is, unless the client is relying on the component's concrete type.
- Decorators can result in many small objects in our design, and overuse can be complex.

SHARPEN YOUR PENCIL SOLUTION

Write the cost() methods for the following classes (pseudo-Java is okay). Here's our solution:

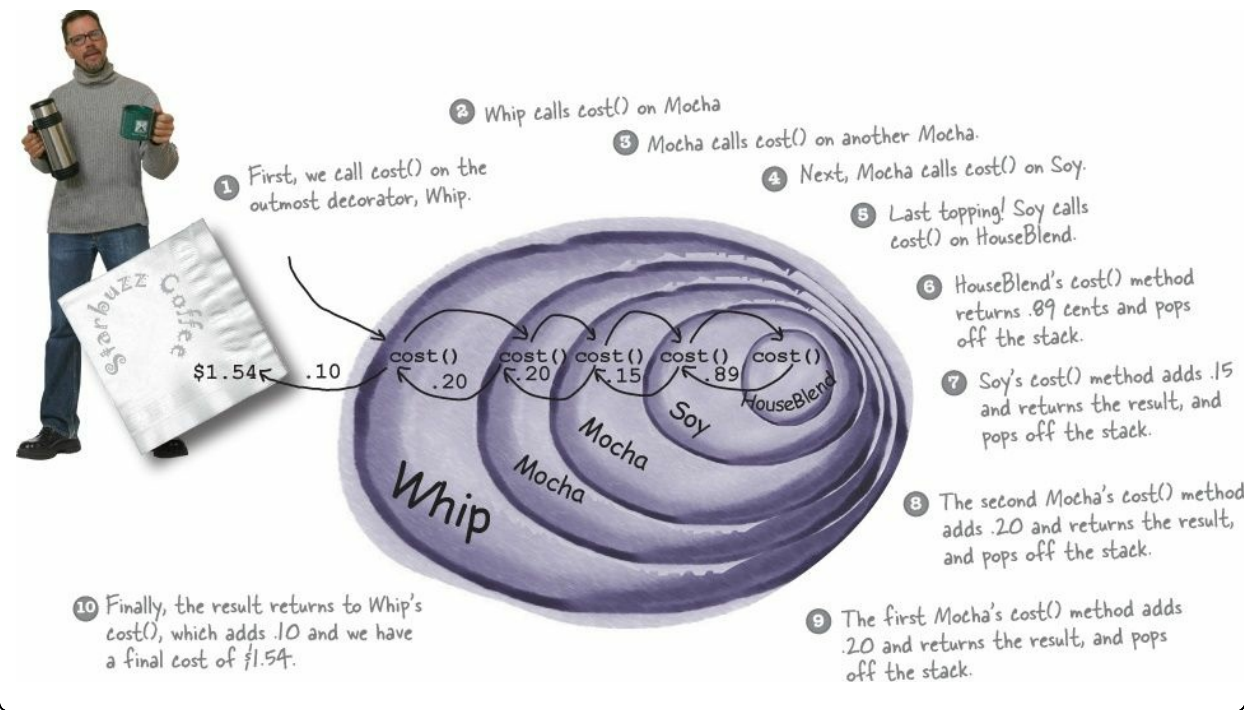
```
public class Beverage {

// declare instance variables for milkCost,
// soyCost, mochaCost, and whipCost, and
// getters and setters for milk, soy, mocha
// and whip.

public double cost() {

        float condimentCost = 0.0;
        if (hasMilk()) {
            condimentCost += milkCost;
        }
        if (hasSoy()) {
            condimentCost += soyCost;
        }
        if (hasMocha()) {
            condimentCost += mochaCost;
        }
        if (hasWhip()) {
            condimentCost += whipCost;
        }
        return condimentCost;
    }
```

```
}

public class DarkRoast extends Beverage {

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

    public double cost() {
        return 1.99 + super.cost();
    }
}
```

SHARPEN YOUR PENCIL SOLUTION

New barista training

“double mocha soy latte with whip”



SHARPEN YOUR PENCIL SOLUTION

Our friends at Starbuzz have introduced sizes to their menu. You can now order a coffee in tall, grande, and venti sizes (for us normal folk: small, medium, and large). Starbuzz saw this as an intrinsic part of the coffee class, so they’ve added two methods to the Beverage class: setSize() and getSize(). They’d also like for the condiments to be charged according to size, so for instance, Soy costs 10¢, 15¢, and 20¢, respectively, for tall, grande, and venti coffees.

How would you alter the decorator classes to handle this change in requirements? Here’s

our solution.

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

    public Size getSize() {
        return beverage.getSize();
    }
}
```

We moved the Beverage instance variable into CondimentDecorator, and added a method, getSize() for the decorators that simply returns the size of the beverage.

```
public class Soy extends CondimentDecorator {
    public Soy(Beverage beverage) {
        this.beverage = beverage;
    }

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

    public double cost() {
        double cost = beverage.cost();
        if (beverage.getSize() == Size.TALL) {
            cost += .10;
        } else if (beverage.getSize() == Size.GRANDE) {
            cost += .15;
        } else if (beverage.getSize() == Size.VENTI) {
            cost += .20;
        }
        return cost;
    }
}
```

Here we get the size (which propagates all the way to the concrete beverage) and then add the appropriate cost.

Chapter 4. The Factory Pattern: Baking with OO Goodness

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

When you see "new," think "concrete."

Yes, when you use **new** you are certainly instantiating a concrete class, so that's definitely an implementation, not an interface. And it's a good question; you've learned that tying your code to a concrete class can make it more fragile and less flexible.

```
Duck duck = new MallardDuck();
```

We want to use interfaces to keep code flexible.

But we have to create an instance of a concrete class!

When you have a whole set of related concrete classes, often you're forced to write code like this:

```
Duck duck;

if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDuck();
}
```

We have a bunch of different duck classes, and we don't know until runtime which one we need to instantiate.

Here we've got several concrete classes being instantiated, and the decision of which to instantiate is made at runtime depending on some set of conditions.

When you see code like this, you know that when it comes time for changes or extensions, you'll have to reopen this code and examine what needs to be added (or deleted). Often this kind of code ends up in several parts of the application making maintenance and updates more difficult and error-prone.

What's wrong with "new"?

Technically there's nothing wrong with **new**. After all, it's a fundamental part of Java. The real culprit is our old friend CHANGE and how change impacts our use of **new**.

By coding to an interface, you know you can insulate yourself from a lot of changes that might happen to a system down the road. Why? If your code is written to an interface, then it will work with any new classes implementing that interface through polymorphism. However, when you have code that makes use of lots of concrete classes, you're looking for trouble because that code may have to be changed as new concrete classes are added. So, in other words, your code will not be "closed for modification." To extend it with new concrete types, you'll have to reopen it.

NOTE

Remember that designs should be "open for extension but closed for modification" - see Chapter 3 for a review.

So what can you do? It's times like these that you can fall back on OO Design Principles to look for clues. Remember, our first principle deals with change and guides us to *identify the aspects that vary and separate them from what stays the same*.

BRAIN POWER

How might you take all the parts of your application that instantiate concrete classes and separate or encapsulate them from the rest of your application?

Identifying the aspects that vary

Let's say you have a pizza shop, and as a cutting-edge pizza store owner in Objectville you might end up writing some code like this:

```
Pizza orderPizza() {
        Pizza pizza = new Pizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

For flexibility, we really want this to be an abstract class or interface, but we can't directly instantiate either of those.

But you need more than one type of pizza...

So then you'd add some code that *determines* the appropriate type of pizza and then goes about *making* the pizza:

```
Pizza orderPizza(String type) {
      Pizza pizza;

      if (type.equals("cheese")) {
          pizza = new CheesePizza();
      } else if (type.equals("greek")) {
          pizza = new GreekPizza();
      } else if (type.equals("pepperoni")) {
          pizza = new PepperoniPizza();
      }

      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
}
```

We're now passing in the type of pizza to orderPizza.

Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable. Note that each pizza here has to implement the Pizza interface.

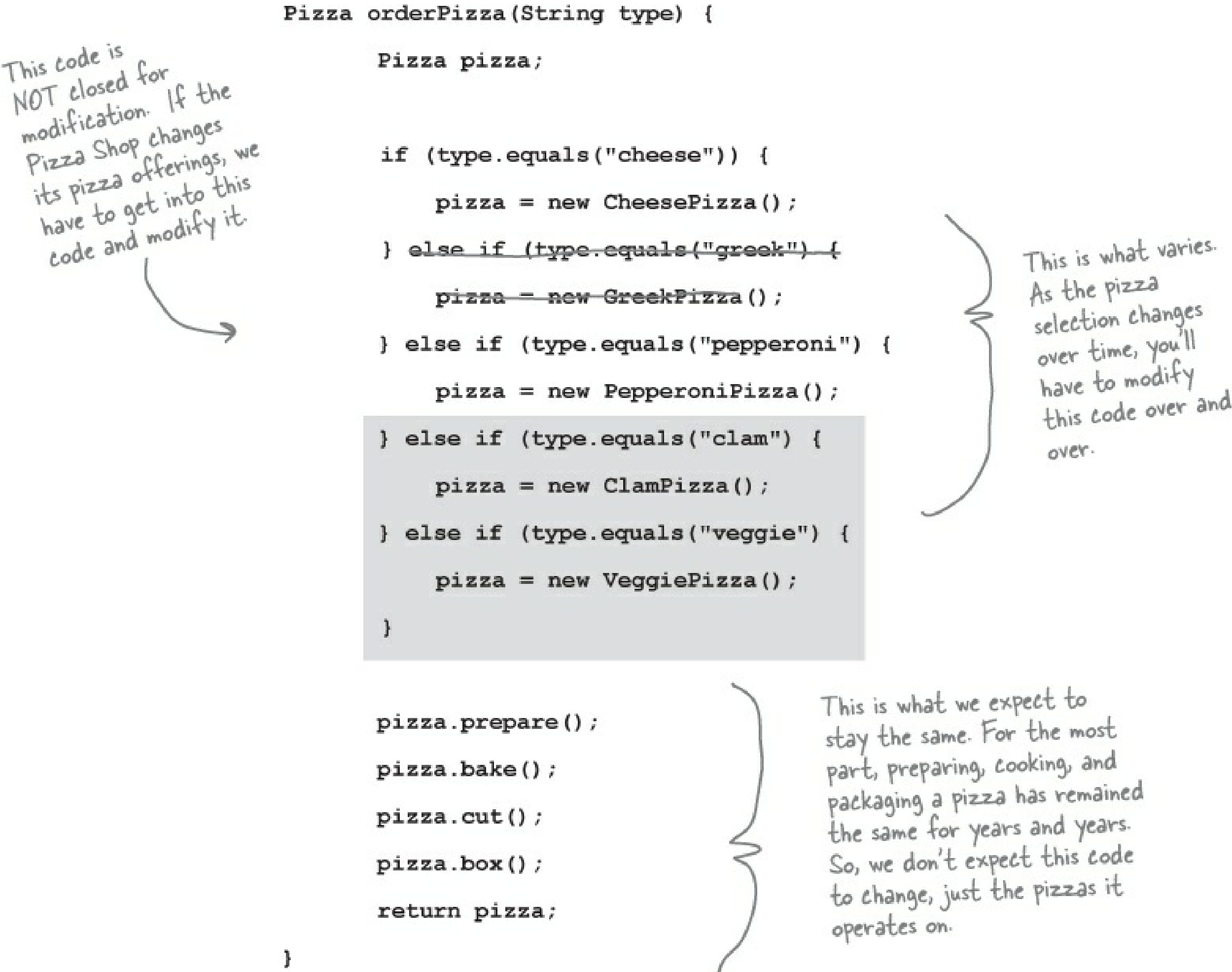
Once we have a Pizza, we prepare it (you know, roll the dough, put on the sauce and add the toppings & cheese), then we bake it, cut it and box it!

Each Pizza subtype (CheesePizza, VeggiePizza, etc.) knows how to prepare itself.

But the pressure is on to add more pizza types

You realize that all of your competitors have added a couple of trendy pizzas

to their menus: the Clam Pizza and the Veggie Pizza. Obviously you need to keep up with the competition, so you'll add these items to your menu. And you haven't been selling many Greek Pizzas lately, so you decide to take that off the menu:

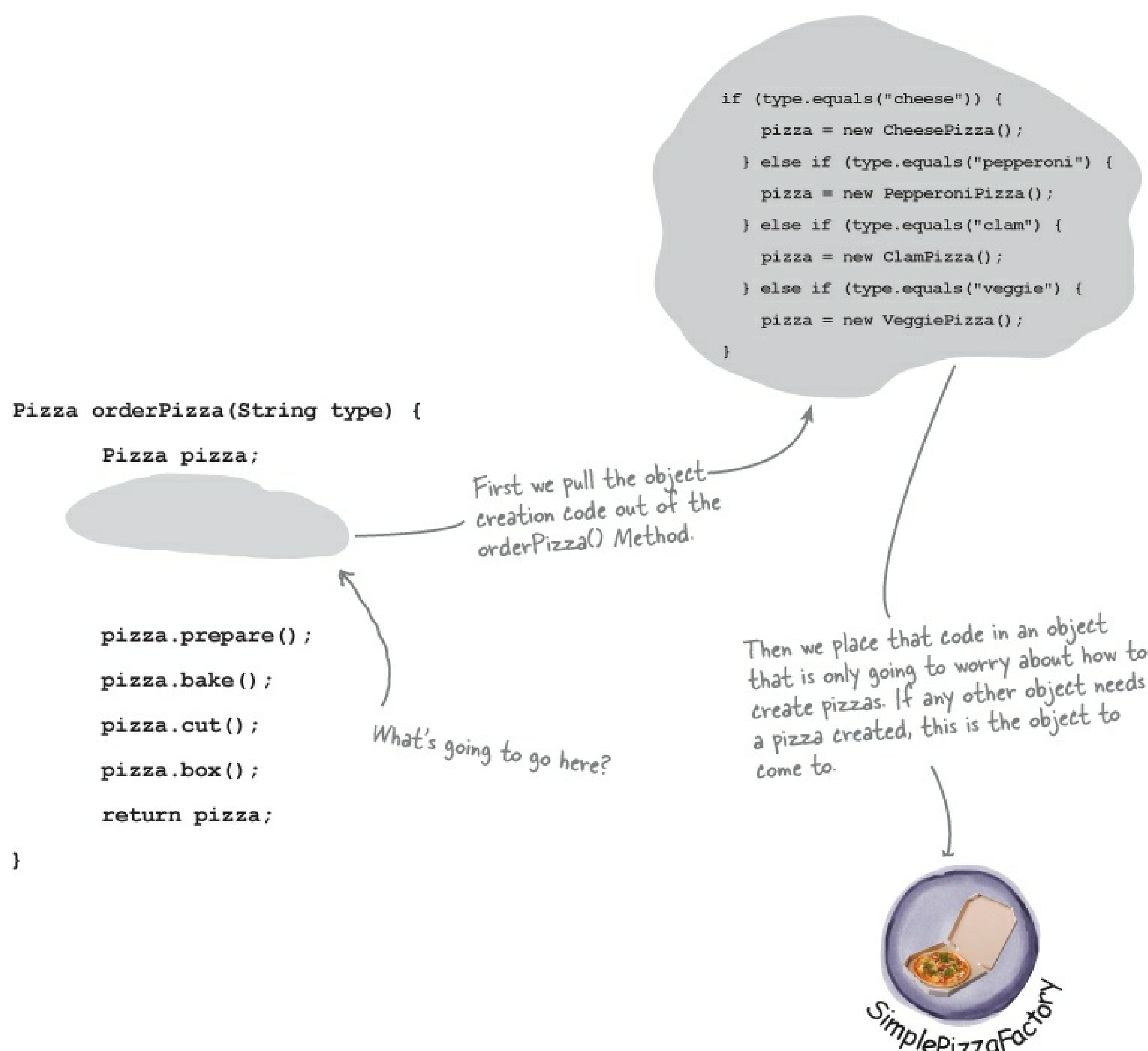



```
Pizza orderPizza(String type) {
    Pizza pizza;

    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek") {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni") {
        pizza = new PepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new ClamPizza();
    } else if (type.equals("veggie") {
        pizza = new VeggiePizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

Clearly, dealing with *which* concrete class is instantiated is really messing up our orderPizza() method and preventing it from being closed for modification. But now that we know what is varying and what isn't, it's probably time to encapsulate it.

Encapsulating object creation

So now we know we'd be better off moving the object creation out of the orderPizza() method. But how? Well, what we're going to do is take the creation code and move it out into another object that is only going to be concerned with creating pizzas.



We've got a name for this new object: we call it a Factory.

Factories handle the details of object creation. Once we have a SimplePizzaFactory, our orderPizza() method just becomes a client of that object. Any time it needs a pizza it asks the pizza factory to make one. Gone are the days when the orderPizza() method needs to know about Greek versus Clam pizzas. Now the orderPizza() method just cares that it gets a pizza that implements the Pizza interface so that it can call prepare(), bake(), cut(), and box().

We've still got a few details to fill in here; for instance, what does the orderPizza() method replace its creation code with? Let's implement a simple factory for the pizza store and find out...

Building a simple pizza factory

We’ll start with the factory itself. What we’re going to do is define a class that encapsulates the object creation for all pizzas. Here it is...

Here’s our new class, the SimplePizzaFactory. It has one job in life: creating pizzas for its clients.

First we define a createPizza() method in the factory. This is the method all clients will use to instantiate new objects.

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

Here’s the code we plucked out of the orderPizza() method.

This code is still parameterized by the type of the pizza, just like our original orderPizza() method was.

THERE ARE NO DUMB QUESTIONS

Q: Q: What’s the advantage of this? It looks like we are just pushing the problem off to another object.

A: A: One thing to remember is that the SimplePizzaFactory may have many clients. We’ve only seen the orderPizza() method; however, there may be a PizzaShopMenu class that uses the factory to get pizzas for their current description and price. We might also have a HomeDelivery class that handles pizzas in a different way than our PizzaShop class but is also a client of the factory.

So, by encapsulating the pizza creating in one class, we now have only one place to make modifications when the implementation changes.

Don’t forget, we are also just about to remove the concrete instantiations from our client code.

Q: Q: I’ve seen a similar design where a factory like this is defined as a static method. What is the difference?

A: A: Defining a simple factory as a static method is a common technique and is often called a static factory. Why use a static method? Because you don’t need to instantiate an object to make use of the create method. But remember it also has the disadvantage that you can’t subclass and change the behavior of the create method.

Reworking the PizzaStore class

Now it’s time to fix up our client code. What we want to do is rely on the factory to create the pizzas for us. Here are the changes:

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    // other methods here
}
```

Now we give PizzaStore a reference to a SimplePizzaFactory.

PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

Notice that we’ve replaced the **new operator** with a create **method** on the factory object. No more concrete instantiations here!

BRAIN POWER

Q: We know that object composition allows us to change behavior dynamically at runtime (among other things) because we can swap in and out implementations. How might we be able to use that in the PizzaStore? What factory implementations might we be able to swap in and out?

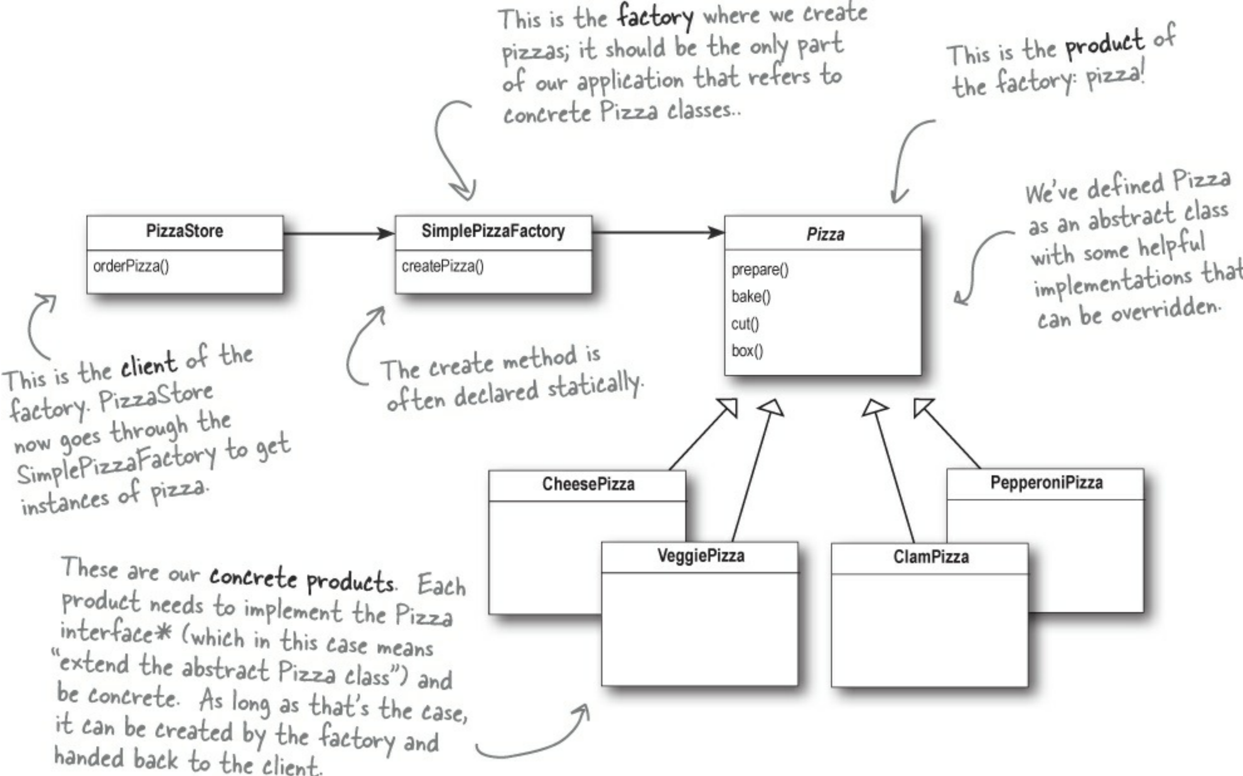
A: We don’t know about you, but we’re thinking New York, Chicago, and California style pizza factories (let’s not forget New Haven, too)

The Simple Factory defined

Pattern Honorable Mention

The Simple Factory isn’t actually a Design Pattern; it’s more of a programming idiom. But it is commonly used, so we’ll give it a Head First Pattern Honorable Mention. Some developers do mistake this idiom for the “Factory Pattern,” so the next time there is an awkward silence between you and another developer, you’ve got a nice topic to break the ice.

Just because Simple Factory isn’t a REAL pattern doesn’t mean we shouldn’t check out how it’s put together. Let’s take a look at the class diagram of our new Pizza Store:



Think of Simple Factory as a warm up. Next, we'll explore two heavy-duty patterns that are both factories. But don't worry, there's more pizza to come!

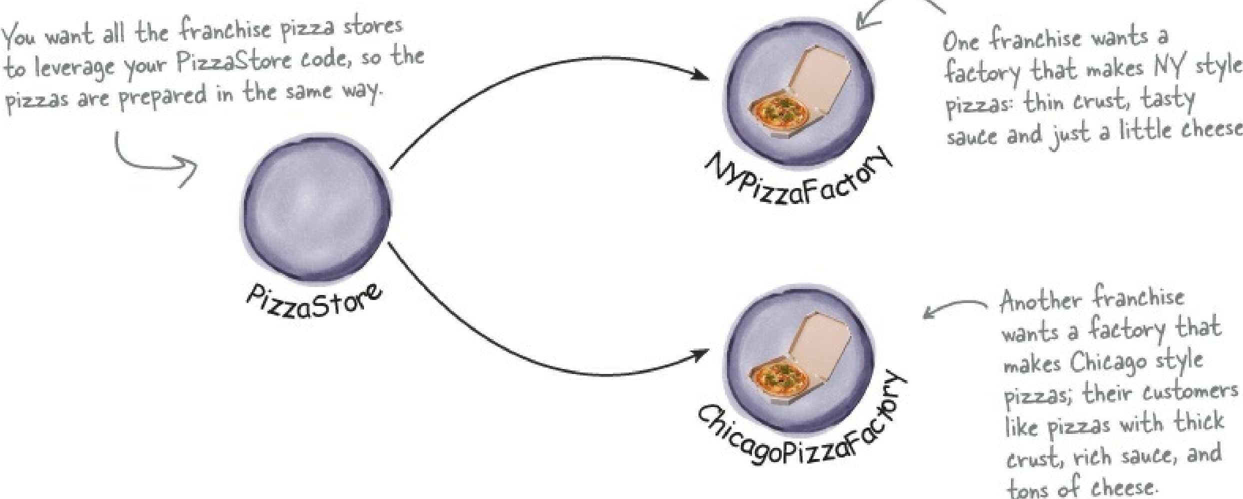
NOTE

*Just another reminder: in design patterns, the phrase "implement an interface" does NOT always mean "write a class that implements a Java interface, by using the 'implements' keyword in the class declaration." In the general use of the phrase, a concrete class implementing a method from a supertype (which could be a class OR interface) is still considered to be "implementing the interface" of that supertype.

Franchising the pizza store

Your Objectville PizzaStore has done so well that you've trounced the competition and now everyone wants a PizzaStore in their own neighborhood. As the franchiser, you want to ensure the quality of the franchise operations and so you want them to use your time-tested code.

But what about regional differences? Each franchise might want to offer different styles of pizzas (New York, Chicago, and California, to name a few), depending on where the franchise store is located and the tastes of the local pizza connoisseurs.

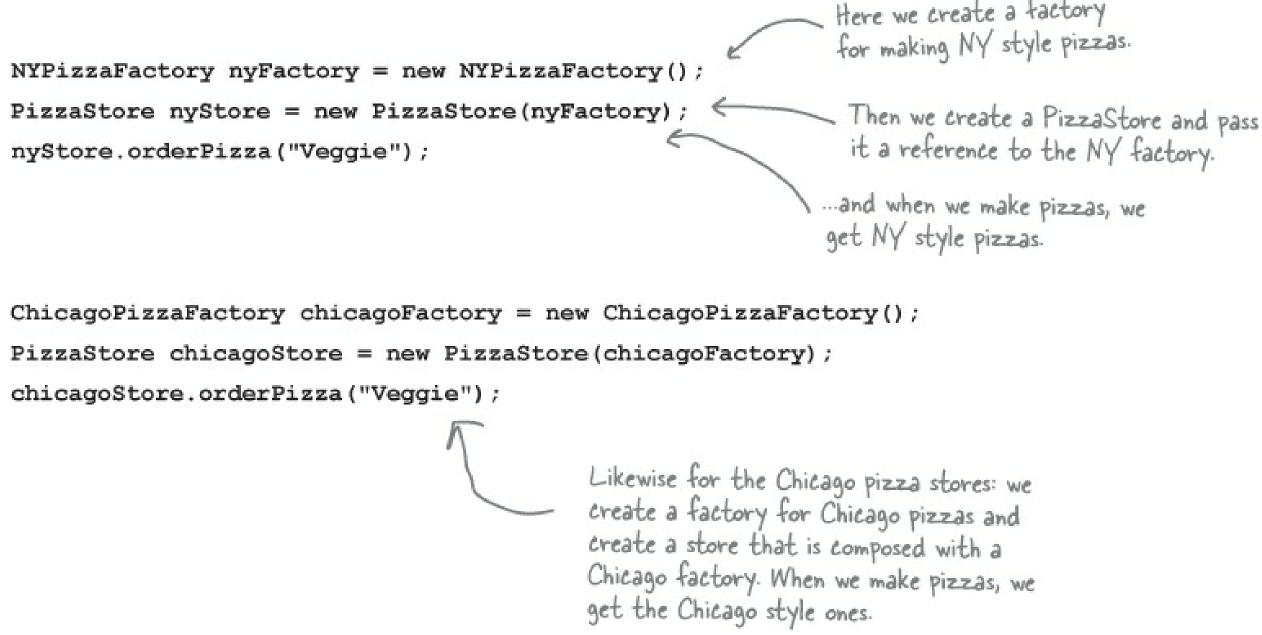


We've seen one approach...

If we take out SimplePizzaFactory and create three different factories — NYPizzaFactory, ChicagoPizzaFactory and CaliforniaPizzaFactory — then we can just compose the PizzaStore with the appropriate factory and a

franchise is good to go. That’s one approach.

Let’s see what that would look like...



But you’d like a little more quality control...

So you test-marketed the SimpleFactory idea, and what you found was that the franchises were using your factory to create pizzas, but starting to employ their own home-grown procedures for the rest of the process: they’d bake things a little differently, they’d forget to cut the pizza and they’d use third-party boxes.

Rethinking the problem a bit, you see that what you’d really like to do is create a framework that ties the store and the pizza creation together, yet still allows things to remain flexible.

In our early code, before the SimplePizzaFactory, we had the pizza-making code tied to the PizzaStore, but it wasn’t flexible. So, how can we have our pizza and eat it too?

A framework for the pizza store

There *is* a way to localize all the pizza-making activities to the PizzaStore class, and yet give the franchises freedom to have their own regional style.

What we're going to do is put the createPizza() method back into PizzaStore, but this time as an **abstract method**, and then create a PizzaStore subclass for each regional style.

First, let's look at the changes to the PizzaStore:

PizzaStore is now abstract (see why below).

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    abstract Pizza createPizza(String type);
}
```

Now createPizza is back to being a call to a method in the PizzaStore rather than on a factory object.

All this looks just the same...

Now we've moved our factory object to this method.

Our "factory method" is now abstract in PizzaStore.

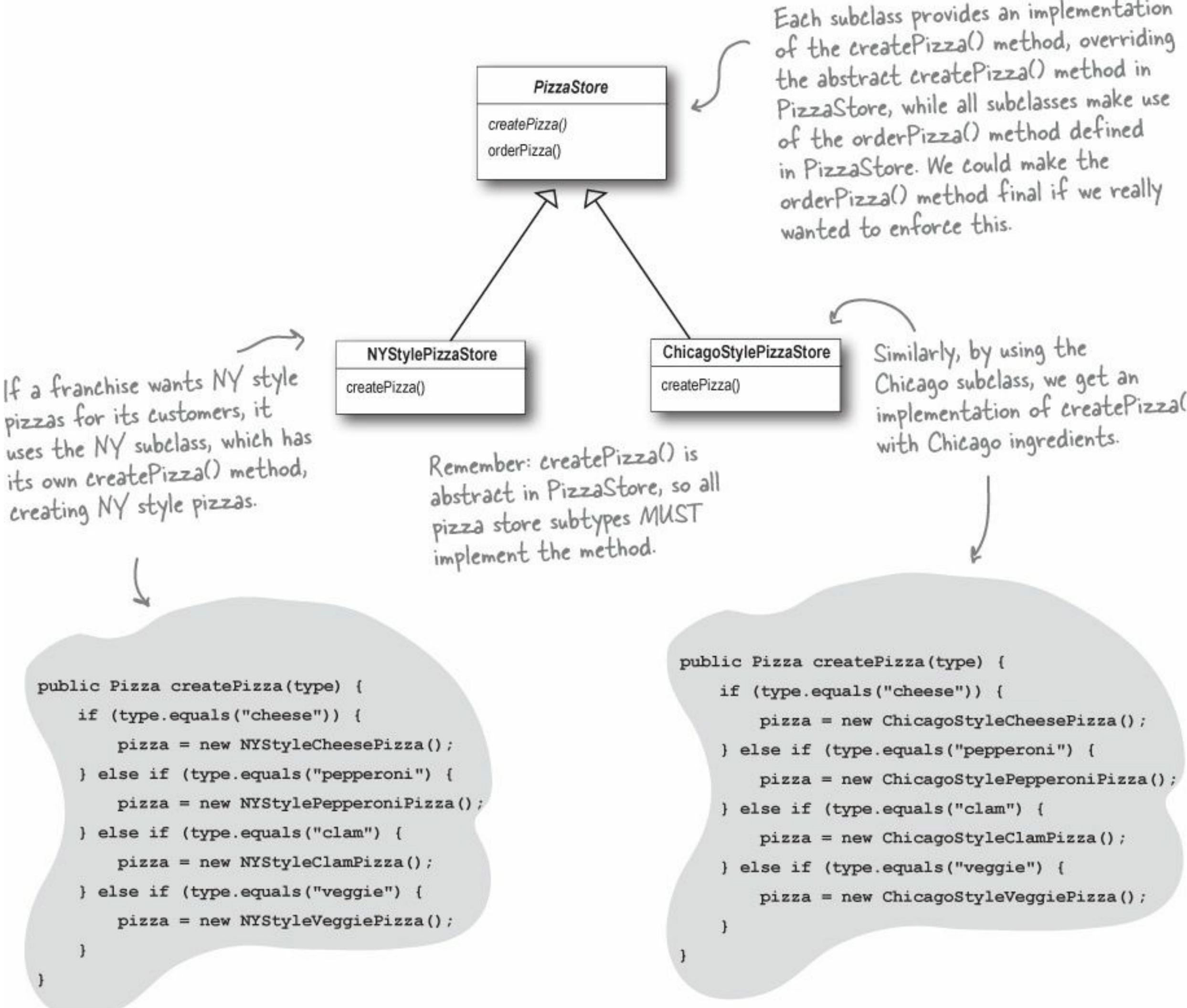
Now we've got a store waiting for subclasses; we're going to have a subclass for each regional type (NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore) and each subclass is going to make the decision about what makes up a pizza. Let's take a look at how this is going to work.

Allowing the subclasses to decide

Remember, the PizzaStore already has a well-honed order system in the orderPizza() method and you want to ensure that it's consistent across all franchises.

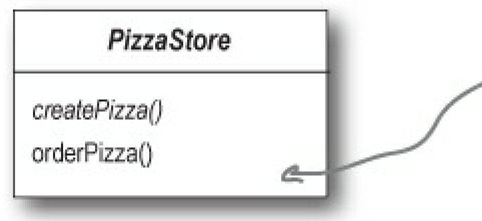
What varies among the regional PizzaStores is the style of pizzas they make — New York Pizza has thin crust, Chicago Pizza has thick, and so on — and we are going to push all these variations into the createPizza() method and

make it responsible for creating the right kind of pizza. The way we do this is by letting each subclass of PizzaStore define what the createPizza() method looks like. So, we will have a number of concrete subclasses of PizzaStore, each with its own pizza variations, all fitting within the PizzaStore framework and still making use of the well-tuned orderPizza() method.



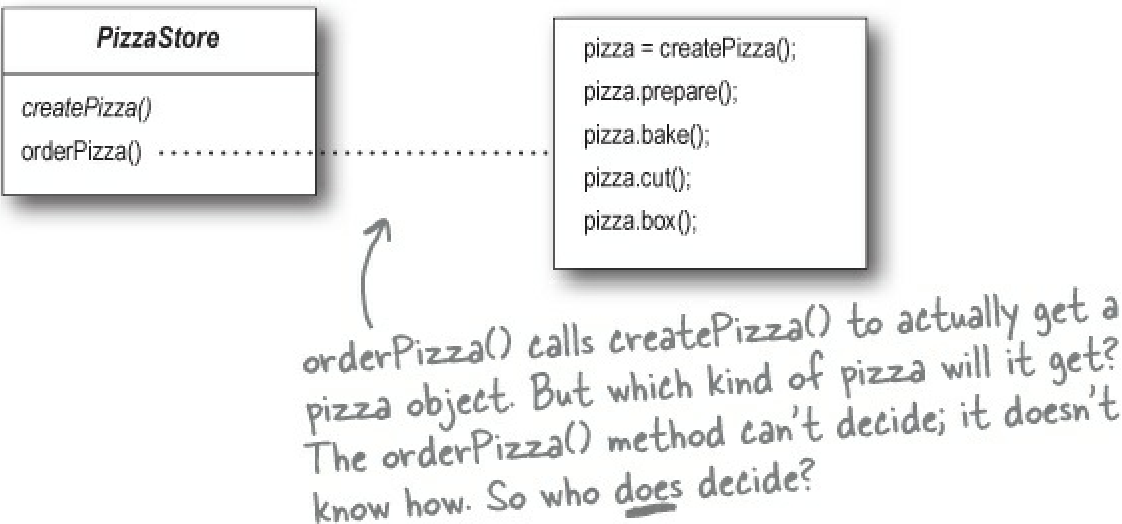
I don't get it. The PizzaStore subclasses are just subclasses. How are they deciding anything? I don't see any logical decision-making code in NYStylePizzaStore....

Well, think about it from the point of view of the PizzaStore's orderPizza() method: it is defined in the abstract PizzaStore, but concrete types are only created in the subclasses.

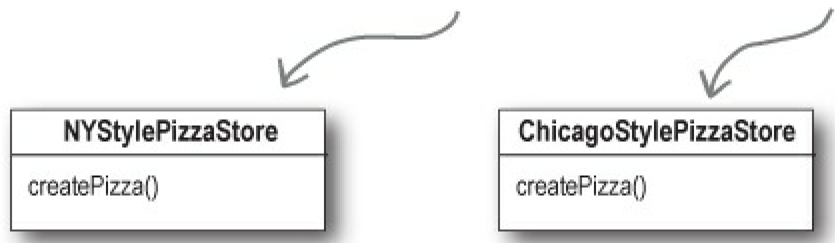


orderPizza() is defined in the abstract PizzaStore, not the subclasses. So, the method has no idea which subclass is actually running the code and making the pizzas.

Now, to take this a little further, the orderPizza() method does a lot of things with a Pizza object (like prepare, bake, cut, box), but because Pizza is abstract, orderPizza() has no idea what real concrete classes are involved. In other words, it's decoupled!



When orderPizza() calls createPizza(), one of your subclasses will be called into action to create a pizza. Which kind of pizza will be made? Well, that's decided by the choice of pizza store you order from, NYStylePizzaStore or ChicagoStylePizzaStore.



So, is there a real-time decision that subclasses make? No, but from the perspective of orderPizza(), if you chose a NYStylePizzaStore, that subclass gets to determine which pizza is made. So the subclasses aren't really "deciding" — it was *you* who decided by choosing which store you wanted — but they do determine which kind of pizza gets made.

Let's make a PizzaStore

Being a franchise has its benefits. You get all the PizzaStore functionality for free. All the regional stores need to do is subclass PizzaStore and supply a createPizza() method that implements their style of Pizza. We'll take care of the big three pizza styles for the franchisees.

Here's the New York regional style:

createPizza() returns a Pizza, and the subclass is fully responsible for which concrete Pizza it instantiates.

The NYPizzaStore extends PizzaStore, so it inherits the orderPizza() method (among others).

```
public class NYPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

We've got to implement createPizza(), since it is abstract in PizzaStore.

Here's where we create our concrete classes. For each type of Pizza we create the NY style.

NOTE

* Note that the orderPizza() method in the superclass has no clue which Pizza we are creating; it just knows it can prepare, bake, cut, and box it!

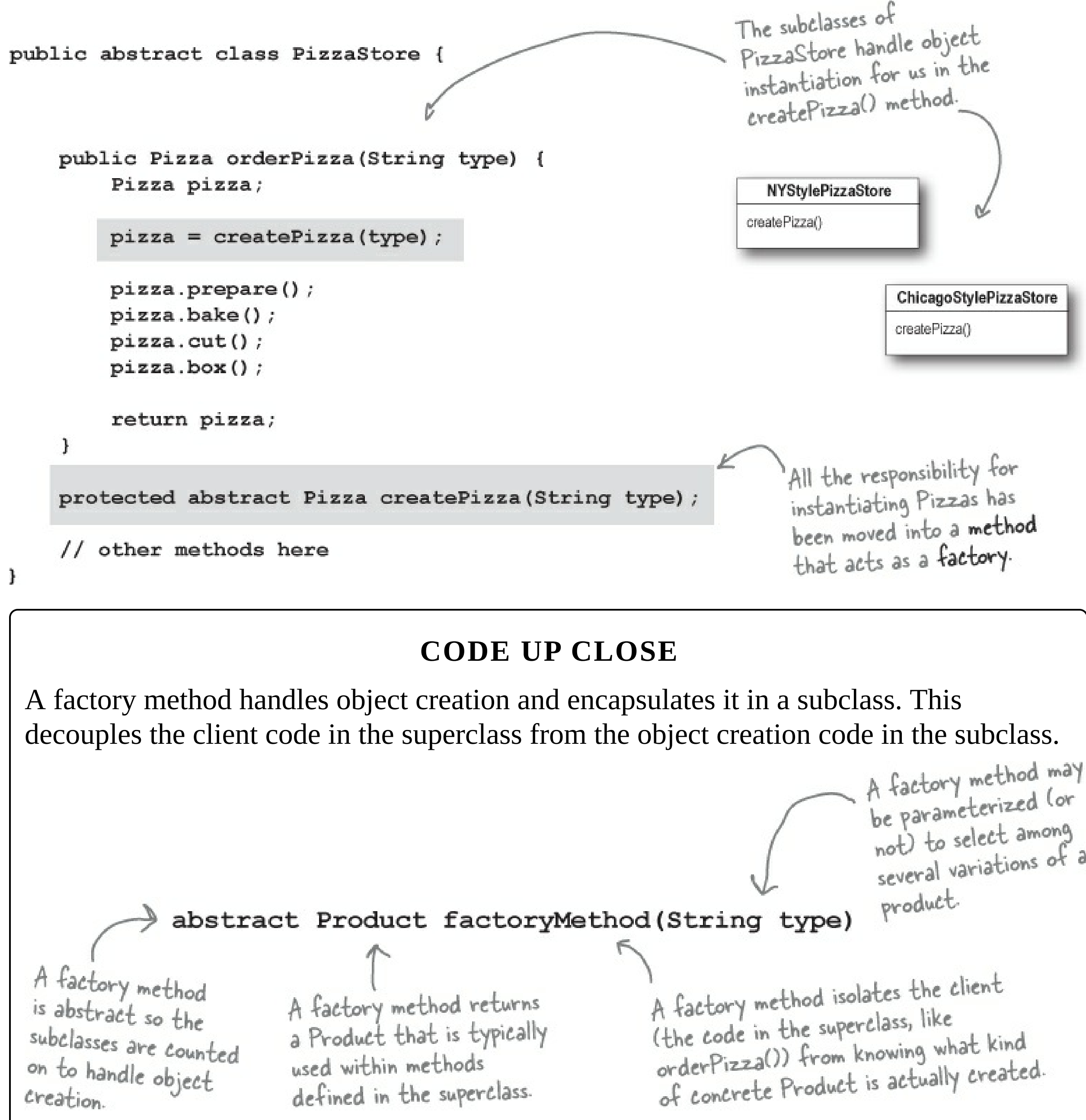
Once we've got our PizzaStore subclasses built, it will be time to see about ordering up a pizza or two. But before we do that, why don't you take a crack at building the Chicago Style and California Style pizza stores on the next page.

SHARPEN YOUR PENCIL

We've knocked out the NYPizzaStore; just two more to go and we'll be ready to franchise! Write the Chicago and California PizzaStore implementations here:

Declaring a factory method

With just a couple of transformations to the PizzaStore we've gone from having an object handle the instantiation of our concrete classes to a set of subclasses that are now taking on that responsibility. Let's take a closer look:



CODE UP CLOSE

A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

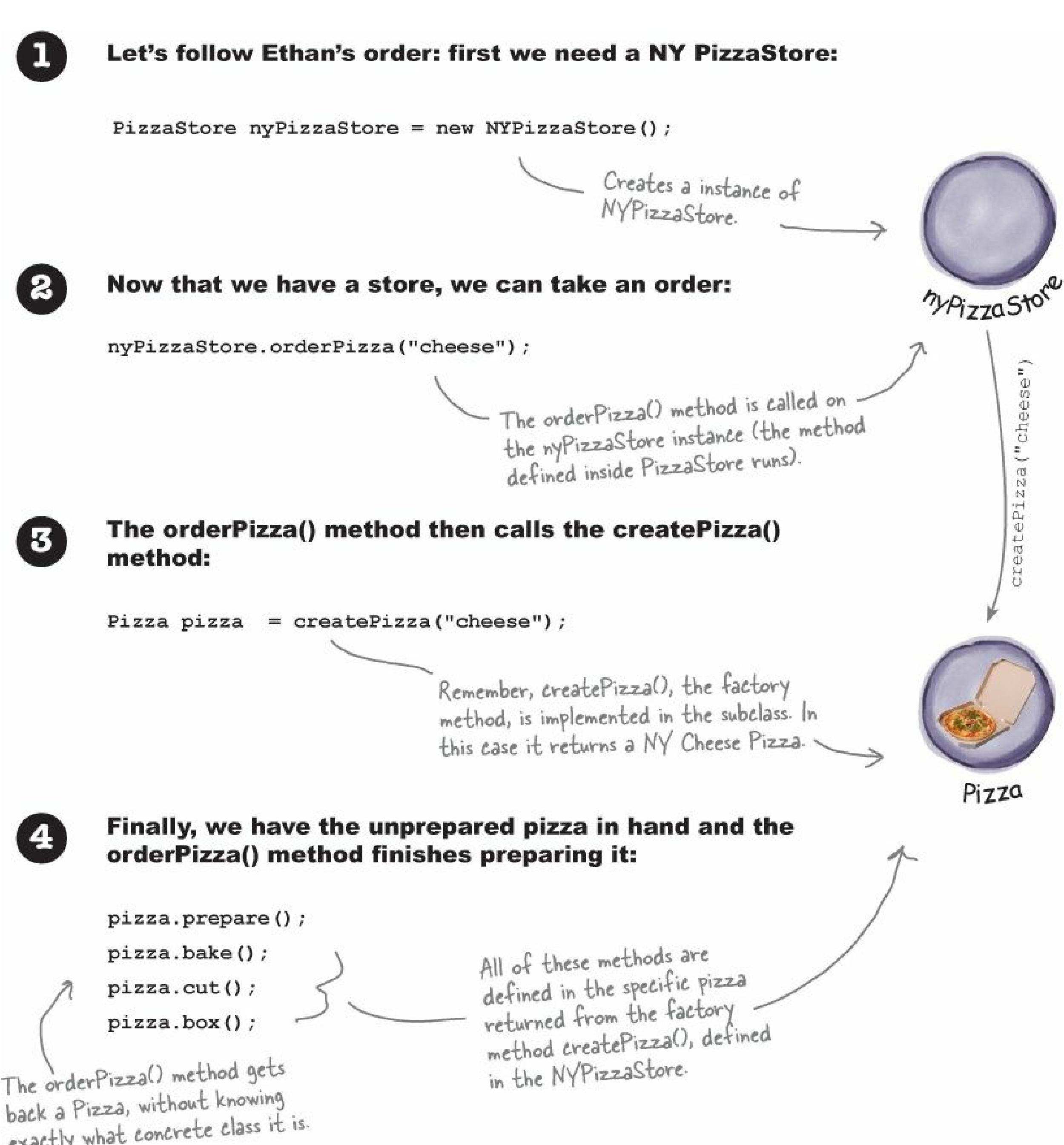
Let’s see how it works: ordering pizzas with the pizza factory method

So how do they order?

① First, Joel and Ethan need an instance of a PizzaStore. Joel needs to instantiate a ChicagoPizzaStore and Ethan needs a NYPizzaStore.
② With a PizzaStore in hand, both Ethan and Joel call the orderPizza() method and pass in the type of pizza they want (cheese, veggie, and so on).
③ To create the pizzas, the createPizza() method is called, which is defined in the two subclasses NYPizzaStore and ChicagoPizzaStore. As we defined them, the NYPizzaStore instantiates a NY style pizza, and the ChicagoPizzaStore instantiates a Chicago style pizza. In either case, the Pizza is returned to the orderPizza() method.
④ The orderPizza() method has no idea what kind of pizza was created, but it knows it is a pizza and it prepares, bakes, cuts, and boxes it for Ethan and Joel.

Let’s check out how these pizzas are really made to order...

Behind the Scenes



We're just missing one thing: PIZZA!

Our PizzaStore isn’t going to be very popular without some pizzas, so let’s implement them

We'll start with an abstract Pizza class and all the concrete pizzas will derive from this.

Each Pizza has a name, a type of dough, a type of sauce, and a set of toppings.

The abstract class provides some basic defaults for baking, cutting and boxing.

Preparation follows a number of steps in a particular sequence.

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList<String> toppings = new ArrayList<String>();

    void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for (String topping : toppings) {
            System.out.println("   " + topping);
        }
    }

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

    public String getName() {
        return name;
    }
}
```

NOTE

REMEMBER: we don't provide import and package statements in the code listings. Get the complete source code from the wickedlysmart website. You'll find the URL on page xxxiii in the Intro.

Now we just need some concrete subclasses... how about defining New York and Chicago style cheese pizzas?

```java
public class NYStyleCheesePizza extends Pizza {

    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
    }
}
```

The NY Pizza has its own marinara style sauce and thin crust.

And one topping, reggiano cheese!

```java
public class ChicagoStyleCheesePizza extends Pizza {

    public ChicagoStyleCheesePizza() {
        name = "Chicago Style Deep Dish Cheese Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";

        toppings.add("Shredded Mozzarella Cheese");
    }

    void cut() {
        System.out.println("Cutting the pizza into square slices");
    }
}
```

The Chicago Pizza uses plum tomatoes as a sauce along with extra-thick crust.

The Chicago style deep dish pizza has lots of mozzarella cheese!

The Chicago style pizza also overrides the cut() method so that the pieces are cut into squares.

You've waited long enough. Time for some pizzas!

```
public class PizzaTestDrive {

    public static void main(String[] args) {
        PizzaStore nyStore = new NYPizzaStore();
        PizzaStore chicagoStore = new ChicagoPizzaStore();

        Pizza pizza = nyStore.orderPizza("cheese");
        System.out.println("Ethan ordered a " + pizza.getName() + "\n");

        pizza = chicagoStore.orderPizza("cheese");
        System.out.println("Joel ordered a " + pizza.getName() + "\n");
    }
}
```

First we create two different stores.

Then use one one store to make Ethan's order.

And the other for Joel's.

```
File Edit Window Help YouWantMootzOnThatPizza?
%java PizzaTestDrive

Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Grated Reggiano cheese
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a NY Style Sauce and Cheese Pizza

Preparing Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Shredded Mozzarella Cheese
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
```

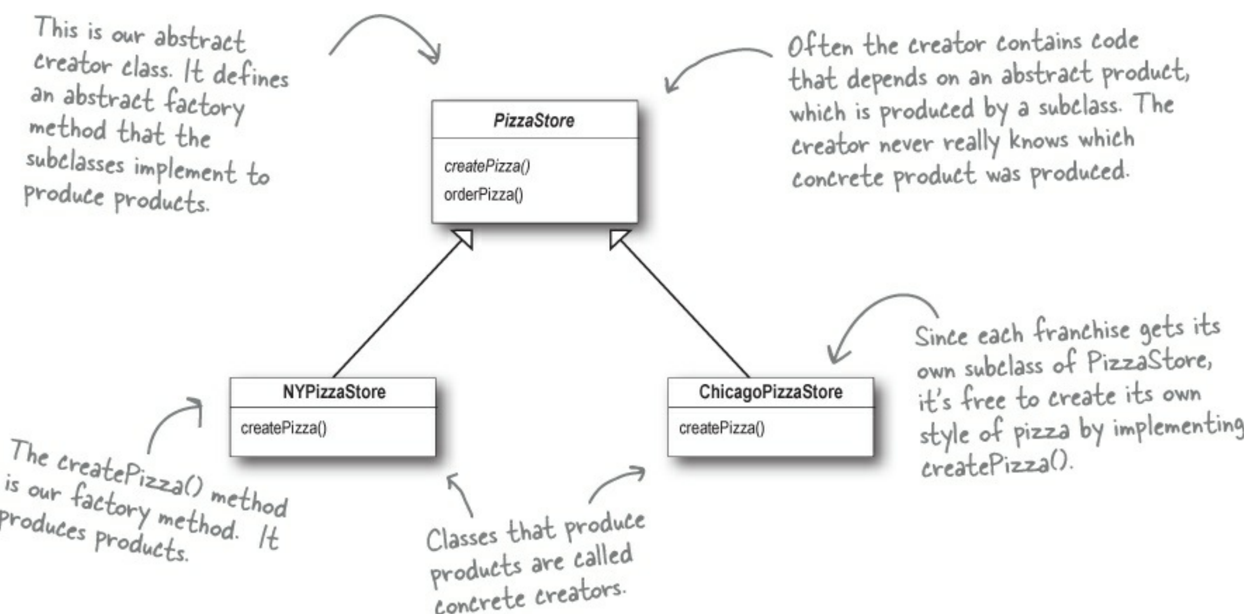
Both pizzas get prepared, the toppings added, and the pizzas baked, cut and boxed. Our superclass never had to know the details, the subclass handled all that just by instantiating the right pizza.

It's finally time to meet the Factory Method Pattern

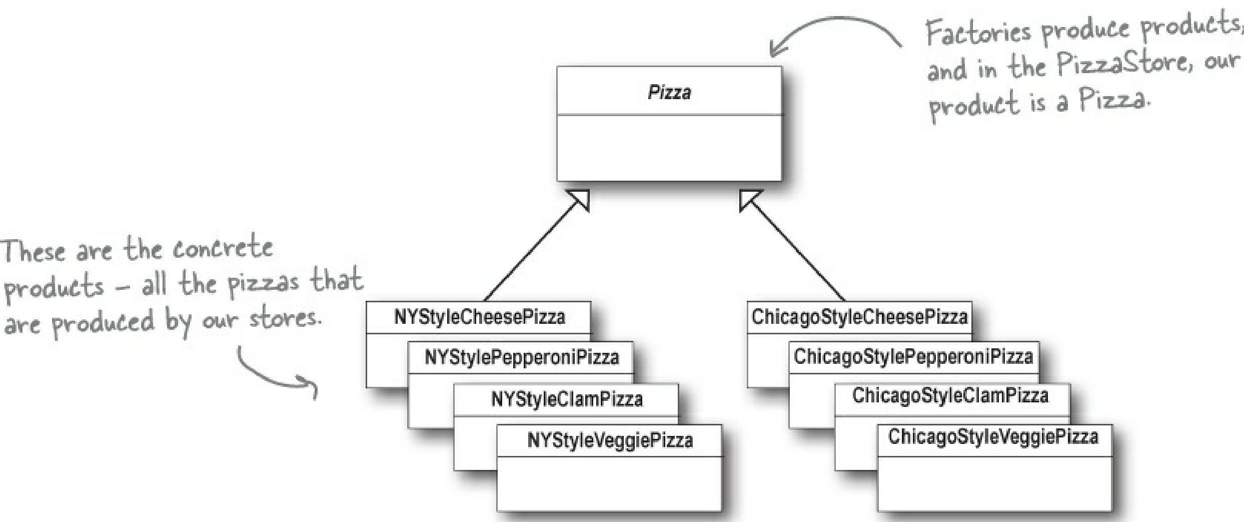
All factory patterns encapsulate object creation. The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create. Let's check out these class diagrams to see who the players are in this

pattern:

The Creator classes



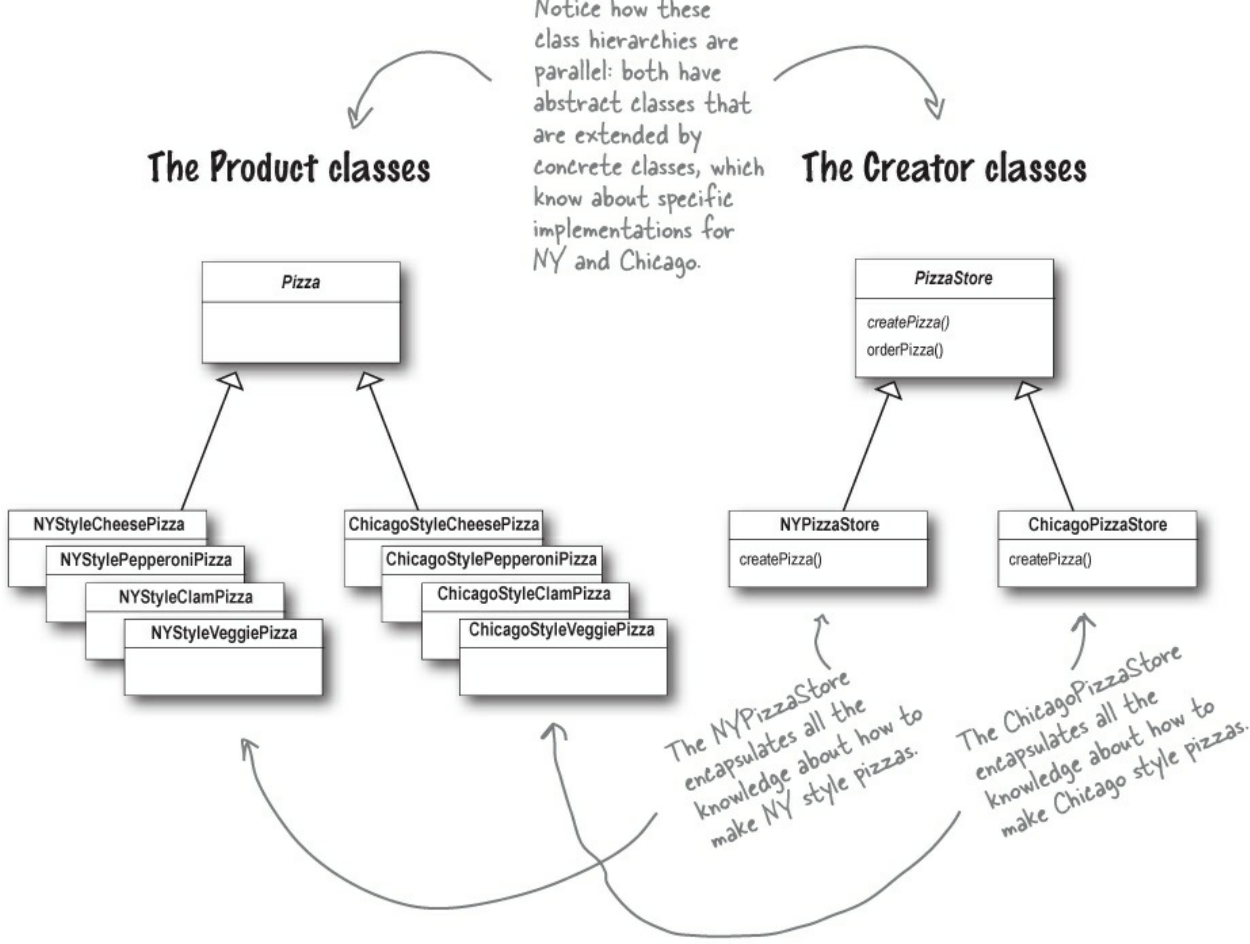
The Product classes



Another perspective: parallel class hierarchies

We've seen that the factory method provides a framework by supplying an orderPizza() method that is combined with a factory method. Another way to look at this pattern as a framework is in the way it encapsulates product knowledge into each creator.

Let’s look at the two parallel class hierarchies and see how they relate:

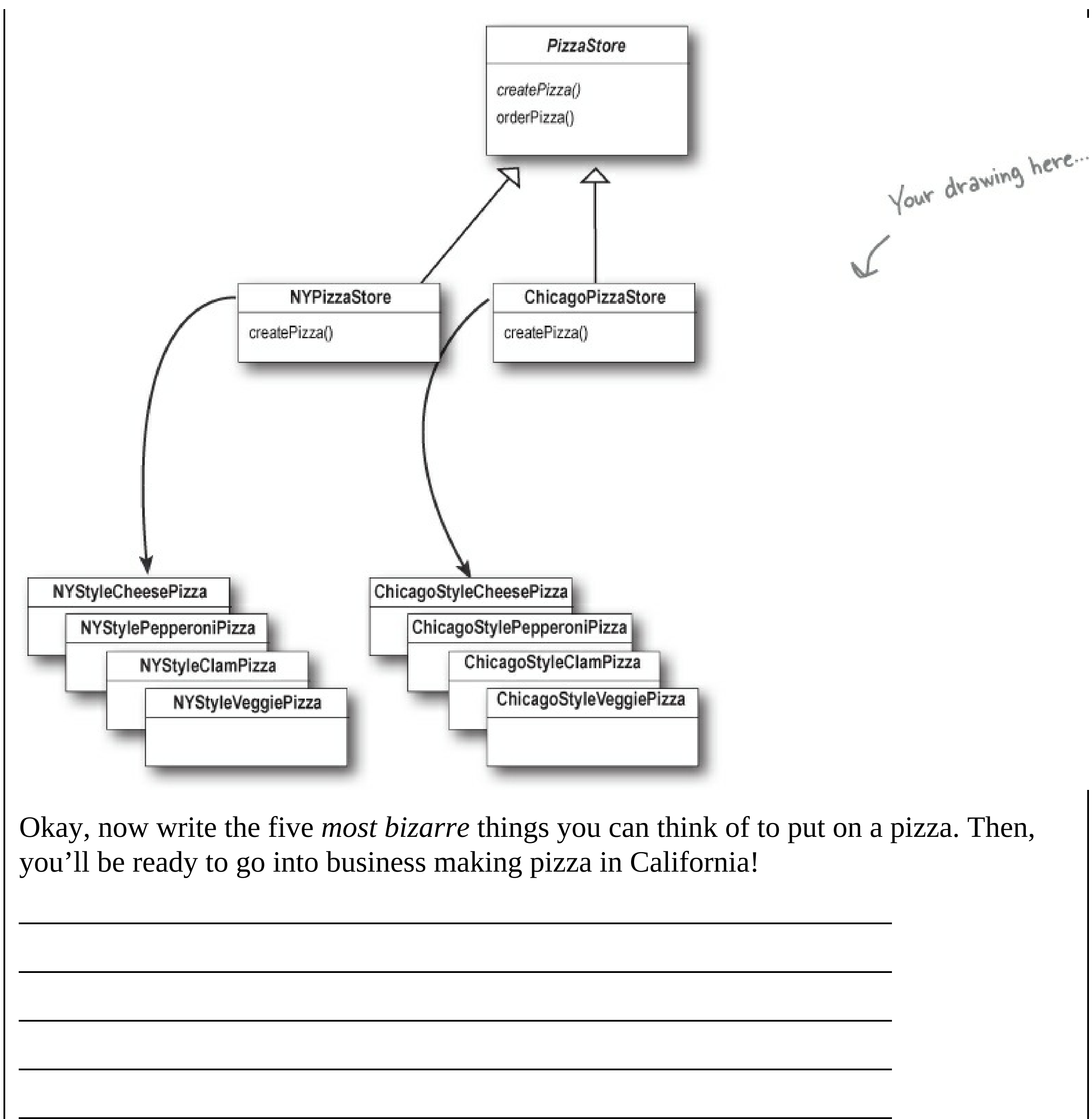


NOTE

The factory method is the key to encapsulating this knowledge.

DESIGN PUZZLE

We need another kind of pizza for those crazy Californians (crazy in a *good* way, of course). Draw another parallel set of classes that you’d need to add a new California region to our PizzaStore.



Okay, now write the five *most bizarre* things you can think of to put on a pizza. Then, you'll be ready to go into business making pizza in California!

____

____

____

____

____

Factory Method Pattern defined

It's time to roll out the official definition of the Factory Method Pattern:

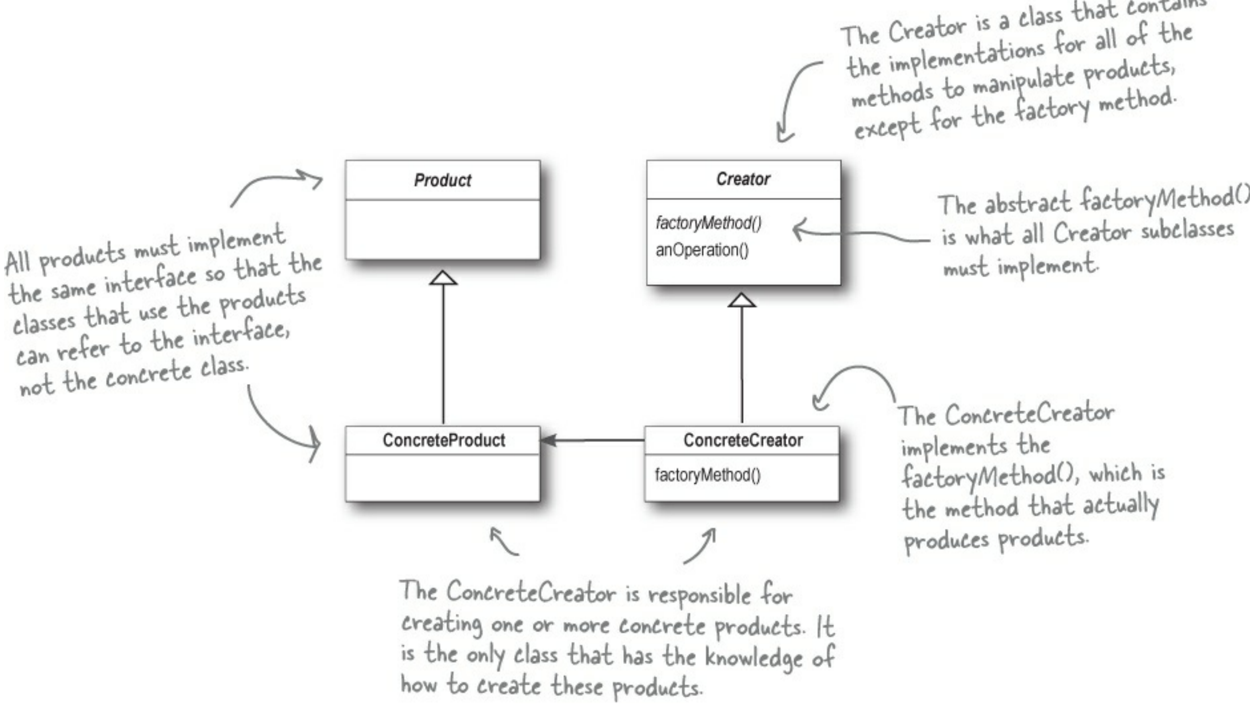
> **NOTE**
>
> **The Factory Method Pattern** defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

As with every factory, the Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. Looking at the class diagram below, you can see that the abstract Creator gives you an interface with a method for creating objects, also known as the “factory method.” Any other methods implemented in the abstract Creator are written to operate on products produced by the factory method. Only subclasses actually implement the factory method and create products.

As in the official definition, you’ll often hear developers say that the Factory Method lets subclasses decide which class to instantiate. They say “decide” not because the pattern allows subclasses themselves to decide at runtime, but because the creator class is written without knowledge of the actual products that will be created, which is decided purely by the choice of the subclass that is used.

NOTE

You could ask them what “decides” means, but we bet you now understand this better than they do!



THERE ARE NO DUMB QUESTIONS

Q: **Q: What’s the advantage of the Factory Method Pattern when you only have one ConcreteCreator?**

A: **A:** The Factory Method Pattern is useful if you’ve only got one concrete creator because you are decoupling the implementation of the product from its use. If you add additional products or change a product’s implementation, it will not affect your Creator (because the Creator is not tightly coupled to any ConcreteProduct).

Q: **Q: Would it be correct to say that our NY and Chicago stores are implemented using Simple Factory? They look just like it.**

A: **A:** They’re similar, but used in different ways. Even though the implementation of each concrete store looks a lot like the SimplePizzaFactory, remember that the concrete stores are extending a class that has defined createPizza() as an abstract method. It is up to each store to define the behavior of the createPizza() method. In Simple Factory, the factory is another object that is composed with the PizzaStore.

Q: **Q: Are the factory method and the Creator always abstract?**

A: **A:** No, you can define a default factory method to produce some concrete product. Then you always have a means of creating products even if there are no subclasses of the Creator.

Q: **Q: Each store can make four different kinds of pizzas based on the type passed in. Do all concrete creators make multiple products, or do they sometimes just make one?**

A: **A:** We implemented what is known as the parameterized factory method. It can make more than one object based on a parameter passed in, as you noticed. Often, however, a factory just produces one object and is not parameterized. Both are valid forms of the pattern.

Q: **Q: Your parameterized types don’t seem “type-safe.” I’m just passing in a String! What if I asked for a “CalmPizza”?**

A: **A:** You are certainly correct and that would cause, what we call in the business, a “runtime error.” There are several other more sophisticated techniques that can be used to make parameters more “type safe,” or, in other words, to ensure errors in parameters can be caught at compile time. For instance, you can create objects that represent the parameter types, use static constants, or use enums.

Q: **Q: I’m still a bit confused about the difference between Simple Factory and Factory Method. They look very similar, except that in Factory Method, the class that returns the pizza is a subclass. Can you explain?**

A: **A:** You’re right that the subclasses do look a lot like Simple Factory; however, think of Simple Factory as a one-shot deal, while with Factory Method you are creating a framework that lets the subclasses decide which implementation will be used. For example, the orderPizza() method in the Factory Method provides a general framework for creating pizzas that relies on a factory method to actually create the concrete classes that go into making a pizza. By subclassing the PizzaStore class, you decide what concrete products go into making the pizza that orderPizza() returns. Compare that with SimpleFactory, which gives you a way to encapsulate object creation, but doesn’t give you the flexibility of the Factory Method because there is no way to vary the products you’re creating.

MASTER AND STUDENT...

Master: *Grasshopper, tell me how your training is going.*

Student: *Master, I have taken my study of “encapsulate what varies” further.*

Master: *Go on...*

Student: *I have learned that one can encapsulate the code that creates objects. When you have code that instantiates concrete classes, this is an area of frequent change. I’ve*

learned a technique called “factories” that allows you to encapsulate this behavior of instantiation.

Master: *And these “factories,” of what benefit are they?*

Student: *There are many. By placing all my creation code in one object or method, I avoid duplication in my code and provide one place to perform maintenance. That also means clients depend only upon interfaces rather than the concrete classes required to instantiate objects. As I have learned in my studies, this allows me to program to an interface, not an implementation, and that makes my code more flexible and extensible in the future.*

Master: *Yes Grasshopper, your OO instincts are growing. Do you have any questions for your master today?*

Student: *Master, I know that by encapsulating object creation I am coding to abstractions and decoupling my client code from actual implementations. But my factory code must still use concrete classes to instantiate real objects. Am I not pulling the wool over my own eyes?*

Master: *Grasshopper, object creation is a reality of life; we must create objects or we will never create a single Java program. But, with knowledge of this reality, we can design our code so that we have corralled this creation code like the sheep whose wool you would pull over your eyes. Once corralled, we can protect and care for the creation code. If we let our creation code run wild, then we will never collect its “wool.”*

Student: *Master, I see the truth in this.*

Master: *As I knew you would. Now, please go and meditate on object dependencies.*

A very dependent PizzaStore

SHARPEN YOUR PENCIL

Let’s pretend you’ve never heard of an OO factory. Here’s a version of the PizzaStore that doesn’t use a factory; make a count of the number of concrete pizza objects this class is dependent on. If you added California style pizzas to this PizzaStore, how many objects would it be dependent on then?

```
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Handles all the NY style pizzas

Handles all the Chicago style pizzas

You can write your answers here: ________ number ________ number with California too

Looking at object dependencies

When you directly instantiate an object, you are depending on its concrete class. Take a look at our very dependent PizzaStore one page back. It creates all the pizza objects right in the PizzaStore class instead of delegating to a factory.

If we draw a diagram representing that version of the PizzaStore and all the objects it depends on, here’s what it looks like:

This version of the PizzaStore depends on all those pizza objects, because it's creating them directly.

If the implementation of these classes changes, then we may have to modify in PizzaStore.

Because any changes to the concrete implementations of pizzas affects the PizzaStore, we say that the PizzaStore "depends on" the pizza implementations.

PizzaStore

NYStyleCheesePizza

NYStyleVeggiePizza

NYStylePepperoniPizza

NYStyleClamPizza

ChicagoStyleVeggiePizza

ChicagoStylePepperoniPizza

ChicagoStyleClamPizza

ChicagoStyleCheesePizza

Every new kind of pizza we add creates another dependency for PizzaStore.

The Dependency Inversion Principle

It should be pretty clear that reducing dependencies to concrete classes in our code is a "good thing." In fact, we've got an OO design principle that formalizes this notion; it even has a big, formal name: *Dependency Inversion Principle*.

> **NOTE**
>
> Yet another phrase you can use to impress the execs in the room! Your raise will more than offset the cost of this book, and you'll gain the admiration of your fellow developers.

Here's the general principle:

DESIGN PRINCIPLE

Depend upon abstractions. Do not depend upon concrete classes.

At first, this principle sounds a lot like “Program to an interface, not an implementation,” right? It is similar; however, the Dependency Inversion Principle makes an even stronger statement about abstraction. It suggests that our high-level components should not depend on our low-level components; rather, they should *both* depend on abstractions.

NOTE

A “high-level” component is a class with behavior defined in terms of other, “low-level” components.

For example, PizzaStore is a high-level component because its behavior is defined in terms of pizzas - it creates all the different pizza objects, and prepares, bakes, cuts, and boxes them, while the pizzas it uses are low-level components.

But what the heck does that mean?

Well, let’s start by looking again at the pizza store diagram on the previous page. PizzaStore is our “high-level component” and the pizza implementations are our “low-level components,” and clearly the PizzaStore is dependent on the concrete pizza classes.

Now, this principle tells us we should instead write our code so that we are depending on abstractions, not concrete classes. That goes for both our high-level modules and our low-level modules.

But how do we do this? Let’s think about how we’d apply this principle to our Very Dependent PizzaStore implementation...

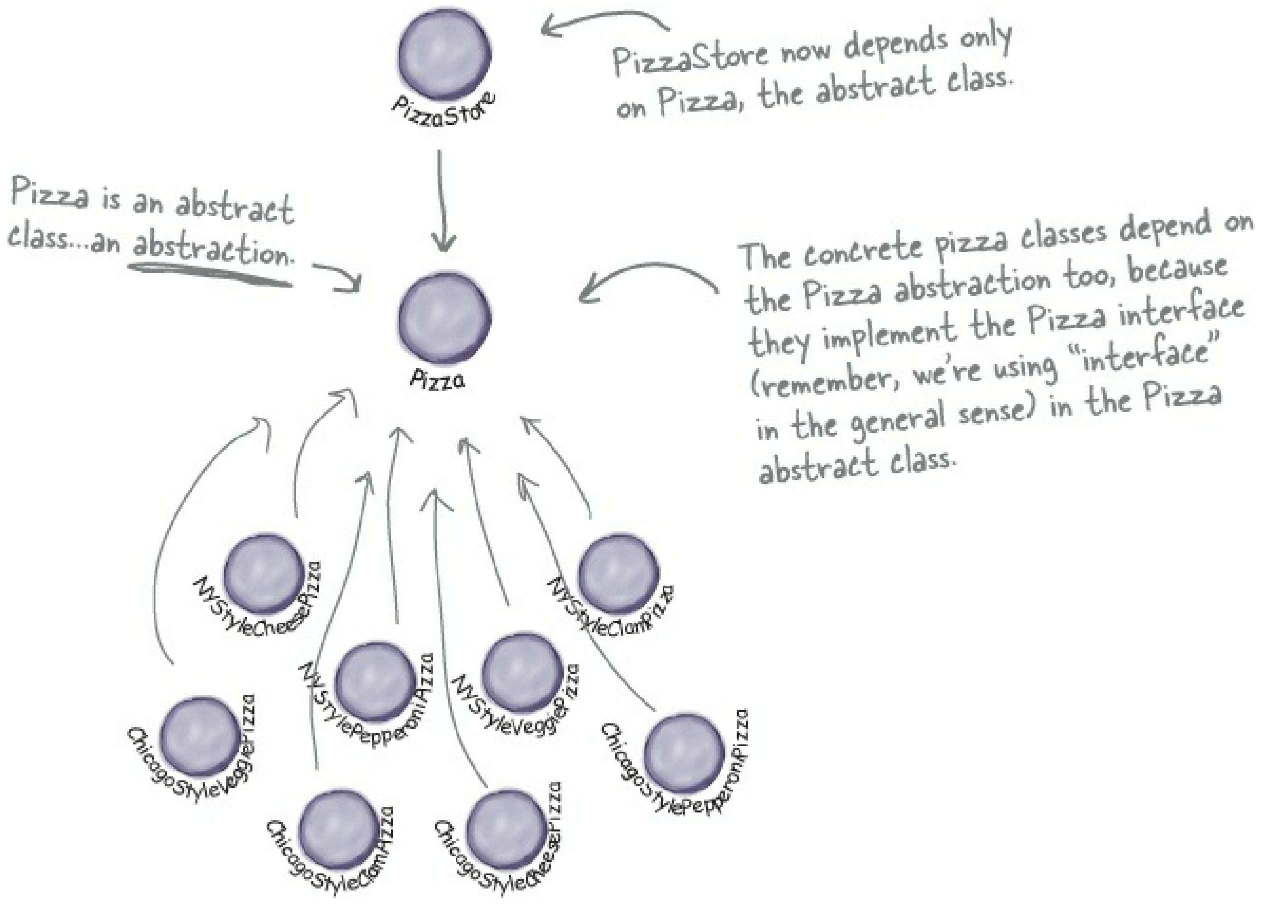
Applying the Principle

Now, the main problem with the Very Dependent PizzaStore is that it depends on every type of pizza because it actually instantiates concrete types in its orderPizza() method.

While we’ve created an abstraction, Pizza, we’re nevertheless creating concrete Pizzas in this code, so we don’t get a lot of leverage out of this abstraction.

How can we get those instantiations out of the orderPizza() method? Well, as we know, the Factory Method allows us to do just that.

So, after we’ve applied the Factory Method, our diagram looks like this:



After applying the Factory Method, you’ll notice that our high-level component, the PizzaStore, and our low-level components, the pizzas, both depend on Pizza, the abstraction. Factory Method is not the only technique for adhering to the Dependency Inversion Principle, but it is one of the more powerful ones.

Where's the "inversion" in Dependency Inversion Principle?

The "inversion" in the name Dependency Inversion Principle is there because it inverts the way you typically might think about your OO design. Look at the diagram on the previous page. Notice that the low-level components now depend on a higher level abstraction. Likewise, the high-level component is also tied to the same abstraction. So, the top-to-bottom dependency chart we drew a couple of pages back has inverted itself, with both high-level and low-level modules now depending on the abstraction.

Let's also walk through the thinking behind the typical design process and see how introducing the principle can invert the way we think about the design...

Inverting your thinking...